

Checkin 8

Assume an LL(1) parser with...

this selector table:

	(	)	{	}
S	(S)	)	{	}

this syntax stack:

S
)
)
eof

and this (lookahead token:

(

Draw the configuration of the parser after it processes the tokens ()
assume the next character thereafter is an eof

Checkin 8

Assume an LL(1) parser with...

this selector table:

	(	)	{	}
S	(S)	)	{	}

this syntax stack:

S
)
)
eof

and this (lookahead token:

(

Draw the configuration of the parser after it processes the tokens ()
assume the next character thereafter is an eof

Housekeeping

Administrivia

Projects

- P2 due on Wednesday

Trials

- Trial 1 due tonight

Housekeeping

Administrivia

Grades

- P1 should be on Canvas Tomorrow
- Q1 should be on Canvas by Wednesday

We'll talk about Q1 in class on Wednesday

P2 Oracle - tonight

University of Kansas | Drew Davidson

ECCS 665 **COMPILER** ***CONSTRUCTION***

FIRST Sets

Last Time

Review – Predictive Parsing

Intro to Parsing

- Complexity

A New Type of Language – LL(k)

- Intro
- LL(1) parsing

You Should Know

- What parsing is
- What LL(1) languages are
- How an LL(1) parser operates

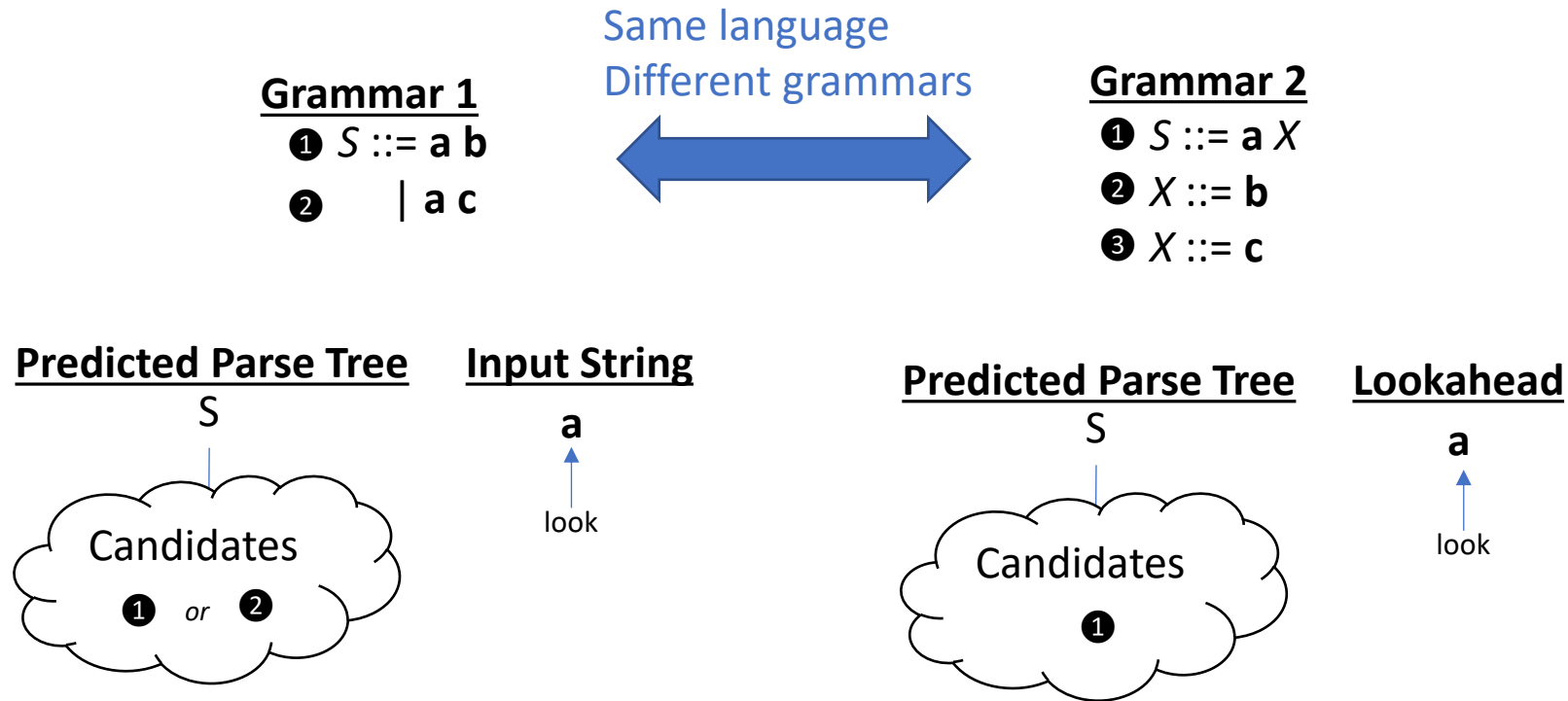


Parsing

Where we Left Off

Review – Predictive Parsing

The language might be LL(1) ... even when the grammar is not!



