Grammars en Parsing

Principes

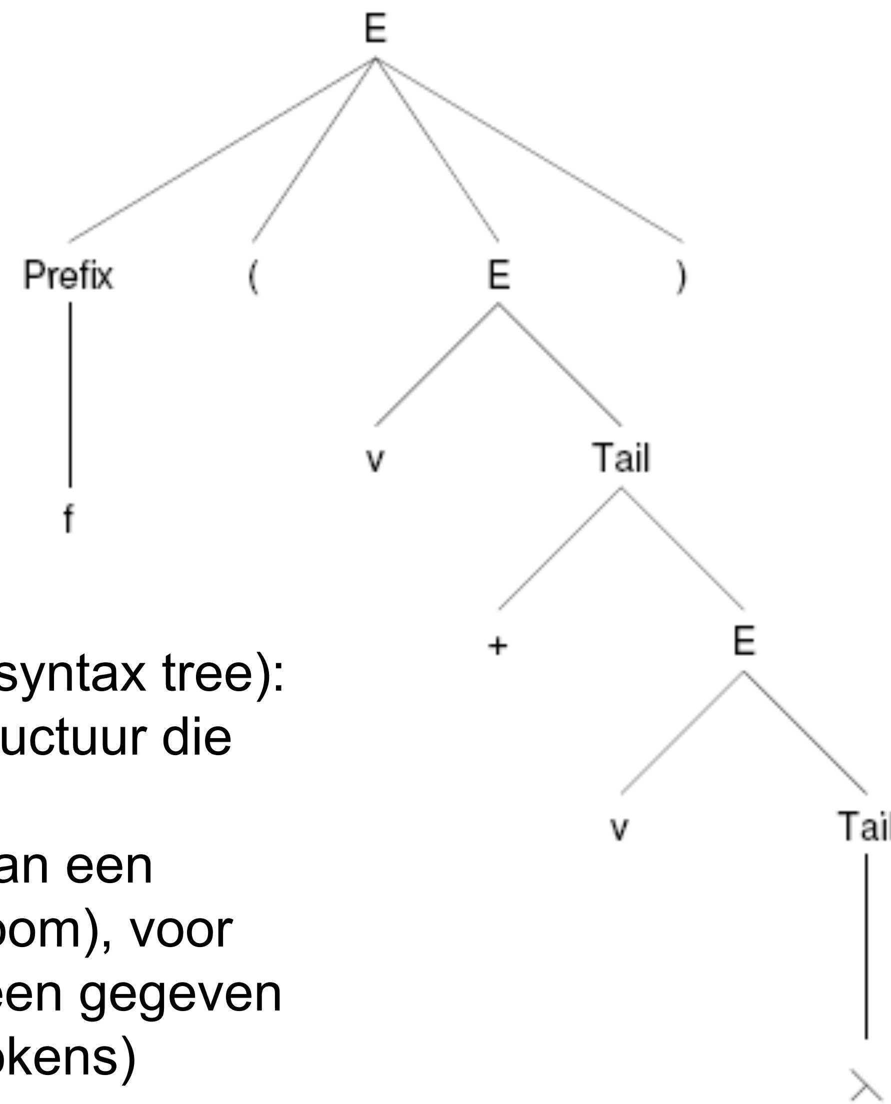
Grammar flow analyse

Context-free grammars (eventueel in BNF)

- Definiëren de structuur van geldige programma's
- Drukken de gewenste betekenis uit.

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Figure 4.1: A simple expression grammar.



Parse tree (of abstract syntax tree):
de belangrijkste hulpstructuur die
de compiler gebruikt
Parsing = constructie van een
parse tree (afleidingsboom), voor
een gegeven CFG en een gegeven
invoerwoord (=rij van tokens)

Figure 4.2: The parse tree for f (v + v) .

Dubbelzinnigheid is ongewenst: de boom moet de juiste betekenis uitdrukken
Niet elke CFG is geschikt voor elke parsing methode.

