aabbbcc

$$
\begin{aligned}
T &\to R\\
T &\to aTc\\
R &\to \\
R &\to RbR
\end{aligned}
$$

ambiguous

$\alpha\, R\, \beta$

$\beta = \varepsilon$

$$
\begin{aligned}
T &\to R\\
T &\to aTc\\
R &\to \\
R &\to bR
\end{aligned}
$$

unambiguous

$$
\begin{aligned}
&\underline{T}\\
\Rightarrow\ & a\underline{T}c\\
\Rightarrow\ & aa\underline{T}cc\\
\Rightarrow\ & aa\underline{R}cc\\
\Rightarrow\ & aa\underline{R}bRcc\\
\Rightarrow\ & aa\underline{R}bRbRcc\\
\Rightarrow\ & aab\underline{R}bRcc\\
\Rightarrow\ & aab\underline{R}bRbRcc\\
\Rightarrow\ & aabb\underline{R}bRcc\\
\Rightarrow\ & aabbb\underline{R}cc\\
\Rightarrow\ & aabbbcc
\end{aligned}
$$

using the ambiguous grammar

$$
\begin{aligned}
&T\\
\Rightarrow\ & aTc\\
\Rightarrow\ & aaTcc\\
\Rightarrow\ & aaRcc\\
& aabRcc\\
& aabbRcc\\
& aabbbRcc\\
& aabbbcc
\end{aligned}
$$

using the unambiguous grammar

Definition ~~3.2~~ 2.2 *A symbol c is in* $\boxed{FIRST(\alpha)}$ *if and only if* $\alpha \Rightarrow c\beta$ *for some (possibly empty) sequence* β *of grammar symbols*, and $FIRST(\underline{\underline{N}} \rightarrow \alpha) = FIRST(\alpha)$.

Definition ~~3.3~~ 2.3 *A sequence* α *of grammar symbols is* Nullable *(we write this as Nullable(*α*)) if and only if* $\alpha \Rightarrow \varepsilon$, and a production $N \rightarrow \alpha$ is called Nullable if $Nullable(\alpha)$.

Algorithm ~~3~~2.4

$$
\begin{aligned}
Nullable(\varepsilon) &= true \\
Nullable(\mathtt{a}) &= false \\
Nullable(\alpha\beta) &= Nullable(\alpha) \wedge Nullable(\beta) \\
Nullable(N) &= Nullable(\alpha_1) \vee \ldots \vee Nullable(\alpha_n), \\
& \quad \textit{where the productions for } N \textit{ are} \\
& \quad N \rightarrow \alpha_1, \quad \ldots \quad , N \rightarrow \alpha_n
\end{aligned}
$$

$$
\begin{array}{lcl}
T & \rightarrow & R \\
T & \rightarrow & \mathtt{a}T\mathtt{c} \\
R & \rightarrow & \\
R & \rightarrow & \mathtt{b}R
\end{array}
$$

Grammar 2.9

$$
\begin{aligned}
Nullable(T) &= Nullable(R) \vee Nullable(\mathtt{a}T\mathtt{c}) \\
&= Nullable(R) \vee (Nullable(a) \wedge Nullable(T) \wedge Nullable(c)) \\
&= Nullable(R) \vee (false \wedge Nullable(T) \wedge false) \\
&= \underline{Nullable(R)}
\end{aligned}
$$

$$
\begin{aligned}
Nullable(R) &= Nullable(\varepsilon) \vee Nullable(\mathtt{b}R) \\
&= true \vee Nullable(bR) = \underline{true}
\end{aligned}
$$

Nonterminal				
T	*false*	*false*	*true*	*true*
R	*false*	*true*	*true*	*true*

✓

Production	*Nullable*	
$T \to R$	*true*	, because R is Nullable
$T \to \mathtt{a}T\mathtt{c}$	*false*	, because aTc is not Nullable
$R \to$	*true*	,
$R \to \mathtt{b}R$	*false*	,

