acbobic GaTcTJata R \Rightarrow 2 aa T cc=) & a T c c aTcDaaRcc ^baa<u>R</u>cc R d R. aabRcc \Rightarrow aa*R*b*R*cc (RbR) R am bigoous aabbRcc \Rightarrow aa*R*b*R*b*R*cc ß \Rightarrow aab*R*b*R*cc 8 gabbbRcc TR \Rightarrow aab*R*b*R*b*R*cc aabbbcc T \rightarrow aTc aabb*R*b*R*cc \Rightarrow Usily the R \rightarrow vuenbigvous grammer aabbb*R*cc \Rightarrow $R \rightarrow$ (bR) \Rightarrow abbbcc U numbique ous " sue the embigrous grammer

2.2 **Definition 3.2** A symbol c is in FIRST(α) if and only if $\alpha \Rightarrow c\beta$ for some (possibly) empty) sequence β of grammar symbols, and $F(RST(N \rightarrow \alpha) = F(RST(\alpha))$. 2.3 **Definition 3.3** A sequence α of grammar symbols is Nullable (we write this as Nullable(α)) if and only if $\alpha \Rightarrow \varepsilon$, and α production $N \rightarrow \alpha$ is called Nullable if Nullable (a) Algorithm 3.4 *Nullable*(ε) true =*Nullable*(a) = false *Nullable*($\alpha\beta$) = $Nullable(\alpha) \land Nullable(\beta)$ = $Nullable(\alpha_1) \lor \ldots \lor Nullable(\alpha_n),$ Nullable(N)where the productions for N are $N \rightarrow \alpha_1, \quad \dots \quad , N \rightarrow \alpha_n$

Production	Nullable	
Production	Nullable	
$T \rightarrow R$	true	, because Ris Nullable
$T \rightarrow aTc$	false	, because R is Nullable , because a Tc is Nullable
$R \rightarrow$	true	Kot.
$R \rightarrow bR$		
$N \rightarrow DN$	Juise)
		· · ·

A	2 Igorithm 3.5						
	$FIRST(N) = FIRST(\alpha_1) \cup \ldots$	•	· · · · ·)			
	$N \rightarrow \alpha_1, \dots$ FIRST(T) = FIRST(R) \cup FIRST(aTc)	v			D		
	$= FIRST(R) \cup FIRST(a) = FIRST(R) \cup \{a\}$			\rightarrow	R aTc		
I	$FIRST(R) = FIRST() \cup FIRST(bR) - $ = $\emptyset \cup FIRST(b) - $ = {b}		R R	\rightarrow \rightarrow	b <i>R</i>		
	Fig. 2.16 Fixed-point	Nonterminal				l Iteration 2	Iteration 3
- 30	iteration for calculation of FIRST	T R	0		{a} {b}	{a,b} {b}	{a,b} {b}

Fig. 2.16 Fixed-point iteration for calculation of			on Iteration 1		
FIRST	T R	0	{a} {b}	{a,b} {b}	{a,b} {b}
Production FIRST					
$\begin{array}{ccc} T \to R & \text{[b]} \\ T \to aTc & \text{[a]} \end{array}$					
$R \rightarrow \emptyset \mathcal{A}$					
$R \rightarrow bR$ {b}					

We have so far simply chosen a nullable production if and only if no other choice is possible. This is, however, not always the right thing to do, so we must change the rule to say that we choose a production $N \rightarrow \alpha$ on symbol *c* if one of the two conditions below are satisfied:

1) $c \in FIRST(\alpha)$.

2) α is nullable and the sequence *Nc* can occur somewhere in a derivation starting from the start symbol of the grammar.

Definition 3.6 A terminal symbol a is in FOLLOW(N) if and only if there is a derivation from the start symbol S of the grammar such that $S \Rightarrow \alpha Na\beta$, where α and β are (possibly empty) sequences of grammar symbols.