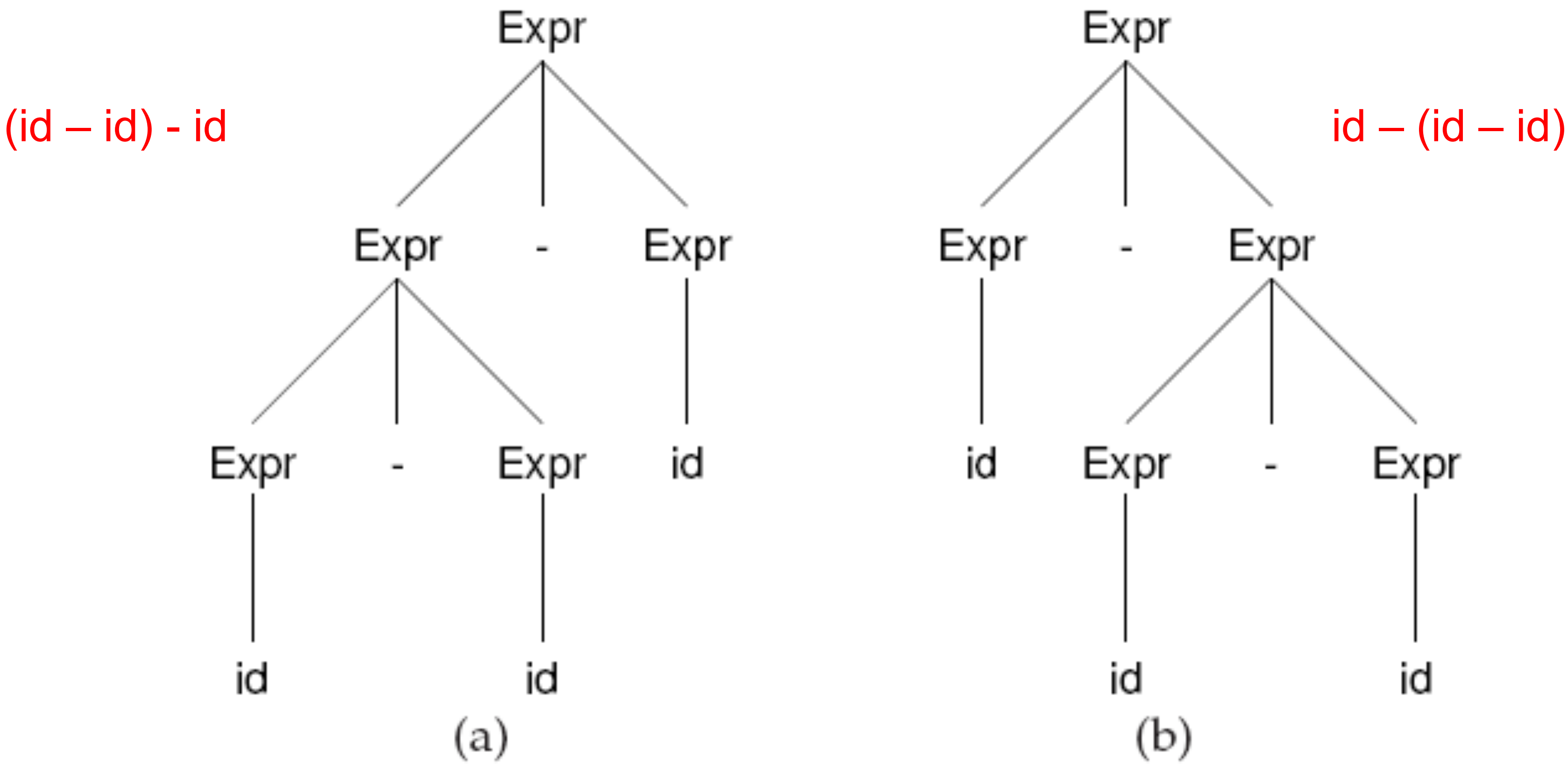


Figure 4.3: Two parse trees for id - id - id.

Voor de leesbaarheid gebruikt men vaak de Backus-Naur form, die kan gemakkelijk omgezet worden naar “echte” CFG –vorm zonder extra constructies

foreach $p \in Prods$ of the form “$\mathsf{A} \rightarrow \alpha \, [\, X_1 \ldots X_n \,] \, \beta$” **do**
  $N \leftarrow$ NEWNONTERM( )
  $p \leftarrow$ “$\mathsf{A} \rightarrow \alpha \; N \; \beta$”
  $Prods \leftarrow Prods \cup \{$“$N \rightarrow X_1 \ldots X_n$”$\}$
  $Prods \leftarrow Prods \cup \{$“$N \rightarrow \lambda$”$\}$
foreach $p \in Prods$ of the form “$\mathsf{B} \rightarrow \gamma \, \{\, X_1 \ldots X_m \,\} \, \delta$” **do**
  $M \leftarrow$ NEWNONTERM( )
  $p \leftarrow$ “$\mathsf{B} \rightarrow \gamma \; M \; \delta$”
  $Prods \leftarrow Prods \cup \{$“$M \rightarrow X_1 \ldots X_n \; M$”$\}$
  $Prods \leftarrow Prods \cup \{$“$M \rightarrow \lambda$”$\}$