Algorithm ~~3~~2.5

$$
\begin{array}{lcl}
FIRST(\varepsilon) & = & \emptyset \\
FIRST(\mathtt{a}) & = & \{\mathtt{a}\} \\
FIRST(\alpha\beta) & = & \begin{cases} FIRST(\alpha) \cup FIRST(\beta) & \text{if } Nullable(\alpha) \\ FIRST(\alpha) & \text{if not } Nullable(\alpha) \end{cases} \\
FIRST(N) & = & FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n) \\
 & & \text{where the productions for } N \text{ are} \\
 & & N \to \alpha_1, \quad \ldots \quad , N \to \alpha_n
\end{array}
$$

$$
\begin{aligned}
FIRST(T) &= FIRST(R) \cup FIRST(\mathtt{a}T\mathtt{c}) \\
&= FIRST(R) \cup FIRST(\mathtt{a}) \\
&= FIRST(R) \cup \{\mathtt{a}\} \\
FIRST(R) &= FIRST() \cup FIRST(\mathtt{b}R) \\
&= \emptyset \cup FIRST(\mathtt{b}) \\
&= \{\mathtt{b}\}
\end{aligned}
$$

$$
\begin{array}{lcl}
T & \to & R \\
T & \to & \mathtt{a}T\mathtt{c} \\
R & \to & \\
R & \to & \mathtt{b}R
\end{array}
$$

Fig. 2.16 Fixed-point iteration for calculation of *FIRST*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
T	$\emptyset$	$\{\mathtt{a}\}$	$\{\mathtt{a},\mathtt{b}\}$	$\{\mathtt{a},\mathtt{b}\}$
R	$\emptyset$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$

Fig. 2.16 Fixed-point iteration for calculation of *FIRST*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
T	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{a, b\}$
R	$\emptyset$	$\{b\}$	$\{b\}$	$\{b\}$

Production	*FIRST*
$T \rightarrow R$	$\{b\}$
$T \rightarrow aTc$	$\{a\}$
$R \rightarrow$	$\emptyset$
$R \rightarrow bR$	$\{b\}$

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~~2.6 *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* $S \Rightarrow \alpha \boxed{N\mathtt{a}}\beta$, *where* α *and* β *are (possibly empty) sequences of grammar symbols.*

1

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:

$$S' \to S\$$$

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.

A production

$$M \to \alpha N\beta$$

generates the constraint $FIRST(\beta) \subseteq FOLLOW(N)$ since β, obviously, can follow N. Furthermore, if $Nullable(\beta)$ the production also generates the constraint $FOLLOW(M) \subseteq FOLLOW(N)$ (note the direction of the inclusion). The reason is that, if a symbol c is in $FOLLOW(M)$, then there (by definition) is a derivation $S' \Rightarrow \gamma Mc\delta$. But since $M \to \alpha N\beta$ and β is nullable, we can continue this by $\gamma Mc\delta \Rightarrow \gamma\alpha Nc\delta$, so c is also in $FOLLOW(N)$.

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 $FOLLOW(N)$.

1. $T' \rightarrow T\$$

$$
\begin{aligned}
T &\rightarrow R \\
T &\rightarrow \mathtt{a}T\mathtt{c} \\
R &\rightarrow \\
R &\rightarrow \cancel{\mathtt{b}R}\ \ R\mathtt{b}R
\end{aligned}
$$

$\underset{\varepsilon}{\alpha} R \underset{\varepsilon}{\beta}$ Nullable (β)

Production	Constraints
$T' \rightarrow T\$$	$\{\$\} \subseteq FOLLOW(T)$
$T \rightarrow R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \rightarrow \mathtt{a}T\mathtt{c}$	$\{\mathtt{c}\} \subseteq FOLLOW(T)$
$R \rightarrow$	
$R \rightarrow R\mathtt{b}R$	$\{\mathtt{b}\} \subseteq FOLLOW(R)$, $FOLLOW(R) \subseteq FOLLOW(R)$

NB

$\underset{N}{R}\ \underbrace{b\ R}_{\beta}$, $\alpha = \varepsilon$ FIRST(β) = {b}

$\underbrace{R\ b}_{\alpha}\ \underset{N}{R}\ \underset{\beta}{\varepsilon}$ β is Nullable

Production	Constraints
$T' \to T\$$	$\{\$\} \subseteq FOLLOW(T)$
$T \to R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \to \mathtt{a}T\mathtt{c}$	$\{\mathtt{c}\} \subseteq FOLLOW(T)$
$R \to$	
$R \to R\mathtt{b}R$	$\{\mathtt{b}\} \subseteq FOLLOW(R),\ FOLLOW(R) \subseteq FOLLOW(R)$