Today's Outline

Preview – FIRST Sets

Transforming Grammars

- Fixing LL(1) “near misses”

Building LL(1) Parsers

- What the selector table needs
- FIRST Sets



Parsing

LL(1) Grammar Limitations

Transforming Grammars – Fixing LL(1) Near Misses

Given a language, we can't always find an LL(1) grammar *even if one exists*

- Best we can do: simple transformations that remove “obvious” disqualifiers



Checking if a Grammar is LL(1)

Transforming Grammars – Fixing LL(1) Near Misses

If either of the following hold, the grammar is not LL(1):

- The grammar is left-recursive

$X ::= X a \mid b$

- The grammar **isn't** left-factored

$X ::= c X$

$X ::= c a Y$



There are two language-preserving transforms
that will “re-qualify” *some* grammars

(Immediate) Left Recursion

Transforming Grammars – Fixing LL(1) Near Misses

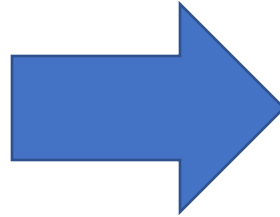
- Recall, a grammar such that $X \xRightarrow{+} X \alpha$ is left recursive
- A grammar is ***immediately*** left recursive if this can happen in one step:
$$A \rightarrow A \alpha \mid \beta$$

Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

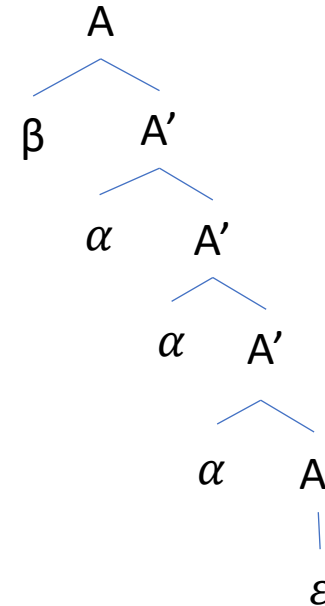
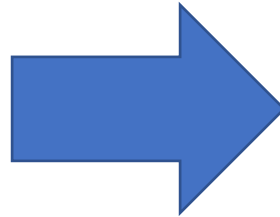
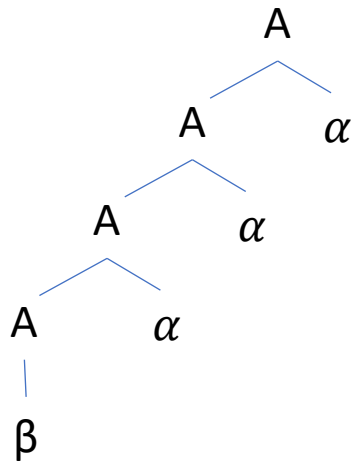
(for a single immediately left-recursive rule)

$$A \rightarrow A\alpha \mid \beta$$



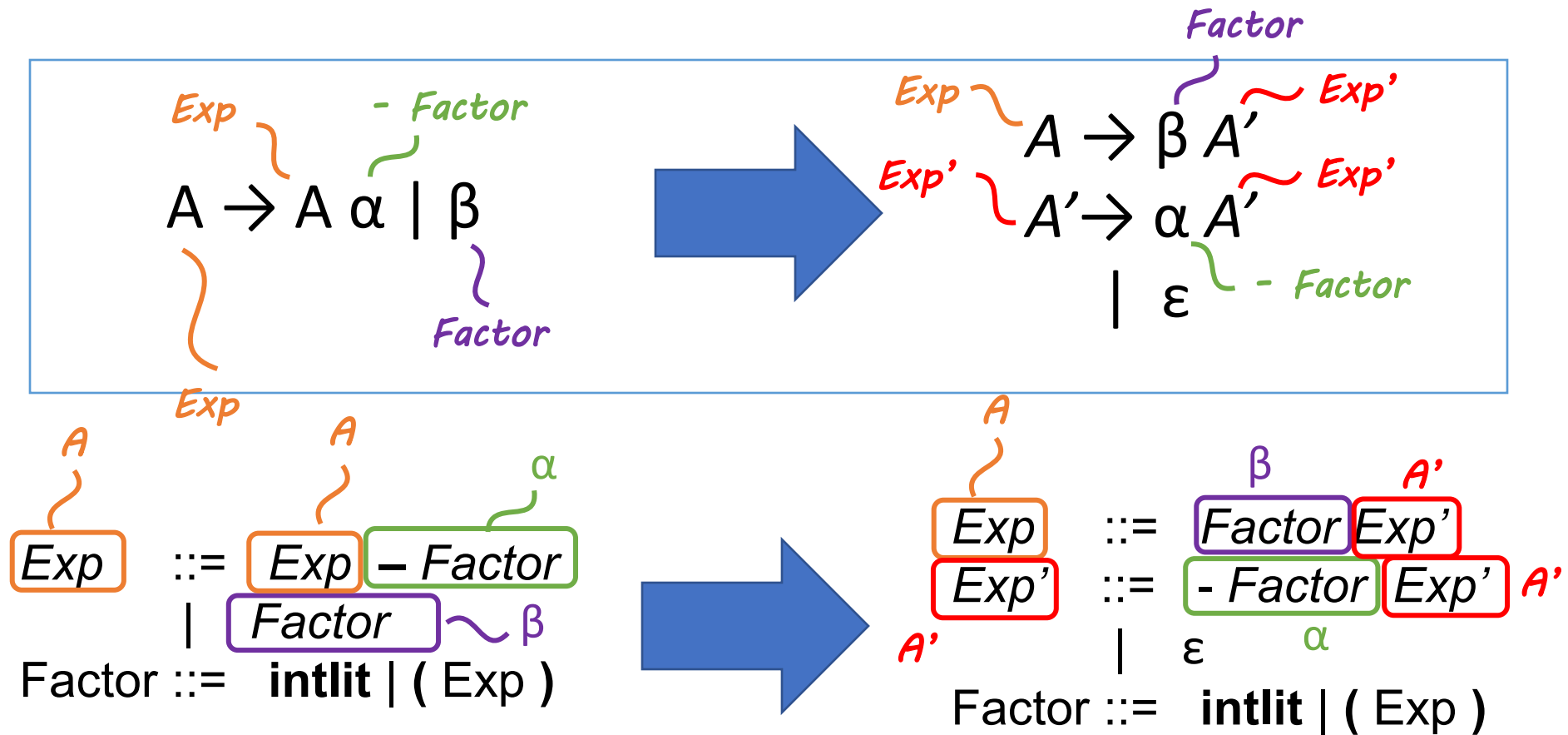
$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \\ &\mid \varepsilon \end{aligned}$$