[ ]: optioneel

{ }: iteratie

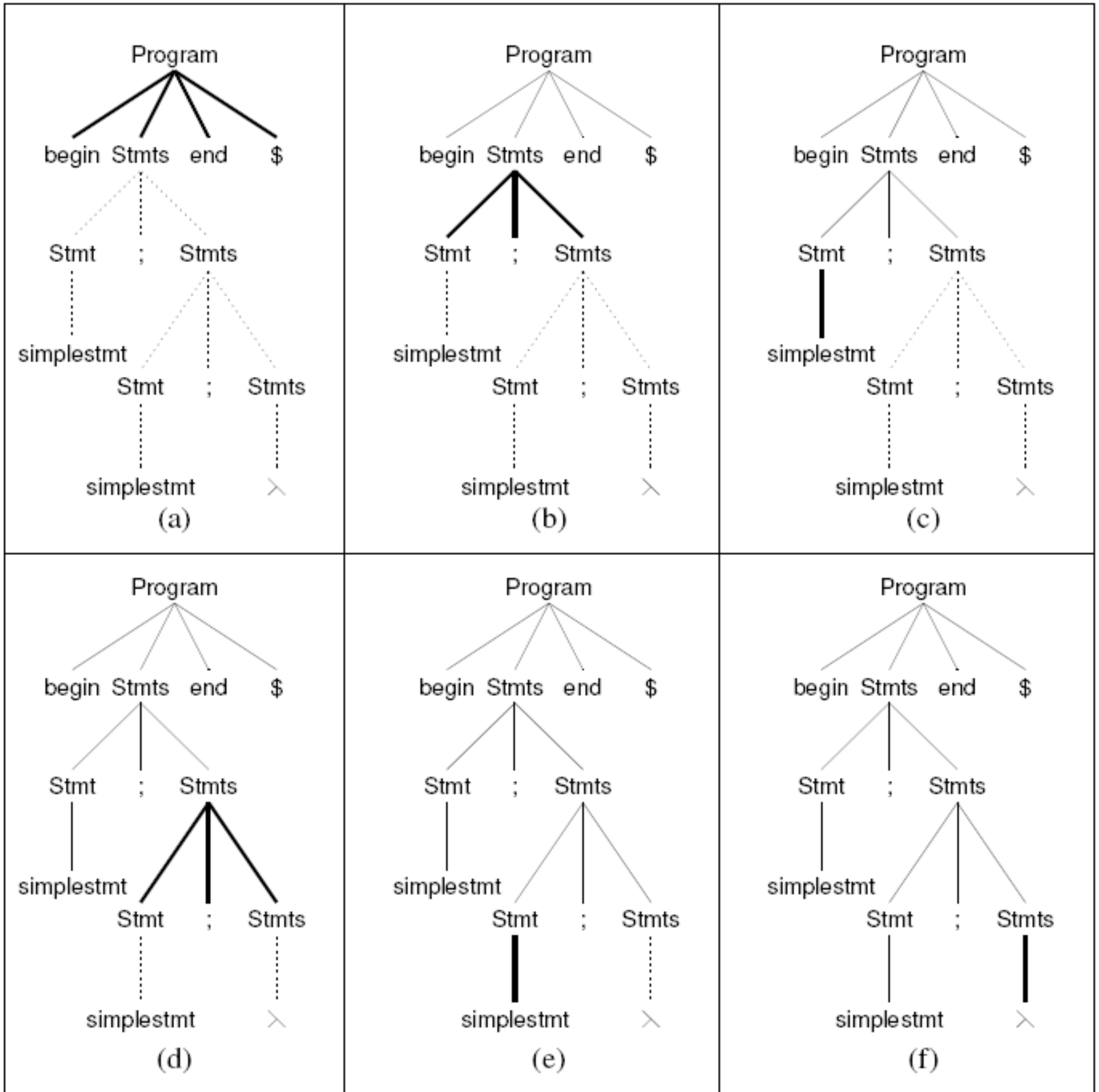
Figure 4.4: Algorithm to transform a BNF grammar into standard form.