C

To correctly handle end-of-string conditions, we want to detect if $S \Rightarrow \alpha N$, *i.e.*, if there are derivations where N can be followed by the end of input. It turns out to be easy to do this by adding an extra production to the grammar:

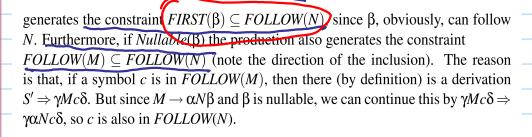
$S' \rightarrow S$

where *S'* is a new nonterminal that replaces *S* as start symbol and \$ is a new terminal symbol representing the end of input. Hence, in the new grammar, \$ will be in *FOLLOW(N)* exactly if $S' \Rightarrow \alpha N$ \$ which is the case exactly when $S \Rightarrow \alpha N$.

The easiest way to calculate *FOLLOW* is to generate a collection of *set constraints*, which are subsequently solved to find the least sets that obey the constraints.

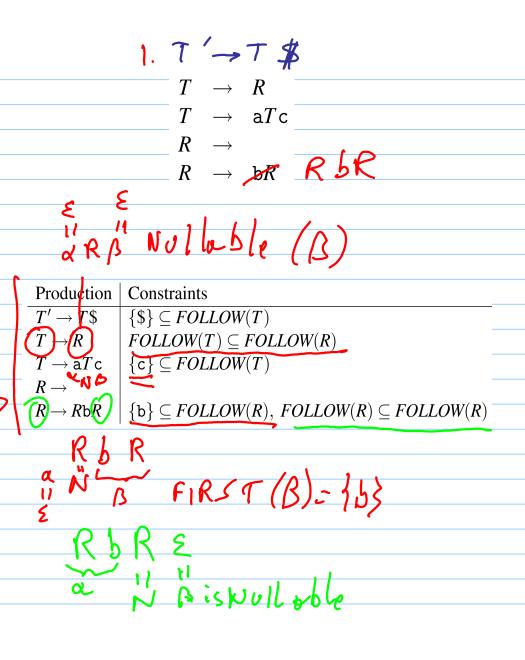
 $M \rightarrow \alpha N$

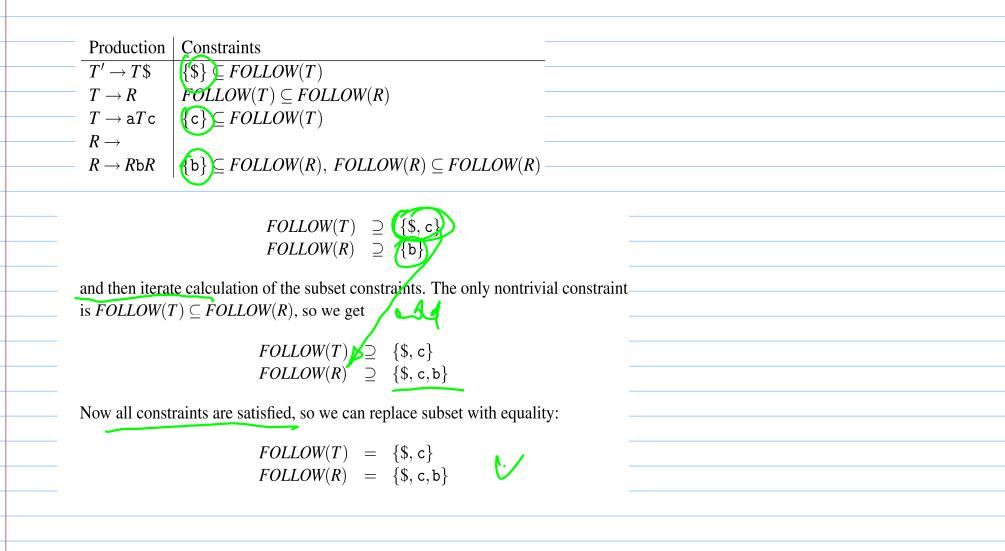
A production



The steps taken to calculate the follow sets of a grammar are, hence:

- -1. Add a new nonterminal $S' \rightarrow S$, where S is the start symbol for the original grammar. S' is the start symbol for the extended grammar.
- 2. For each nonterminal *N*, locate all occurrences of *N* on the right-hand sides of productions. For each occurrence do the following:
 - 2.1 Let β be the rest of the right-hand side after the occurrence of *N*. Note that β may be empty. In other words, the production is of the form $M \rightarrow \alpha N\beta$, where *M* is a nonterminal (possibly equal to *N*) and α and β are (possibly empty) sequences of grammar symbols. Note that if a right-hand-side contains several occurrences of *N*, we make a split for each occurrence.
 - 2.2 Let $m = FIRST(\beta)$. Add the constraint $m \subseteq FOLLOW(N)$ to the set of constraints. If β is empty, you can omit this constraint, as it does not add anything.
 - 2.3 If $Nullable(\beta)$, find the nonterminal *M* at the left-hand side of the production and add the constraint $FOLLOW(M) \subseteq FOLLOW(N)$. If M = N, you can omit the constraint, as it does not add anything. Note that if β is empty, $Nullable(\beta)$ is true.
- 3. Solve the constraints using the following steps:
 - 3.1 Start with empty sets for *FOLLOW*(*N*) for all nonterminals *N* (not including *S'*).
 - 3.2 For each constraint of the form $m \subseteq FOLLOW(N)$ constructed in step 2.1, add the contents of *m* to FOLLOW(N).
 - 3.3 Iterating until a fixed-point is reached, for each constraint of the form $FOLLOW(M) \subseteq FOLLOW(N)$, add the contents of FOLLOW(M) to FOLLOW(N).





LL(1) Porsing

In LL(1) Porsy, we can choose a production N -> & uniquely if:

- $c \in FIRST(\alpha)$, or
- $Nullable(\alpha)$ and $c \in FOLLOW(N)$, where c is the next input symbol.

L: left-to-right reading Li Laft most derivation 1: me-symbol look-ahead

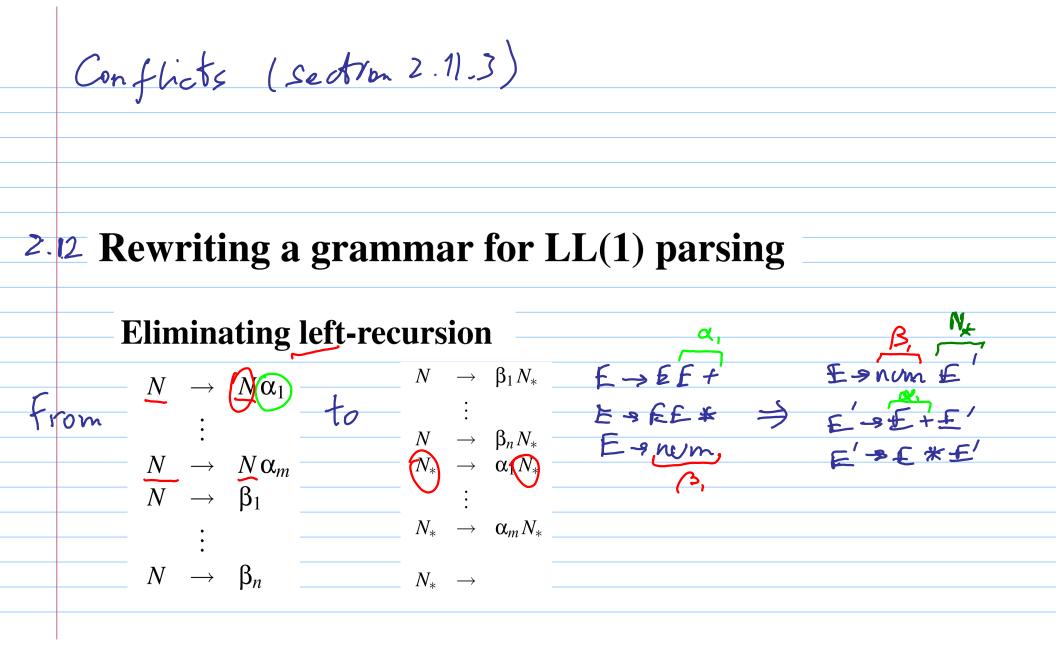
sometimes called "nerth gives the most in put cymed function parseT () = function parse T() = X → aTc —& if input = 'b' or input = 'c' or input = '\$' then parseR() else if input = 'a' then match('a') ; parseT() ; match('c') R bR else reportError(function parseR() = if input = 'c' or input = '\$' then (* do nothing, just return *) else if input = 'b' then match('b') ; parseR() else reportError() Fig. 2.17 Recursive descent parser for Grammar 2.9 top-down

```
function parseT'() =
        if input = 'a' or input = 'b' or input = '$' then
           let tree = parseT() in
              match('$');
              return tree
        else reportError()
      function parseT() =
        if input = 'b' or input = 'c' or input = '$' then
          let tree = parseR() in
            return nNode ('T', [tree])
        else if input = 'a' then
          match('a') ;
          let tree = parseT() in
            match('c') ;
            return nNbde('T', [tNode('a'), tree, tNode('c')])
        else reportError()
      function parseR() =
        if input = 'c' or input = '$' then
          return (Node ('R', [])
        else if input = 'b' then
          match('b') ;
          let tree_parseR() in
            return (Node('R', [tNode('b'), tree])
        else reportError()
Fig. 2.18 (Tree-building) ecursive descent parser for Grammar 2.9
```

```
\leq
```