*Arbitrary Strings
(nonterminal or terminal)*



Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

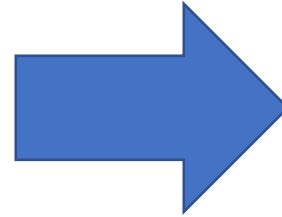


Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

(general rule)

Given Productions

$$\begin{array}{l} A ::= \beta_1 \\ | \beta_2 \\ | \beta_n \\ | A \alpha_1 \\ | A \alpha_2 \\ | A \alpha_m \end{array}$$


Convert to

$$\begin{array}{l} A ::= \beta_1 A' \\ | \beta_2 A' \\ | \beta_n A' \\ A' ::= \alpha_1 A' \\ | \alpha_2 A' \\ | \alpha_m A' \\ | \varepsilon \end{array}$$

Left Factoring Grammar

(Predictive) Parsing - LL(1) Transformations

- If a nonterminal has (at least) two productions whose RHS has a common prefix, the grammar is **not** left factored
(and **not** an LL(1) grammar)

Question: What makes this grammar not left-factored?

$Exp ::= (Exp)$
| $\{ Exp \}$
| $()$
| ab
| bb

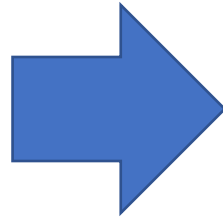
Left Factoring: Simple Rule

(Predictive) Parsing - LL(1) Transformations

Given Productions

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$$

*Pull suffix into
a new nonterminal*

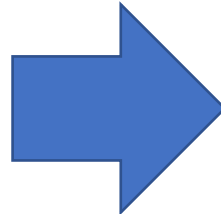


Convert to

$$A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2$$

*Add a new rule
for suffixes*

$$\begin{array}{l} X ::= \alpha \boxed{a \ b \ c \ d} \beta_1 \\ X ::= \alpha \boxed{a \ b \ e \ f} \beta_2 \end{array}$$

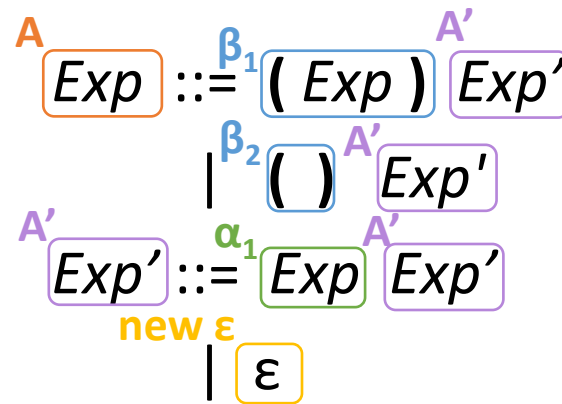
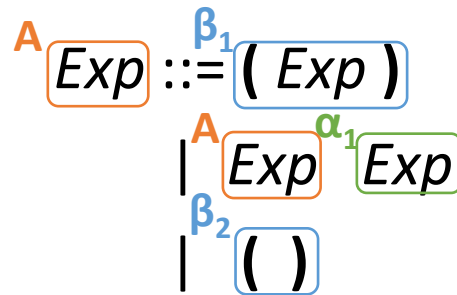


$$\begin{array}{l} X ::= a \ b \ X' \\ X' ::= c \ d \mid e \ f \end{array}$$

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

Remove immediate left-recursion



$$A \rightarrow A \alpha \mid \beta$$

becomes

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \\ &\mid \epsilon \end{aligned}$$

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

$Exp ::= (Exp)$
 $\quad | Exp \ Exp$
 $\quad | ()$

Remove immediate left-recursion



$Exp \overset{A}{::=} \overset{\alpha}{(Exp)} \overset{\beta_1}{Exp'}$
 $\quad | \overset{\alpha}{()} \overset{\beta_2}{Exp'}$
 $Exp' ::= Exp \ Exp'$
 $\quad | \epsilon$

Left-factored

$Exp \overset{A}{::=} \overset{\alpha}{(Exp''$
 $Exp'' \overset{A'}{::=} \overset{\beta_1}{Exp)} \overset{\beta_2}{Exp'}$
 $\quad |) Exp'$
 $Exp' ::= Exp \ Exp'$
 $\quad | \epsilon$

$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$ becomes

$A \rightarrow \alpha A'$
 $A' \rightarrow \beta_1 \mid \beta_2$

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

$Exp ::= (Exp)$
 $\quad | Exp \ Exp$
 $\quad | ()$

Remove immediate left-recursion



$Exp ::= (Exp) Exp'$
 $\quad | () Exp'$
 $Exp' ::= Exp \ Exp'$
 $\quad | \epsilon$

Left-factored



$Exp ::= (Exp''$
 $Exp'' ::= Exp) Exp'$
 $\quad |) Exp'$
 $Exp' ::= Exp \ Exp'$
 $\quad | \epsilon$