Parsing (constructie van de parse tree):
ofwel top-down ofwel bottom-up

Voorbeeld:

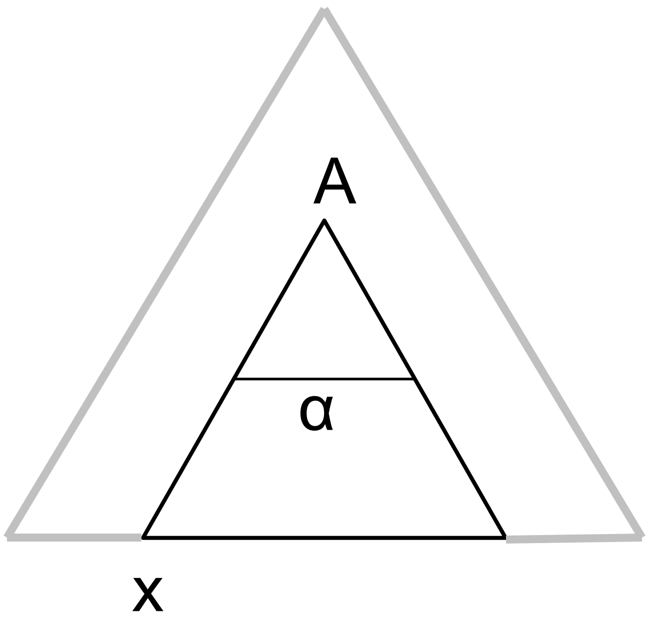
$$
\begin{aligned}
\text{Program} &\rightarrow \text{begin Stmts end \$} \\
\text{Stmts} &\rightarrow \text{Stmt ; Stmts} \\
&\quad\mid \lambda \\
\text{Stmt} &\rightarrow \text{simplestmt} \\
&\quad\mid \text{begin Stmts end}
\end{aligned}
$$

Top-down parse van “begin simplestmt ; simplestmt ; end $”



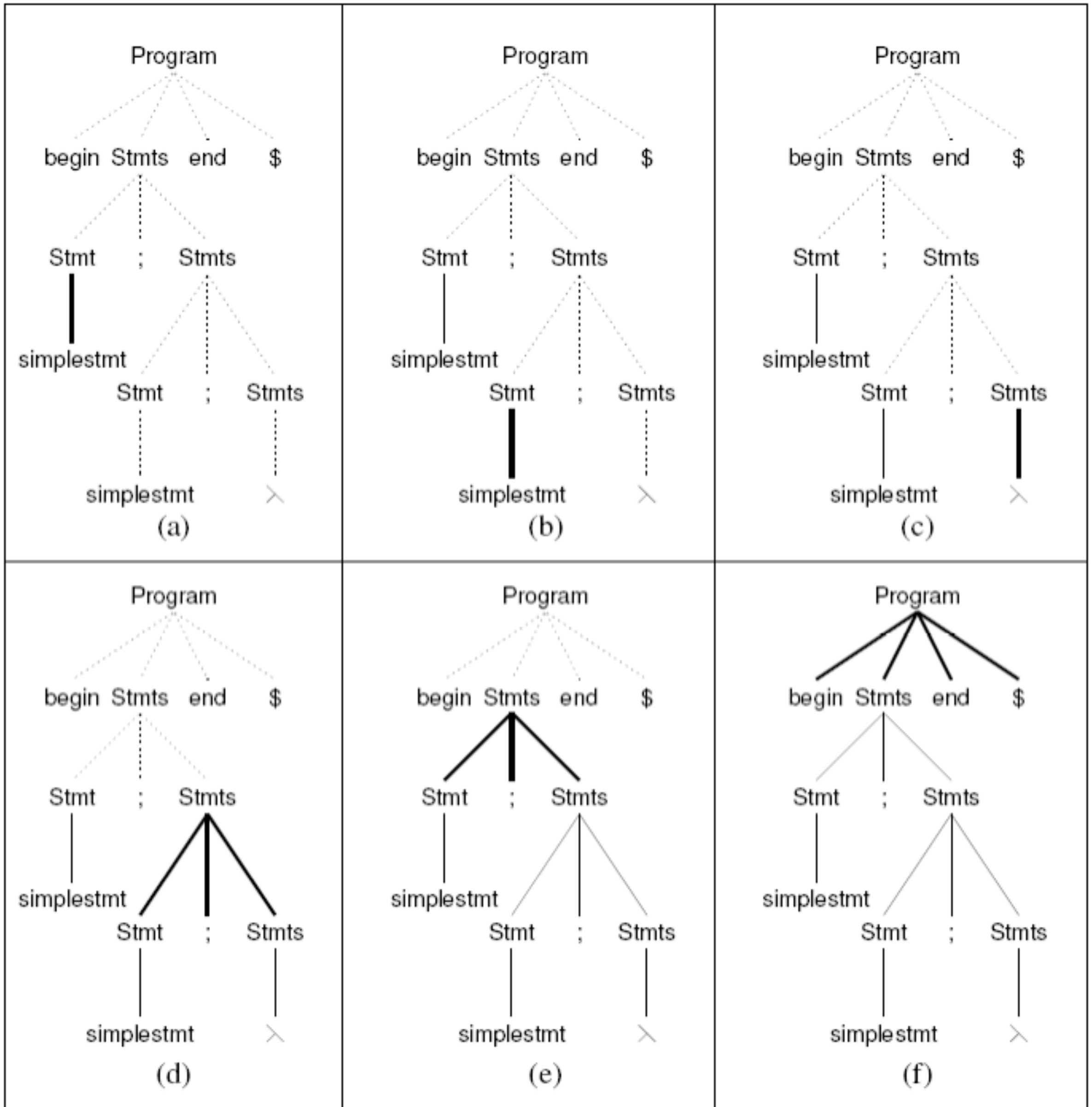
- De “recursive descent” methode (zie voorbeeld uit inleiding) was top-down
- Belangrijkste probleem: hoe kiezen tussen de alternatieven voor een nonterminaal A? Test of het volgende invoersymbool x kan voorkomen als eerste symbool van een woord afgeleid van het gekozen alternatief α.