LL(I) Toble - which production to Toble - vie for each NT & inpot symbol		(top on left
	input	stack	side)
$\begin{array}{c ccccccccccccccccccccccccccccccccccc$	aabbbcc\$	T'	
$T' \xrightarrow{T' \to T} T' \to T \xrightarrow{T' \to T} T' \to T \xrightarrow{T' \to T} T' \to T \xrightarrow{T' \to T} T' \to T$	aabbbcc\$	T	
$\begin{array}{ c c c c c c c c c c c c c c c c c c c$	aabbbcc\$	aTc\$	
$R \mid R \rightarrow bR R \rightarrow R \rightarrow K \rightarrow bR $	abbbcc\$	Tc\$	
	abbbcc\$	aTcc\$	
T > T \$ > a T c \$ > Ra T c \$ > a a R c c \$	(bbcc\$	Dcc\$	
4 4 4 3	bbbcc\$	Rcc\$	
Jaab RCC\$ Jaabb Rcc\$ Jaabbb Rcc\$ Joabbb cc\$	bbbcc\$	bRcc\$	
at a la comptation of an and (m) at a she	bbcc\$	Rcc $$$	
<pre>stack := empty ; push(T',stack) while stack <> empty do</pre>	bbcc\$	bRcc\$	
if top(stack) is a terminal then	bcc\$	Rcc $$$	
<pre>match(top(stack)) ; pop(stack)</pre>	bcc\$	bRcc\$	
<pre>else if table(top(stack), input) = empty then reportError</pre>	cc\$	Rcc $$$	
else	cc\$	cc\$	
<pre>rhs := rightHandSide(table(top(stack), input))</pre>	c\$	c\$	
pop(stack) ;	\$	\$	
Profram for tuble. Olviven U(1) porging			
A KING THE COLLECT COLLECT COLLECT COLLECTION			

```
stack := empty ; push(T'-node, stack)
        while stack <> empty do
           if top(stack) is a terminal then
             match(top(stack)) ; pop(stack)
           else if table(top(stack), input) = empty then
             reportError
           else
             terminal := pop(stack) ;
             rhs := rightHandSide(table(terminal, input)) ;
             children := makeNodes(rhs);
             addChildren(terminal, children);
             pushList(children, stack)
Fig. 2.22 Tree-building program for table-driven LL(1) parsing
```