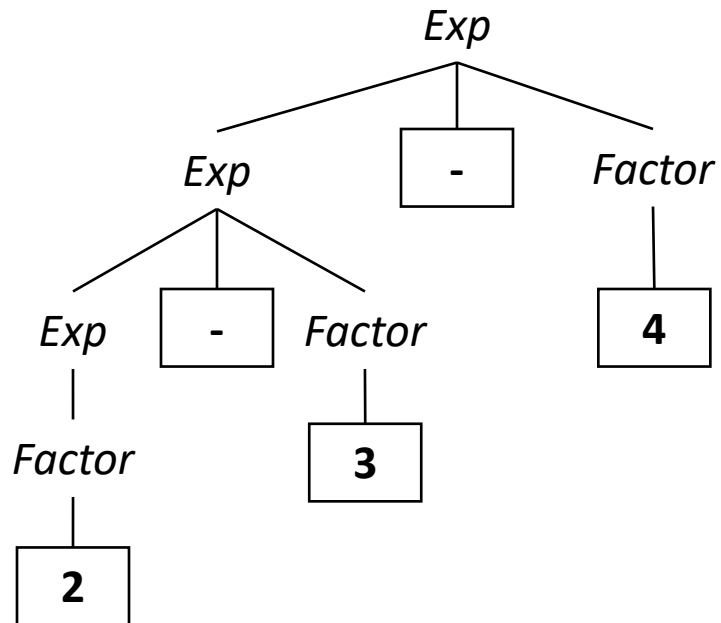
Current Status

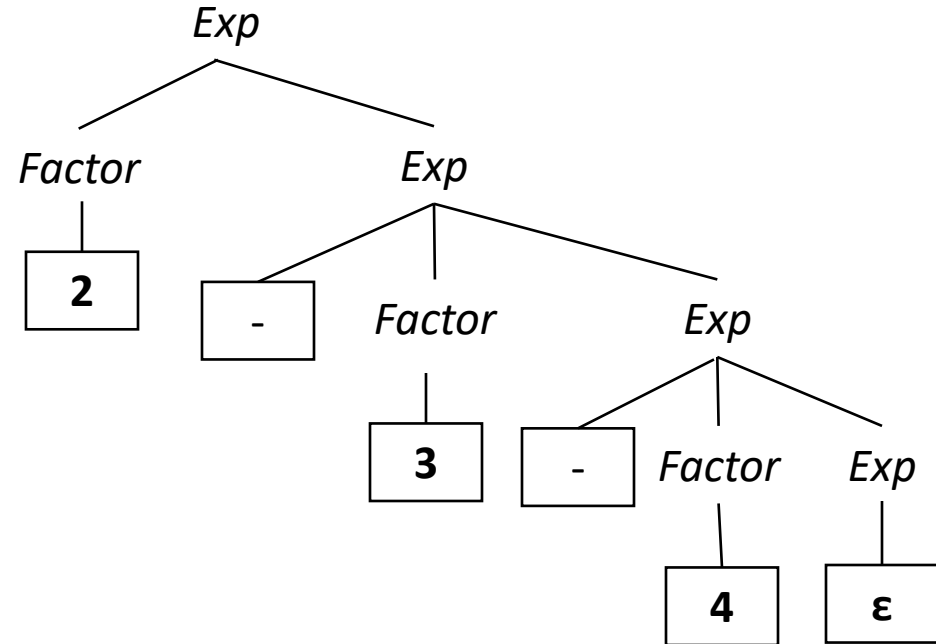
(Predictive) Parsing - LL(1) Transformations

- We've removed 2 disqualifiers from LL(1)
 - Left-recursive grammar
 - **Not** Left-Factored grammar

Let's Check on the Parse Tree

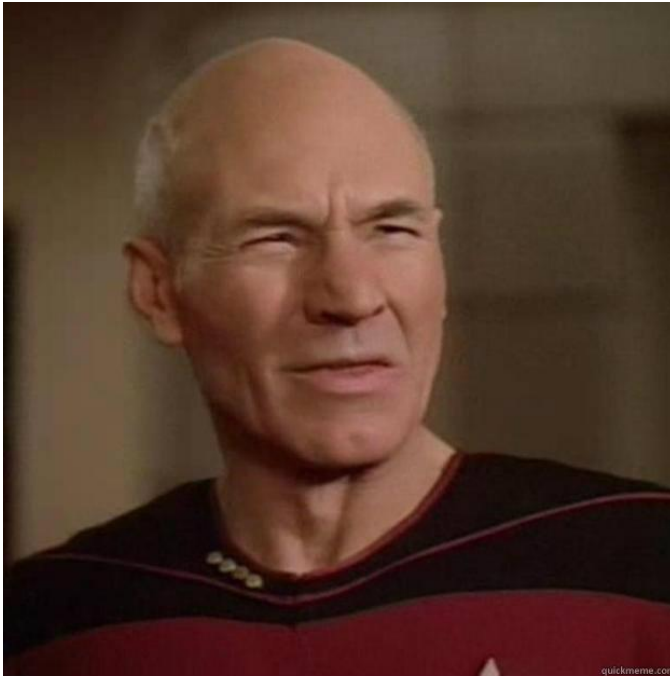
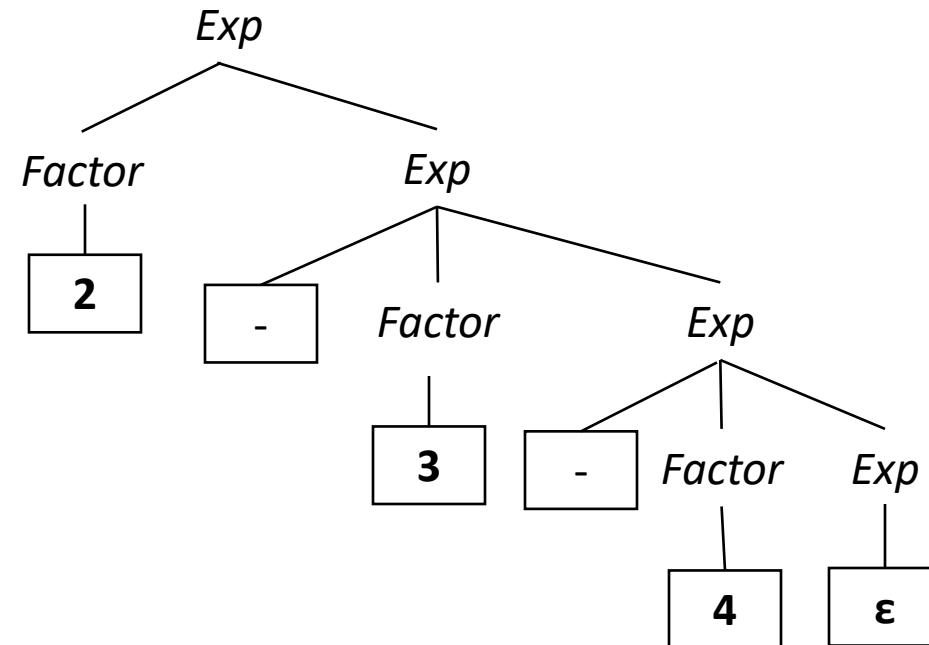
LL(1) Grammar Transformations

