LL(1) Limitations

#	Rules
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ endif\ \$$
2	$Stmt \rightarrow if\ Expr\ then\ StmtList\ else\ StmtList\ endif\ \$$
3	$StmtList \rightarrow StmtList\ ;\ Stmt$
4	$StmtList \rightarrow Stmt$
5	$Expr \rightarrow var + Expr$
6	$Expr \rightarrow var$

1. Suppose we want to generate an LL(1) parsing table (LLT) for this grammar, will we encounter problems? Discuss...

LL(1) Limitations

The problem is that rules 1 and 2 share common prefixes (`if`), as well do rules 5 & 6 (`var`).

#	$p \in P$	Computed By	Predict Set
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ endif\ \$$	$FirstSet(RHS)$	`if`
2	$Stmt \rightarrow if\ Expr\ then\ StmtList\ else\ StmtList\ endif\ \$$	$FirstSet(RHS)$	`if`
3	$StmtList \rightarrow StmtList\ ;\ Stmt$	$FirstSet(RHS)$	`if`
4	$StmtList \rightarrow Stmt$	$FirstSet(RHS)$	`if`
5	$Expr \rightarrow var + Expr$	$FirstSet(RHS)$	`var`
6	$Expr \rightarrow var$	$FirstSet(RHS)$	`var`

	+	;	else	endif	if	then	var	$
Stmt					⋆			
Expr							⋆	
StmtList					⋆			

Common-Prefix Consolidation ("Left Factoring")

Solution: **manipulate the rule suffices into a new non-terminal.**

#	$p \in P$	Computed By	Predict Set
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ Q\ \$$	$FirstSet(RHS)$	`if`
2	$Q \rightarrow endif$	$FirstSet(RHS)$	`endif`
3	$Q \rightarrow else\ StmtList\ endif$	$FirstSet(RHS)$	`else`
4	$StmtList \rightarrow StmtList\ ;\ Stmt$	$FirstSet(RHS)$	`if`
5	$StmtList \rightarrow Stmt$	$FirstSet(RHS)$	`if`
6	$Expr \rightarrow var\ W$	$FirstSet(RHS)$	`var`
7	$W \rightarrow +\ Expr$	$FirstSet(RHS)$	`+`
8	$W \rightarrow \lambda$	$FollowSet(LHS)$	`then`

1. Add Q so that $Stmt$ can be written with one rule.
2. Add W so that $Expr$ can be written with one rule.

	+	;	else	endif	if	then	var	$
Stmt					1			
Expr							6	
StmtList					⋆			
Q			3	2				
W	7					8		

Eliminate Left Recursion

We still have a problem with the predict set for $StmtList$, this is because

$$StmtList \quad \rightarrow \quad StmtList \ ; Stmt$$

is a **left recursive rule**.

Solution: **refactor the recursive rule of the grammar to be RIGHT recursive, adding a new non-terminal to maintain the same language.**

Eliminate Left Recursion

Solution: **refactor the recursive rule of the grammar to be RIGHT recursive, adding a new non-terminal to maintain the same language.**

The grammar, before refactoring, permits statement lists of the form

$$\begin{array}{rcl} \mathit{StmtList} & \rightarrow & \mathit{StmtList}\ ;\ \mathit{Stmt} \\ & | & \mathit{Stmt} \end{array} \quad \Rightarrow \quad \texttt{Stmt ; Stmt ; Stmt ; ... ; Stmt}$$

Making the rule **right recursive** lets a statement list **begin** the same way...

$$\mathit{StmtList} \rightarrow \mathit{Stmt}\ R \quad \Rightarrow \quad \texttt{Stmt ...}$$

... now we just have to permit $(\texttt{;}\ \texttt{Stmt})\star$ with rules for R ...

$$\begin{array}{rcl} \mathit{StmtList} & \rightarrow & \mathit{Stmt}\ R \\ R & \rightarrow & ;\ \mathit{Stmt}\ R \\ & | & \lambda \end{array}$$