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



In het voorbeeld, stap(c): de input simplestmt zorgt ervoor dat het 1e alternatief gekozen wordt voor Stmt

Bottom-up parse van "begin simplestmt ; simplestmt ; end $"



- Belangrijkste probleem: herkennen van de rechterkant van een productie, in een sentential form (= woord met zowel terminale als niet-terminale symbolen), om dan een “reductie” stap te kunnen uitvoeren.
- Je hebt meer informatie wanneer je de keuzes moet maken: de bottom-up methode is krachtiger (werkt voor meer grammatica’s) dan de top-down methode.

Algoritmen voor Grammar Analyse

Bij het gebruik van CFGs en parsing algoritmen gebruiken we een aantal eigenschappen van CFGs en hun symbolen die op voorhand kunnen berekend worden (m.a.w. ze hangen enkel af van de CFG, niet van het te parsen woord)

- Kan er van een gegeven symbool wel een terminaal woord afgeleid worden (= productiveness)
- Kan een gegeven symbool wel voorkomen in een woord afgeleid van het beginsymbool? (= reachability)

- Kan het lege woord afgeleid worden van een gegeven nonterminaal symbool?
- Welke terminale symbolen kunnen voorkomen als eerste symbool van een woord afgeleid van een gegeven woord α ? Notatie: First(α)
- Welke symbolen kunnen volgen op een gegeven nonterminal A ? Notatie: Follow(A)

Productiveness: kan van een nonterminal A een terminaal woord (inclusief het lege) afgeleid worden?

Initializeer W (worklist met nonterminals): initieel de nonterminals die LHS zijn van een productie zonder nonterminals in de RHS
count[r]: initieel het aantal nonterminals in de RHS van de productie r
Productive: initieel leeg.

```
while nonempty(W) {
    X := head(W); W := tail(W);
    if (X not in Productive) {
        Productive := Productive ∪ {X};
        forall (productions r having X in their RHS) {
            count[r]-- ;
            if (count[r] = 0)  W := (LHS(r)) :: W ;
            }
        }
}
```

Reachability: kan een nonterminal voorkomen in een woord afgeleid van het beginsymbool?