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp} - \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow \text{intlit} \mid (\text{Exp}) \end{aligned}$$


$$\begin{aligned} \text{Exp} &\rightarrow \text{Factor Exp}' \\ \text{Exp}' &\rightarrow - \text{Factor Exp}' \\ &\quad | \epsilon \\ \text{Factor} &\rightarrow \text{intlit} \mid (\text{Exp}) \end{aligned}$$


Let's Check on the Parse Tree

LL(1) Grammar Transformations


$$\begin{aligned} \text{Exp} &\rightarrow \text{Factor Exp}' \\ \text{Exp}' &\rightarrow - \text{Factor Exp}' \\ &\quad | \quad \epsilon \\ \text{Factor} &\rightarrow \text{intlit} \mid (\text{Exp}) \end{aligned}$$


Nevermind, We'll Fix Parse Trees Later

LL(1) Grammar Transformations

\\(ツ)\\

Today's Outline

Lecture 9 – FIRST sets

Transforming Grammars



- Fixing LL(1) “near misses”

Building LL(1) Parsers

- Understanding LL(1) Selector Tables
- FIRST Sets



Parsing

Recall the LL(1) Parser's Operation

Building LL(1)Selector Table

LL(1)

- Processes **Left-to-right**
- **Leftmost** derivation
- **1** token of lookahead

Predictive Parser: “guess & check”

- Starts at the root, *guesses* how to unfold a nonterminal (derivation step)
- *Checks* that terminals match prediction

Recall the LL(1) Parser's Operation

Building LL(1) Selector Table

Example LL(1) Grammar:

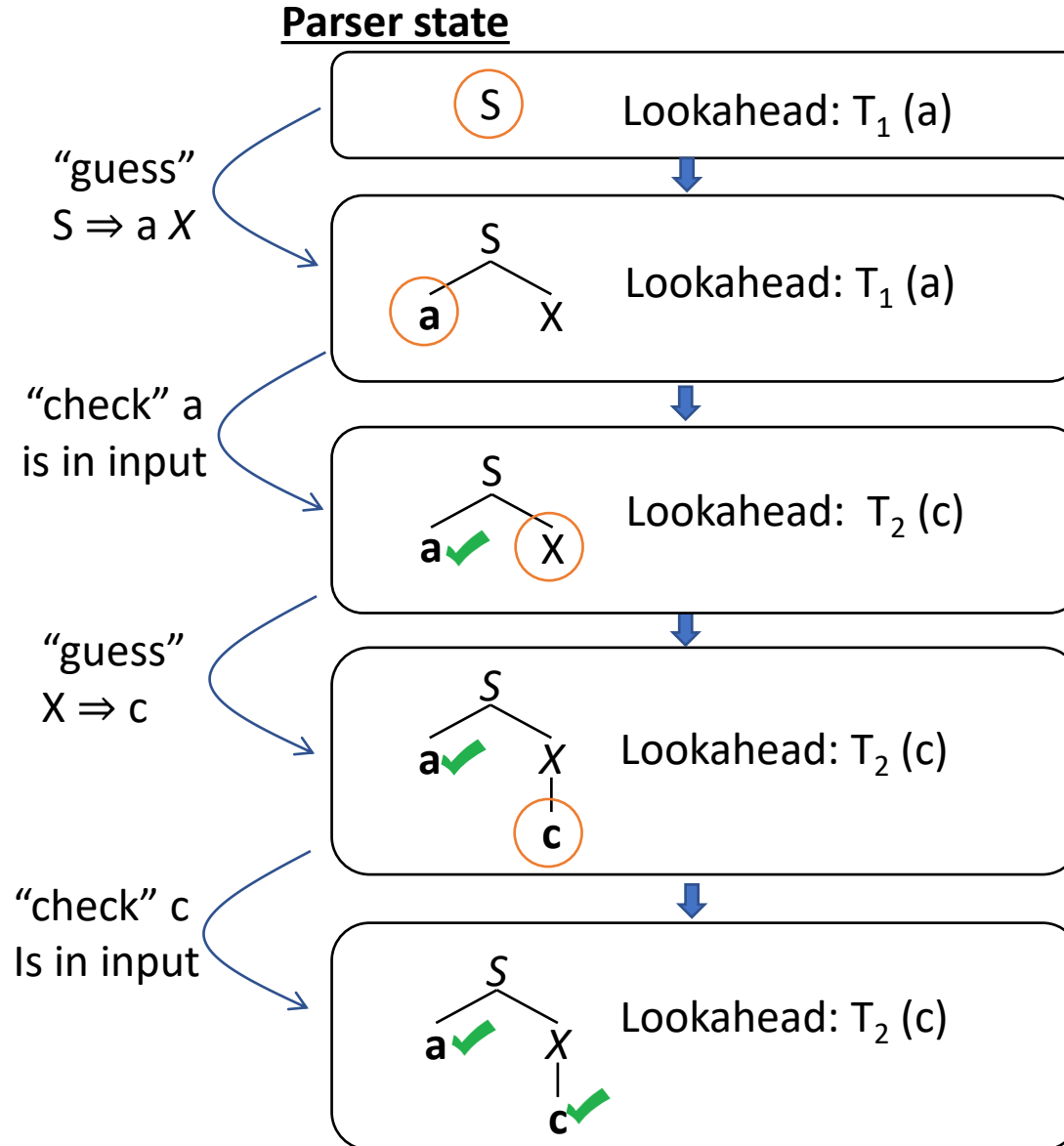
$S ::= a X$

$X ::= b a \mid c$

Example Input:

$a \quad c$
 $\uparrow \quad \uparrow$
 $T_1 \quad T_2$

In practice,
table-driven parser
uses a stack to
match this tree



How does the Parser Guess?

Building Parser Tables

The intuition is a bit tricky

- We need to get into the mindset of the parser



Pretend your consciousness has been transported inside an LL(1) parser

Become the Parser

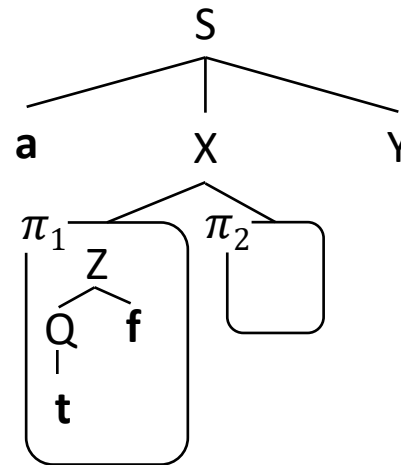
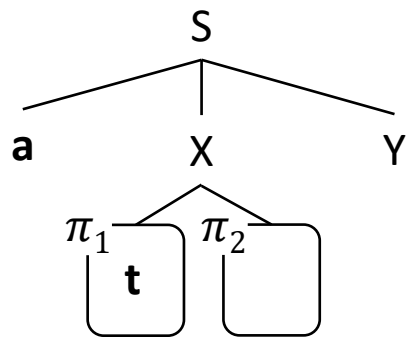
Building Parser Tables

You need to unfold a nonterminal X with lookahead token $\mathbf{t}$

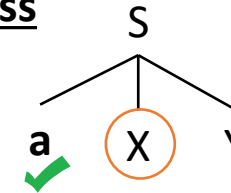
Assume there's an X production $X ::= \pi_1 \pi_2$ (where π_1 and π_2 are some kind of symbol)

How do we know to guess this production?

Case 1: π_1 subtree may start with $\mathbf{t}$



Parse in Progress



Lookahead: $T_2(\mathbf{t})$

Grammar Fragment

...
 $X ::= \pi_1 \pi_2$
 ...

Become the Parser

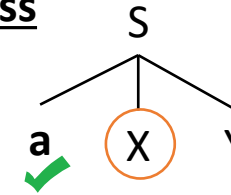
Building Parser Tables

You need to unfold a nonterminal X with lookahead token t

Assume there's an X production $X ::= \pi_1 \pi_2$ (where π_1 and π_2 are some kind of symbol)

How do we know to guess this production?

Parse in Progress

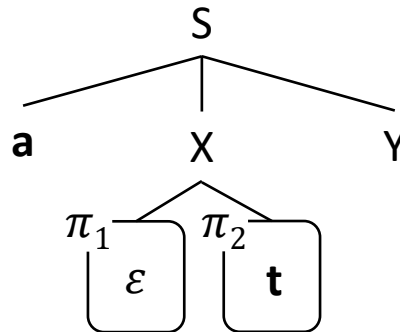


Lookahead: $T_2(t)$

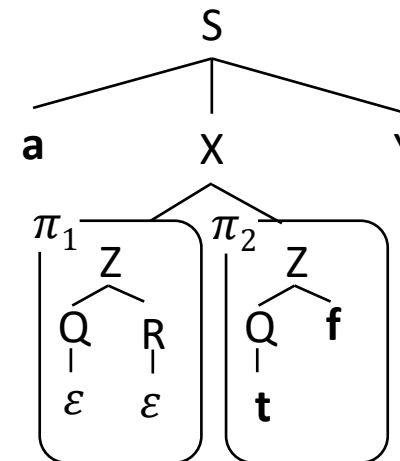
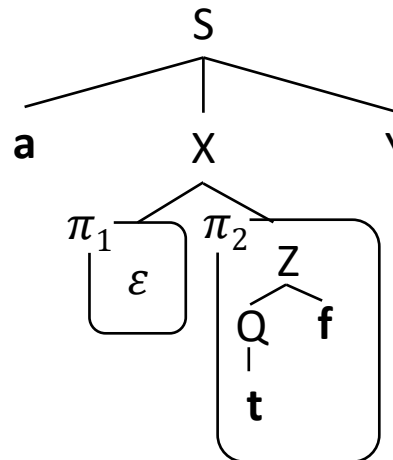
Grammar Fragment

...
 $X ::= \pi_1 \pi_2$
 ...