Another example of left rear sign elimination

 $Exp \rightarrow Exp+Exp2$ Exp $Exp \rightarrow Exp-Exp2$ Exp_* $Exp \rightarrow Exp2$ Exp_* $Exp2 \rightarrow Exp2*Exp3$ Exp_* $Exp2 \rightarrow Exp2/Exp3$ Exp2 $Exp2 \rightarrow Exp3$ Exp2 $Exp3 \rightarrow num$ Exp2 $Exp3 \rightarrow (Exp)$ Exp2

 $Exp \rightarrow Exp2 Exp_*$ $Exp_* \rightarrow + Exp2 Exp_*$ $Exp_* \rightarrow - Exp2 Exp_*$ $Exp_* \rightarrow$ $Exp_* \rightarrow$

Indirect left-recursion

1. There are mutually left-recursive productions

 $\begin{array}{cccc} \overbrace{N_1} & \rightarrow & N_2 \alpha_1 \\ & \searrow & N_3 \alpha_2 \\ & \vdots \\ & N_{k-1} & \rightarrow & N_k \alpha_{k-1} \\ & N_k & \rightarrow & N_1 \alpha_k \end{array}$

Two besic coses of indirect left recorsion $N_1 \ni N_{2\alpha_1} \ni \cdots \ni \bigwedge_{k=1}^{N_1} \alpha_k \alpha_{k-1} \cdots \alpha_k$

2. There is a production $N \rightarrow \alpha N\beta$ where α is *Nullable*.

In general, N=>aNB, where Nis Nollable.

Left-factorisation *Stat* \rightarrow **id** := *Exp* Stat \rightarrow id := Exp Stat Stat Star (ignore for the example) Stat \rightarrow if Exp then $Stat \ Elsepart$ Stat \rightarrow **if** Exp **then** Statelse Stat $(1) Elsepart \rightarrow else Stat$ *Stat* \rightarrow **if** *Exp* **then** *Stat* (2) Elsepart \rightarrow Lowmon RHS prepixes: foctorizo! The resulting grammer is still not LL(1), because else is both in FOLLOW (Else pert) (which would lead us to choose (2)) and in FIRST (ely Stat) (which would lead us to choose (1)).

This unombiguous grommer \rightarrow Stat2 ; Stat Stat Stat \rightarrow Stat2 \rightarrow Matched Stat2 \rightarrow Unmatched Stat2 \rightarrow if *Exp* then *Matched* else *Matched* Matched Matched \rightarrow id := Exp $Unmatched \rightarrow if Exp$ then Matched else Unmatched *Unmatched* \rightarrow if *Exp* then *Stat*2 is hand to rewrite in a form suitable for LL(1), so here $Stat \rightarrow id := Ern$ prochla the ambigrov gronner Stat \rightarrow if *Exp* then *Stat Elsepart Elsepart* \rightarrow else *Stat* Elsepart \rightarrow is used anyway, and the first production for Elcepert is pribritized over the second one. Prioritize the rerely resolves conflicts in LL(1) orommers!

Construction of LL(1) parsers summarized

1. Eliminate ambiguity that cannot be resolved by prioritizing productions (in
2. Eliminate left-recursion
3. Perform left factorisation where required
4. Add an extra start production $S' \rightarrow S$ \$ to the grammar.
5. Calculate <i>FIRST</i> for every production and <i>FOLLOW</i> for every nonterminal.
6. For nonterminal N and input symbol c, choose production $N \rightarrow \alpha$ when:
• $c \in FIRST(\alpha)$, or
• <i>Nullable</i> (α) and $c \in FOLLOW(N)$.
 This choice is encoded either in a table or a recursive-descent program.
 7. Use production priorities to eliminate conflicts where appropriate.