Eliminate Left Recursion

#	$p \in P$	Computed By	Predict Set
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ Q\ \$$	$FirstSet(RHS)$	`if`
2	$Q \rightarrow endif$	$FirstSet(RHS)$	`endif`
3	$Q \rightarrow else\ StmtList\ endif$	$FirstSet(RHS)$	`else`
4	$StmtList \rightarrow Stmt\ R$	$FirstSet(RHS)$	`if`
5	$R \rightarrow ;\ Stmt\ R$	$FirstSet(RHS)$	`;`
6	$R \rightarrow \lambda$	$FollowSet(LHS)$	`else`, `endif`
7	$Expr \rightarrow var\ W$	$FirstSet(RHS)$	`var`
8	$W \rightarrow +\ Expr$	$FirstSet(RHS)$	`+`
9	$W \rightarrow \lambda$	$FollowSet(LHS)$	`then`

$$\begin{array}{lcl} A & \rightarrow & A\gamma\beta \\ A & \rightarrow & \beta \end{array} \quad \Rightarrow \quad \begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \gamma\beta R \\ & | & \lambda \end{array}$$

(γ may be "empty," recall lower Greek letters are $(\Sigma + N)*$)

	+	;	else	endif	if	then	var	$
Stmt					1			
Expr							7	
StmtList					4			
Q			3	2				
R		5	6	6				
W	8					9		

Dangling Brackets and LL(1) Grammars

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S\, T$
3	$S \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

The language for this grammar is

$$\{\langle^i \rangle^j \mid i \geq j \geq 0\}$$

(there are always as many or more opening brackets than closing brackets).

	$\rangle$	$\langle$	$\$$
P		1	1
S	3	2	3
T	$\star$		5

A seemingly unimportant language with a predict set conflict.

#	$p \in P$	Computed By	Predict Set
1	$P \rightarrow S\,\$$	$FirstSet(RHS)$	$\langle$, $\$$
2	$S \rightarrow \langle\, S\, T$	$FirstSet(RHS)$	$\langle$
3	$S \rightarrow \lambda$	$FollowSet(LHS)$	$\rangle$, $\$$
4	$T \rightarrow \rangle$	$FirstSet(RHS)$	$\rangle$
5	$T \rightarrow \lambda$	$FollowSet(LHS)$	$\rangle$, $\$$

Dangling Brackets and LL(1) Grammars

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S\, T$
3	$S \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

In the sentence $\langle\ \langle\ \rangle$, the closing bracket could be associated with either of the opening brackets depending on the order in which rules 4 and 5 are applied.

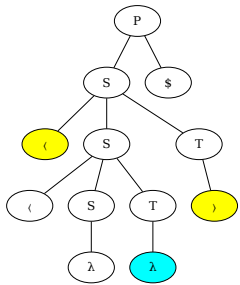
"T doesn't know" to which opening bracket a closing bracket belongs.

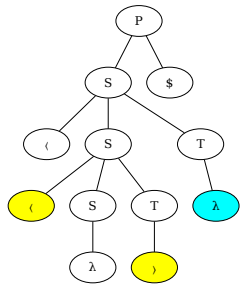
The language for this grammar is

$$\{\, \langle^i \rangle^j \mid i \geq j \geq 0 \,\}$$

(there are always as many or more opening brackets than closing brackets).

	$\rangle$	$\langle$	$\$$
P		1	1
S	3	2	3
T	$\star$		5





Dangling Brackets and LL(1) Grammars

#	Rules
1	$P \to S\,\$$
2	$S \to \langle\, S\, T$
3	$S \to \lambda$
4	$T \to \rangle$
5	$T \to \lambda$

The language for this grammar is

$$\{\langle^i \rangle^j \mid i \geq j \geq 0\}$$

(there are always as many or more opening brackets than closing brackets).

	$\rangle$	$\langle$	$\$$
P		1	1
S	3	2	3
T	$\star$		5

This is the "dangling bracket" problem that LL(1) grammars cannot parse.

. . . and why should we care?