Case 1: π_1 subtree may start with t



Case 2: π_1 subtree may be empty and π_2 starts with t



Become the Parser

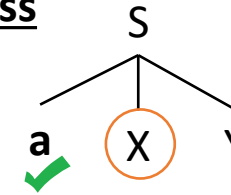
Building Parser Tables

You need to unfold a nonterminal X with lookahead token $\mathbf{t}$

Assume there's an X production $X ::= \pi_1 \pi_2$ (where π_1 and π_2 are some kind of symbol)

How do we know to guess this production?

Parse in Progress

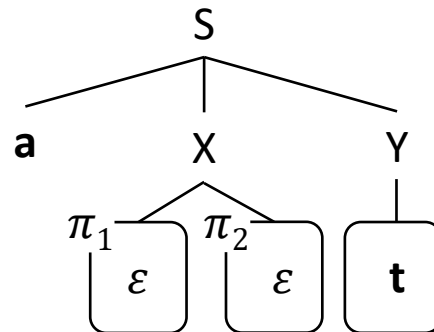


Lookahead: $T_2(\mathbf{t})$

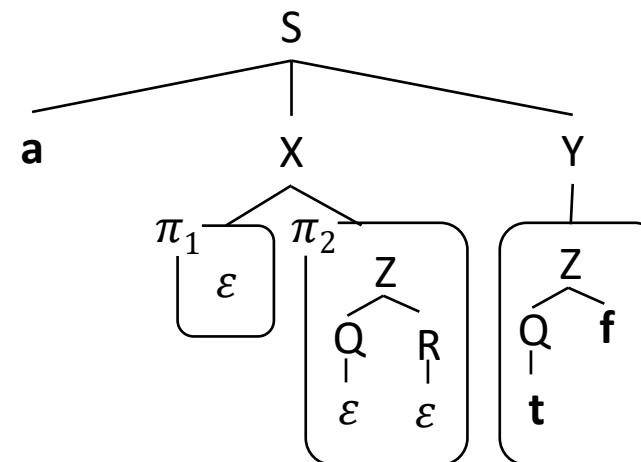
Grammar Fragment

...
 $X ::= \pi_1 \pi_2$
 ...

Case 1: π_1 subtree may start with $\mathbf{t}$



Case 2: π_1 subtree may be empty and π_2 starts with $\mathbf{t}$



Case 3: both π_1 and π_2 may be empty and the sibling may start with $\mathbf{t}$

Become the Parser

Building Parser Tables

You need to unfold a nonterminal X
with lookahead token $\mathbf{t}$

Assume there's an X production $X ::= \pi_1 \pi_2$
(where π_1 and π_2 are some kind of symbol)

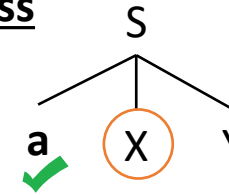
How do we know to guess this production?

Case 1: π_1 subtree
may start with $\mathbf{t}$

Case 2: π_1 subtree may be
empty and π_2 starts with $\mathbf{t}$

Case 3: both π_1 and π_2 may be
empty and the sibling may
start with $\mathbf{t}$

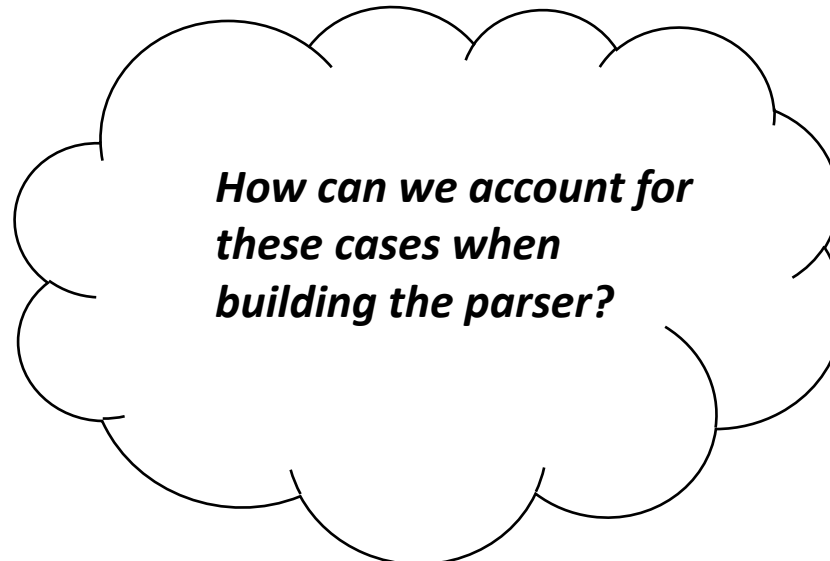
Parse in Progress



Lookahead: $T_2(\mathbf{t})$

Grammar Fragment

...
 $X ::= \pi_1 \pi_2$
...



Become the Parser

Building Parser Tables

You need to unfold a nonterminal X with lookahead token $\mathbf{t}$

Assume there's an X production $X ::= \pi_1 \pi_2$ (where π_1 and π_2 are some kind of symbol)

How do we know to guess this production?