Reduceren van een CFG: laat eerst de niet-productieve symbolen weg, samen met de producties waarin ze voorkomen, en doe dan hetzelfde met onbereikbare symbolen (Niet andersom!)

```
procedure DerivesEmptyString( )
    foreach A ∈ NonTerminals( ) do
        SymbolDerivesEmpty(A) ← false
    foreach p ∈ Productions( ) do
        RuleDerivesEmpty(p) ← false
        Count(p) ← 0
        foreach X ∈ RHS(p) do Count(p) ← Count(p) + 1
        call CheckForEmpty(p)
    foreach X ∈ WorkList do
        WorkList ← WorkList − { X }
        foreach x ∈ Occurrences(X) do
            p ← Production(x)
            Count(p) ← Count(p) − 1
            call CheckForEmpty(p)
end
procedure CheckForEmpty(p)
    if Count(p) = 0
    then
        RuleDerivesEmpty(p) ← true
        A ← LHS(p)
        if not SymbolDerivesEmpty(A)
        then
            SymbolDerivesEmpty(A) ← true
            WorkList ← WorkList ∪ { A }
end
```

Initializeer Count(p) voor elke productie p: de lengte van de RHS van p

Bekijk alle occurrences van x in de RHS van een productie p: Count(p) vermindert voor elk van die occurrences

Check of het lege woord kan afgeleid worden van de RHS van p, dan kan dat uiteraard ook van de LHS van p

Figure 4.7: Algorithm for determining nonterminals and productions that can derive λ.

First(α) is de set van terminale symbolen die als eerste symbool voorkomen in een woord u dat van α kan afgeleid worden. α is een woord dat kan bestaan uit zowel terminale ls niet-terminale symbolen.

First(α) kan als volgt berekend worden.

function FIRST(α) **returns** *Set*

  foreach $\mathsf{A} \in$ NONTERMINALS( ) **do** *VisitedFirst*(A) $\leftarrow$ **false** (9)

  $ans \leftarrow$ INTERNALFIRST(α)

  return (ans)

end

function INTERNALFIRST($\mathcal{X}\beta$) **returns** *Set*

  if $\mathcal{X}\beta = \perp$ (10)

  then return ($\emptyset$)

  if $\mathcal{X} \in \Sigma$ (11)

  then return ($\{\mathcal{X}\}$)

  /⋆ $\mathcal{X}$ is a nonterminal. ⋆/ (12)

  $ans \leftarrow \emptyset$

  if not *VisitedFirst*($\mathcal{X}$)

  then

    VisitedFirst($\mathcal{X}$) $\leftarrow$ **true** (13)

    foreach $rhs \in$ *ProductionsFor*($\mathcal{X}$) **do**

      $ans \leftarrow ans \cup$ INTERNALFIRST(rhs) (14)

  if SymbolDerivesEmpty($\mathcal{X}$) (15)

  then $ans \leftarrow ans \cup$ INTERNALFIRST(β)

  return (ans) (16)

end

Figure 4.8: Algorithm for computing First(α).

Voorbeeld-grammatica:

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Figure 4.1: A simple expression grammar.

Level	First $\mathcal{X}$	β	*ans*	Marker	Done? (⋆=Yes)	Comment
			First(Tail)			
0	Tail	$\perp$	{ }	12		
1	+	E	{ + }	11	⋆	Tail → +E
1	$\perp$	$\perp$	{ }	10	⋆	Tail → λ
0			{ + }	14		After all rules for Tail
1	$\perp$	$\perp$	{ }	10	⋆	Since $\beta = \perp$
0			{ + }	15	⋆	Final answer
			First(Prefix)			
0	Prefix	$\perp$	{ }	12		
1	f	$\perp$	{ f }	11	⋆	Prefix → f
1	$\perp$	$\perp$	{ }	10	⋆	Prefix → λ
0			{ f }	14		After all rules for Prefix
1	$\perp$	$\perp$	{ }	10	⋆	Since $\beta = \perp$
0			{ f }	15	⋆	Final answer
			First(E)			
0	E	$\perp$	{ }	12		
1	Prefix	(E)	{ }	12		E → Prefix (E)
1			{ f }	16		Computation shown above
2	(	E)	{ (}	11	⋆	Since Prefix $\Rightarrow^{\star} \lambda$
1			{ f, (}	15	⋆	Results due to E → Prefix (E)
1	v	Tail	{ v }	11	⋆	E → v Tail
1	$\perp$	$\perp$	{ }	10		Since $\beta = \perp$
0			{ f, (, v }	15	⋆	Final answer

1 $S \rightarrow A\ B\ c$
2 $A \rightarrow a$
3 $\quad\mid\ \lambda$
4 $B \rightarrow b$
5 $\quad\mid\ \lambda$