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, \mathtt{c}\} \\ FOLLOW(R) &\supseteq \{\mathtt{b}\} \end{aligned}$$

and then iterate calculation of the subset constraints. The only nontrivial constraint is $FOLLOW(T) \subseteq FOLLOW(R)$, so we get

add

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, \mathtt{c}\} \\ FOLLOW(R) &\supseteq \{\$, \mathtt{c}, \mathtt{b}\} \end{aligned}$$

Now all constraints are satisfied, so we can replace subset with equality:

$$\begin{aligned} FOLLOW(T) &= \{\$, \mathtt{c}\} \\ FOLLOW(R) &= \{\$, \mathtt{c}, \mathtt{b}\} \end{aligned}$$

LL(1) Parsing

In LL(1) Parsing, we can choose a production $N \to \alpha$ 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

L: Leftmost derivation

1: one-symbol look-ahead

sometimes called "next" gives the next input symbol

FIRST

```
function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    parseR()
  else if input = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    (* do nothing, just return *)
  else if input = 'b' then
    match('b') ; parseR()
  else reportError()
```

checks the next input symbol & consumes it

Fig. 2.17 $\boxed{\text{Recursive}}\,\boxed{\text{descent}}$ parser for Grammar 2.9

top-down

$$T' \rightarrow T\,\$$$

$$\begin{array}{lcl} T & \rightarrow & R \\ T & \rightarrow & \mathrm{a}T\mathrm{c} \\ R & \rightarrow & \\ R & \rightarrow & \mathrm{b}R \end{array}$$

a

```
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 nNode('T', [tNode('a'),tree,tNode('c')])
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    return nNode('R', [])
  else if input = 'b' then
    match('b') ;
    let tree = parseR() in
      return nNode('R', [tNode('b'),tree])
  else reportError()
```

Fig. 2.18 Tree-building recursive descent parser for Grammar 2.9

LL(1) Table: which production to use for each NT & input symbol

	a	b	c	$
T'	$T' \to T\$$	$T' \to T\$$		$T' \to T\$$
T	$T \to \mathtt{a}T\mathtt{c}$	$T \to R$	$T \to R$	$T \to R$
R		$R \to \mathtt{b}R$	$R \to$	$R \to$

0) $T' \to T\$$
1) $T \to R$
2) $T \to \mathtt{a}T\mathtt{c}$
3) $R \to$
4) $R \to \mathtt{b}R$

$$T' \overset{1}{\Rightarrow} T\$ \overset{2}{\Rightarrow} aTc\$ \overset{2}{\Rightarrow} aaTcc\$ \overset{1}{\Rightarrow} aaRcc\$$$

$$\overset{4}{\Rightarrow} aabRcc\$ \overset{4}{\Rightarrow} aabbRcc\$ \overset{4}{\Rightarrow} aabbbRcc\$ \overset{3}{\Rightarrow} aabbbcc\$$$

(top on left side)

input	stack
aabbbcc$	T'
aabbbcc$	$T$$
aabbbcc$	aTc$
abbbcc$	Tc$
abbbcc$	aTcc$
bbbcc$	Tcc$
bbbcc$	Rcc$
bbbcc$	bRcc$
bbcc$	Rcc$
bbcc$	bRcc$
bcc$	Rcc$
bcc$	bRcc$
cc$	Rcc$
cc$	cc$
c$	c$
$	$

```
stack := empty ; push(T',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
    rhs := rightHandSide(table(top(stack),input))
    pop(stack) ;
    pushList(rhs,stack)
```