Dangling Brackets, `if-then-else` and LL(1) Grammars

Why do we care about this silly bracket language?

Because it is near and dear to a favorite programming construct. . .

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S\, T$
3	$S \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

$\Rrightarrow$

#	Rules
1	$Program \rightarrow Stmt\,\$$
2	$Stmt \rightarrow \langle\, Stmt\, T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

$\Rrightarrow^{1}$

#	Rules
1	$Program \rightarrow Stmt\,\$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

[1] Substitute $\langle$ with $if\ query\ then$ and $\rangle$ with $else\ Stmt$.

Technically, this is no longer DBL (we've added terminals), but it demonstrates DBL's closeness to `if-then-else`.

Dangling Brackets, `if-then-else` and LL(1) Grammars

But everyone knows the "dangling bracket" belongs to the nearest sibling before it...

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S\, T$
3	$S \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

$\equiv$

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S$
3	$S \rightarrow T$
4	$T \rightarrow \langle\, T\, \rangle$
5	$T \rightarrow \lambda$

Both grammars generate the same language, namely

$$\{\, \langle^i \rangle^j \mid i \geq j \geq 0 \,\}$$

But the new grammar (with rule 4: $T \rightarrow \langle\, T\, \rangle$) assures us that paired brackets belong to the same parse tree node, T's λ production **can't be misplaced!**

. . . would anyone like to see the predict sets for this new improved grammar?

Dangling Brackets, `if-then-else` and LL(1) Grammars

Now it is S that doesn't know if an opening bracket has a closing bracket (use rule 3 $S \rightarrow T\ \rangle$) or does not (use rule 2 $S \rightarrow \langle\ S$).

	$\rangle$	$\langle$	\$
P		1	1
S	3	2	3
T	$\star$		5

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\ S\,T$
3	$S \rightarrow \lambda$
4	$T \rightarrow \rangle$
5	$T \rightarrow \lambda$

$\equiv$

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\ S$
3	$S \rightarrow T$
4	$T \rightarrow \langle\ T\ \rangle$
5	$T \rightarrow \lambda$

	$\rangle$	$\langle$	\$
P		1	1
S		$\star$	3
T	5	4	5

#	$p \in P$	Computed By	Predict Set
1	$P \rightarrow S\,\$$	$FirstSet(RHS)$	$\langle$, \$
2	$S \rightarrow \langle\ S$	$FirstSet(RHS)$	$\langle$
3	$S \rightarrow T$	$FirstSet(RHS) \bigcup FollowSet(LHS)$	$\langle$, \$
4	$T \rightarrow \langle\ T\ \rangle$	$FirstSet(RHS)$	$\langle$
5	$T \rightarrow \lambda$	$FollowSet(LHS)$	$\rangle$, \$

Dangling Brackets, `if-then-else` and LL(1) Grammars

Trying to factor out common prefixes for S is fruitless (you are encouraged to try it!). You will convince yourself that **no amount of look ahead or factoring** solves this problem (for arbitrary i and j).

#	Rules
1	$P \rightarrow S\,\$$
2	$S \rightarrow \langle\, S$
3	$S \rightarrow T$
4	$T \rightarrow \langle\, T\, \rangle$
5	$T \rightarrow \lambda$

	$\rangle$	$\langle$	$\$$
P		1	1
S		$\star$	3
T	5	4	5

#	$p \in P$	Computed By	Predict Set
1	$P \rightarrow S\,\$$	$FirstSet(RHS)$	$\langle$, $\$$
2	$S \rightarrow \langle\, S$	$FirstSet(RHS)$	$\langle$
3	$S \rightarrow T$	$FirstSet(RHS) \bigcup FollowSet(LHS)$	$\langle$, $\$$
4	$T \rightarrow \langle\, T\, \rangle$	$FirstSet(RHS)$	$\langle$
5	$T \rightarrow \lambda$	$FollowSet(LHS)$	$\rangle$, $\$$

Dangling Brackets, `if-then-else`: A New Hope

All is not lost! A critical observation can be made that tells us which of rules 4 and 5 to use at the LLT conflict. . . any thoughts?