Level	First $\mathcal{X}$	First β	*ans*	Marker	Done? (★=Yes)	Comment
				First (B)		
0	B	$\perp$	{ }	12		
1	b	$\perp$	{b}	11	★	$B \rightarrow b$
1	$\perp$	$\perp$	{ }	10	★	$B \rightarrow \lambda$
0			{b}	15	★	Final answer
				First (A)		
0	A	$\perp$	{ }	12		
1	a	$\perp$	{a}	11	★	$A \rightarrow a$
1	$\perp$	$\perp$	{ }	10	★	$A \rightarrow \lambda$
0			{a}	15	★	Final answer
				First (S)		
0	S	$\perp$	{ }	12		
1	A	B c	{a}	16		Computation shown above
2	B	c	{b}	16		Because $A \Rightarrow^{\star} \lambda$; computation shown above
3	c	$\perp$	{c}	11	★	Because $B \Rightarrow^{\star} \lambda$
2			{b,c}	15	★	
1			{a,b,c}	15	★	
0			{a,b,c}	15	★	

Follow(A) is, voor een nonterminal A, de set van terminale symbolen die op A kunnen volgen in een willekeurige sentential form, m.a.w. in een willekeurig woord dat van het startsymbool kan afgeleid worden.

Tail(a) is, voor een occurrence a in de RHS van een productie, het deel van die RHS dat volgt op a.

```
function Follow(A) returns Set
    foreach A ∈ NonTerminals() do
        VisitedFollow(A) ← false                                  (17)
    ans ← InternalFollow(A)
    return (ans)
end
function InternalFollow(A) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                        (18)
    then
        VisitedFollow(A) ← true                                   (19)
        foreach a ∈ Occurrences(A) do                             (20)
            ans ← ans ∪ First(Tail(a))                            (21)
            if AllDeriveEmpty(Tail(a))                            (22)
            then
                targ ← LHS(Production(a))
                ans ← ans ∪ InternalFollow(targ)                  (23)
    return (ans)                                                  (24)
end
function AllDeriveEmpty(γ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

Level	Rule	Marker	Result	Comment
		Follow (B)		
0				
0	$S \rightarrow A\ \underline{B}\ c$	21	{ c }	
0		24	{ c }	Returns
		Follow (A)		
0				
0	$S \rightarrow \underline{A}\ B\ c$	21	{ b,c }	
0		24	{ b,c }	Returns
		Follow (S)		
0				
0		24	{ }	Returns

Figure 4.12: Follow sets for the grammar in Figure 4.10. Note that Follow(S) = { } because S does not appear on the RHS of any production.

Level	Rule	Marker	Result	Comment
		F	(Prefix)	
0		F	(Prefix)	
0	E → <u>Prefix</u> (E)	21	{ (}	
		F	(E)	
0		F	(E)	
0	E → Prefix (<u>E</u>)	21	{) }	
0	Tail → + <u>E</u>	23	{ }	
1		F	(Tail)	
1	E → v <u>Tail</u>	23	{ }	
2		F	(E)	
		18	{ }	Recursion avoided
1		24	{ }	Returns
0		24	{) }	Returns
		F	(Tail)	
0		F	(Tail)	
0	E → v <u>Tail</u>	23	{ }	
1		F	(E)	
1	E → Prefix (<u>E</u>)	21	{) }	
1	Tail → + <u>E</u>	23	{ }	
2		F	(Tail)	
		18	{ }	Recursion avoided
1		24	{) }	Returns
0		24	{) }	Returns

Figure 4.13: Follow sets for the nonterminals of Figure 4.1.

Top-Down Parsing

Predict sets voor Top-down parsing: voor de keuze van een productie p onderscheiden we 2 gevallen: van de RHS kan het lege woord afgeleid worden, of niet.

function Predict($p : A \rightarrow X_1 \ldots X_m$) : *Set*
    $ans \leftarrow$ First($X_1 \ldots X_m$)
    if RuleDerivesEmpty(p)
    then
        $ans \leftarrow ans \cup$ Follow(A)
    return (ans)
end

Figure 5.1: Computation of Predict sets.

Voorbeeld

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

Figure 5.2: A CFGs.

Rule Number	N	$\mathcal{X}_1 \ldots \mathcal{X}_m$	First($\mathcal{X}_1 \ldots \mathcal{X}_m$)	Derives Empty?	Follow(N)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Predict sets mogen niet overlappen

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

```
procedure MATCH(ts, token)
    if ts.PEEK( ) = token
    then call ts.ADVANCE( )
    else call ERROR(Expected token)
end
```

Figure 5.5: Utility for matching tokens in an input stream.

Gebruik van de Predict sets