Program for table-driven LL(1) parsing

```
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

Conflicts (section 2.11.3)

2.12 Rewriting a grammar for LL(1) parsing

Eliminating left-recursion

from

$$\begin{array}{lcl}
N & \rightarrow & N\alpha_1 \\
 & \vdots & \\
N & \rightarrow & N\alpha_m \\
N & \rightarrow & \beta_1 \\
 & \vdots & \\
N & \rightarrow & \beta_n
\end{array}$$

to

$$\begin{array}{lcl}
N & \rightarrow & \beta_1 N_* \\
 & \vdots & \\
N & \rightarrow & \beta_n N_* \\
N_* & \rightarrow & \alpha_1 N_* \\
 & \vdots & \\
N_* & \rightarrow & \alpha_m N_* \\
N_* & \rightarrow &
\end{array}$$

$$\begin{array}{l}
E \rightarrow E\,\overbrace{E\,+}^{\alpha_1} \\
E \rightarrow E\,E\,* \\
E \rightarrow \underbrace{num}_{\beta_1}
\end{array}
\Rightarrow
\begin{array}{l}
E \rightarrow \overbrace{num}^{\beta_1}\,\overbrace{E'}^{N_*} \\
E' \rightarrow \overbrace{E\,+}^{\alpha_1}\,E' \\
E' \rightarrow E\,*\,E'
\end{array}$$

Another example of left recursion elimination

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 \,/\, Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (\, Exp\,)
\end{array}
\Longrightarrow
\begin{array}{lcl}
Exp & \rightarrow & Exp2\ Exp_{*} \\
Exp_{*} & \rightarrow & +\ Exp2\ Exp_{*} \\
Exp_{*} & \rightarrow & -\ Exp2\ Exp_{*} \\
Exp_{*} & \rightarrow & \\
Exp2 & \rightarrow & Exp3\ Exp2_{*} \\
Exp2_{*} & \rightarrow & *\ Exp3\ Exp2_{*} \\
Exp2_{*} & \rightarrow & /\ Exp3\ Exp2_{*} \\
Exp2_{*} & \rightarrow & \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (\, Exp\,)
\end{array}
$$

Indirect left-recursion

1. There are mutually left-recursive productions

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

Two basic cases of indirect left recursion

$\underline{\underline{N_1}} \Rightarrow N_2\alpha_1 \Rightarrow \cdots \Rightarrow \underline{\underline{N_1}}\alpha_k\alpha_{k-1}\cdots\alpha_1$

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

In general, $N \Rightarrow \alpha N \beta$, where N is Nullable.

Left-factorisation

$$
\begin{array}{lcl}
Stat & \rightarrow & \mathbf{id} := Exp \\
\cancel{Stat} & \cancel{\rightarrow} & \cancel{Stat\ ;\ Stat} \\
Stat & \rightarrow & \boxed{\mathbf{if}\ Exp\ \mathbf{then}\ Stat}\ \mathbf{else}\ Stat \\
Stat & \rightarrow & \boxed{\mathbf{if}\ Exp\ \mathbf{then}\ Stat}
\end{array}
$$

(ignore for the example)

common RHS prefixes: factorize!

$$
\begin{array}{llcl}
 & Stat & \rightarrow & \mathbf{id} := Exp \\
 & Stat & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Stat\ \underline{Elsepart} \\
(1) & Elsepart & \rightarrow & \underline{\texttt{else}}\ Stat \\
(2) & Elsepart & \rightarrow &
\end{array}
$$

The resulting grammar is still not LL(1), because <u>else</u> is both in FOLLOW(Elsepart) (which would lead us to choose (2)) and in FIRST (else Stat) (which would lead us to choose (1)).

This unambiguous grammar

Stat	$\rightarrow$	*Stat*2 ; *Stat*
Stat	$\rightarrow$	*Stat*2
*Stat*2	$\rightarrow$	*Matched*
*Stat*2	$\rightarrow$	*Unmatched*
Matched	$\rightarrow$	`if` *Exp* `then` *Matched* `else` *Matched*
Matched	$\rightarrow$	**id** := *Exp*
Unmatched	$\rightarrow$	`if` *Exp* `then` *Matched* `else` *Unmatched*
Unmatched	$\rightarrow$	`if` *Exp* `then` *Stat*2

is hard to rewrite in a form suitable for LL(1), so the predictable the ambiguous grammar

Stat	$\rightarrow$	**id** := *Exp*
Stat	$\rightarrow$	`if` *Exp* `then` *Stat* *Elsepart*
Elsepart	$\rightarrow$	`else` *Stat*
Elsepart	$\rightarrow$	

is used anyway,
and the first production for Elsepart is prioritized over the second one.
Prioritization rarely resolves conflicts in LL(1) grammars!

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 $S' \to S\$$ to the grammar.
5. Calculate *FIRST* for every production and *FOLLOW* for every nonterminal.
6. For nonterminal N and input symbol c, choose production $N \to \alpha$ when:
   - $c \in \mathit{FIRST}(\alpha)$, or
   - $\mathit{Nullable}(\alpha)$ and $c \in \mathit{FOLLOW}(N)$.

   This choice is encoded either in a table or a recursive-descent program.
7. Use production priorities to eliminate conflicts where appropriate.