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

	`else`	`if`	`query`	`then`	`$`
Program		1			1
Stmt	3	2			3
T	$\star$				5

#	$p \in P$	Computed By	Predict Set
1	$Program \rightarrow Stmt\ \$$	$FirstSet(RHS)$	`if`, `$`
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$	$FirstSet(RHS)$	`if`
3	$Stmt \rightarrow \lambda$	$FollowSet(LHS)$	`else`, `$`
4	$T \rightarrow else\ Stmt$	$FirstSet(RHS)$	`else`
5	$T \rightarrow \lambda$	$FollowSet(LHS)$	`else`, `$`

Dangling Brackets, `if-then-else`: A New Hope

We should use rule 4, since **it isn't represented anywhere else in the LLT**, and without rule 4, we would be parsing a **different language**.

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

	`else`	`if`	`query`	`then`	`$`
Program		1			1
Stmt	3	2			3
T	**4**				5

#	$p \in P$	Computed By	Predict Set
1	$Program \rightarrow Stmt\ \$$	$FirstSet(RHS)$	`if`, `$`
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$	$FirstSet(RHS)$	`if`
3	$Stmt \rightarrow \lambda$	$FollowSet(LHS)$	`else`, `$`
4	$T \rightarrow else\ Stmt$	$FirstSet(RHS)$	`else`
5	$T \rightarrow \lambda$	$FollowSet(LHS)$	`else`, `$`

Didn't `if-then-else` used to work with LL(1)?

Our first example clearly had a language with `if-then-else` structures and we were able to generate LLT tables after **common prefix refactoring** and avoiding **left recursion**.

Then we waded through the *slough of dangling brackets*, now it's unclear what's what `:(`

#	Rules
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ Q\ \$$
2	$Q \rightarrow endif$
3	$Q \rightarrow else\ StmtList\ endif$
4	$StmtList \rightarrow Stmt\ R$
5	$R \rightarrow ;\ Stmt\ R$
6	$R \rightarrow \lambda$
7	$Expr \rightarrow var\ W$
8	$W \rightarrow +\ Expr$
9	$W \rightarrow \lambda$

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

Didn't `if-then-else` used to work with LL(1)?

Our first example clearly had a language with `if-then-else` structures and we were able to generate LLT tables after **common prefix refactoring** and avoiding **left recursion**.

Then we waded through the *slough of dangling brackets*, now it's unclear what's what `:(`

#	Rules
1	$Stmt \rightarrow if\ Expr\ then\ StmtList\ Q\ \$$
2	$Q \rightarrow endif$
3	$Q \rightarrow else\ StmtList\ endif$
4	$StmtList \rightarrow Stmt\ R$
5	$R \rightarrow ;\ Stmt\ R$
6	$R \rightarrow \lambda$
7	$Expr \rightarrow var\ W$
8	$W \rightarrow +\ Expr$
9	$W \rightarrow \lambda$

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

See the difference?

Languages with "`endif`" ...

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

	else	if	query	then	$
Program		1			1
Stmt	3	2			3
T	⋆				5

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T\ endif$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

	else	endif	if	query	then	$
Program			1			1
Stmt	3	3	2			3
T	4	5				

Languages with "`endif`" ...

By pairing if beginnings to $endif$ endings, many recursive descent parsers fixed their "dangling bracket" problem.
This works because it turns these control structures into a **matched bracket** language, which you knew from a recent LGA is an LL(1) language.

#	Rules
1	$Program \rightarrow Stmt\ \$$
2	$Stmt \rightarrow if\ query\ then\ Stmt\ T\ endif$
3	$Stmt \rightarrow \lambda$
4	$T \rightarrow else\ Stmt$
5	$T \rightarrow \lambda$

#	Rules
1	$S \rightarrow M\ \$$
2	$M \rightarrow \langle\ M\ \rangle$
3	$M \rightarrow \lambda$

(Matched Bracket Language)