```
procedure A(ts)
    switch (...)
        case ts.PEEK() ∈ Predict(p1)
            /*  Code for p1                                  */
        case ts.PEEK() ∈ Predict(pi)
            /*  Code for p2                                  */
        /*  .                                                */
        /*  .                                                */
        /*  .                                                */
        case ts.PEEK() ∈ Predict(pn)
            /*  Code for pn                                  */
        case default
            /*  Syntax error                                 */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH($)
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH(c)
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }
            call MATCH(a)
            call B( )
            call C( )
            call MATCH(d)
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH(b)
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH(q)
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Gebruik in een recursive descent parser

Tabel-gestuurde LL(1) parser

```
procedure LLparser(ts)
    call push(S)
    accepted ← false
    while not accepted do
        if TOS() ∈ Σ
        then
            call match(ts, TOS())
            if TOS() = $
            then  accepted ← true
            call pop()
        else
            p ← LLtable[TOS(), ts.peek()]
            if p = 0
            then
                call error(Syntax error—no production applicable)
            else  call apply(p)
end
procedure apply(p : A → X_1 ... X_m)
    call pop()
    for i = m downto 1 do
        call push(X_i)
end
```

Figure 5.8: Generic LL(1) parser.

```
procedure FillTable(LLtable)
    foreach A ∈ N do
        foreach a ∈ Σ do LLtable[A][a] ← 0
    foreach A ∈ N do
        foreach p ∈ ProductionsFor(A) do
            foreach a ∈ Predict(p) do LLtable[A][a] ← p
end
```

Figure 5.9: Construction of an LL(1) parse table.

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. The blank entries should trigger error actions in the parser.

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S→AC$	
$CA		abbdc$
	Apply 4: A→aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B→bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B→bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B→λ	
$CdC		dc$
	Apply 3: C→λ	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C→c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left to right.

1	Stmt	→	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	→	StmtList ; Stmt
4		|	Stmt
5	Expr	→	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

---

procedure FACTOR( )
    foreach $\mathsf{A} \in N$ **do**
        $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
        while $|\alpha| > 0$ **do**
            $V \leftarrow$ **new** *NonTerminal*( )
            $Productions \leftarrow Productions \cup \{ \mathsf{A} \rightarrow \alpha V \}$
            foreach $p \in ProductionsFor(A) \mid RHS(p) = \alpha\beta_p$ **do** 13
                $Productions \leftarrow Productions - \{ p \}$
                $Productions \leftarrow Productions \cup \{ V \rightarrow \beta_p \}$
            $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
end

Figure 5.13: Factoring common prefixes.

1	Stmt	→	if Expr then StmtList V_1
2	V_1	→	endif
3		|	else StmtList endif
4	StmtList	→	StmtList ; Stmt
5		|	Stmt
6	Expr	→	var V_2
7	V_2	→	+ Expr
8		|	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.

procedure ELIMINATELEFTRECURSION( )
    foreach $\mathsf{A} \in N$ **do**
        if $\exists\, r \in \mathit{ProductionsFor}(\mathsf{A}) \mid \mathit{RHS}(r) = \mathsf{A}\alpha$
        then
            $X \leftarrow$ **new** *NonTerminal*( )
            $Y \leftarrow$ **new** *NonTerminal*( )
            foreach $p \in \mathit{ProductionsFor}(\mathsf{A})$ **do**
                if $p = r$
                then $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, \mathsf{A} \rightarrow X\ Y \,\}$ *and remove r*
                else $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, X \rightarrow \mathit{RHS}(p) \,\}$
            $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, Y \rightarrow \alpha Y,\ Y \rightarrow \lambda \,\}$
end

Figure 5.15: Eliminating left recursion.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		$\mid$	else StmtList endif
4	StmtList	$\rightarrow$	X Y
5	X	$\rightarrow$	Stmt
6	Y	$\rightarrow$	; Stmt Y
7		$\mid$	λ
8	Expr	$\rightarrow$	var V_2
9	V_2	$\rightarrow$	+ Expr
10		$\mid$	λ

Figure 5.16: LL(1) version of the grammar in Figure 5.14.

1	S	$\rightarrow$	Stmt \$
2	Stmt	$\rightarrow$	if expr then Stmt else Stmt
3		|	if expr then Stmt
4		|	other

Figure 5.17: Grammar for if-then-else.