Case 1: π_1 subtree may start with $\mathbf{t}$

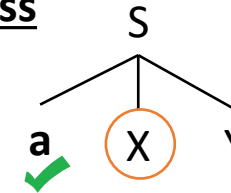
Case 2: π_1 subtree may be empty and π_2 starts with $\mathbf{t}$

Case 3: both π_1 and π_2 may be empty and the sibling may start with $\mathbf{t}$

FIRST Sets

FOLLOW Sets

Parse in Progress



Lookahead: $T_2(\mathbf{t})$

Grammar Fragment

...
 $X ::= \pi_1 \pi_2$
...

Two sets are sufficient to capture these cases and to build the selector table

Today's Outline

Lecture 9 – FIRST sets

Transforming Grammars

- ✓ Fixing LL(1) “near misses”

Building LL(1) Parsers

- ✓ Reverse-Engineering Selector Tables

- FIRST Sets



Parsing

An Informal Definition

Building LL(1) Selector Table: FIRST sets, single symbol

$\text{FIRST}(\alpha)$ = The set of terminals that begin strings derivable from α , and also, if α can derive ϵ , then ϵ is in $\text{FIRST}(\alpha)$.

A Formal Definition

Building LL(1) Selector Table: FIRST sets, single symbol

$\text{FIRST}(\alpha)$ = The set of terminals that begin strings derivable from α , and also, if α can derive ϵ , then ϵ is in $\text{FIRST}(X)$.

Formally, $\text{FIRST}(\alpha) =$

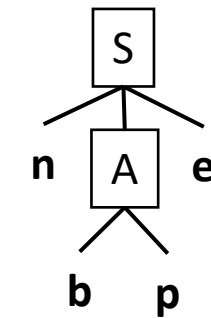
$$\left\{ \hat{\alpha} \mid \left(\hat{\alpha} \in \Sigma \wedge \alpha \xRightarrow{*} \hat{\alpha}\beta \right) \vee \left(\hat{\alpha} = \epsilon \wedge \alpha \xRightarrow{*} \epsilon \right) \right\}$$

A Parse Tree Perspective

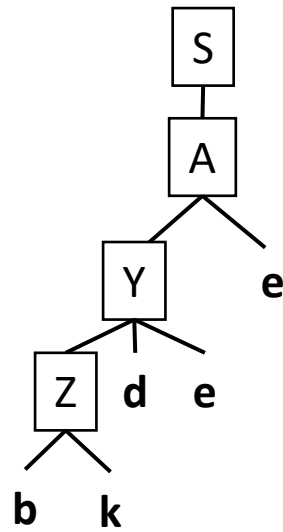
Building LL(1) Selector Table: FIRST sets, single symbol

$\text{FIRST}(\alpha)$ = The set of terminals that begin strings derivable from α , and also, if α can derive ϵ , then ϵ is in $\text{FIRST}(X)$.

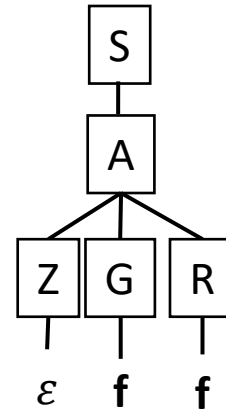
What does the parse tree say about $\text{FIRST}(A)$?



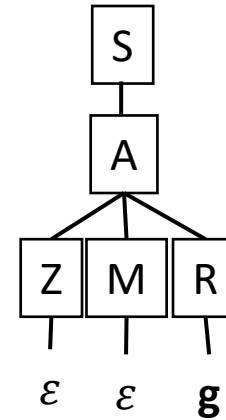
$\text{FIRST}(A)$
includes
 $\{b\}$



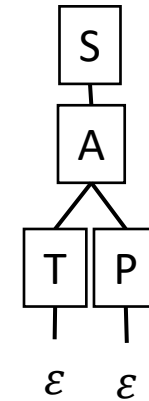
Again, $\text{FIRST}(A)$
includes
 $\{b\}$



$\text{FIRST}(A)$
Includes
 $\{f\}$



$\text{FIRST}(A)$
includes
 $\{g\}$



$\text{FIRST}(A)$
includes
 $\{\epsilon\}$

If these were the only possible parse trees, then $\text{FIRST}(A) = \{b, f, g, \epsilon\}$

A Parse Tree Perspective

Building LL(1) Selector Table: FIRST sets, single symbol

$\text{FIRST}(\alpha)$ = The set of terminals that begin strings derivable from α , and also, if α can derive ϵ , then ϵ is in $\text{FIRST}(X)$.

This isn't how you build FIRST sets

- Looking at parse trees is illustrative for concepts only
- We need to derive FIRST sets directly from the grammar

Building FIRST Sets: Methodology

Building Parser Tables

First sets exist for any arbitrary string of symbols α

- Defined in terms of FIRST sets for a single symbol
 - FIRST of an alphabet terminal
 - FIRST for ϵ
 - FIRST for a nonterminal
- Use single-symbol FIRST to construct symbol-string FIRSTS

Rules for Single Symbols

Building Parser Tables

$\text{FIRST}(X)$ = The set of terminals that begin strings derivable from X , and also, if X can derive ε , then ε is in $\text{FIRST}(X)$.

Building FIRST for terminals

$\text{FIRST}(\mathbf{t}) = \{ \mathbf{t} \}$ for $\mathbf{t}$ in Σ

$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$



Building FIRST(X) for nonterminal X

For each $X ::= \alpha_1 \alpha_2 \dots \alpha_n$

C_1 : add $\text{FIRST}(\alpha_1) - \varepsilon$

C_2 : If ε could “prefix” $\text{FIRST}(\alpha_k)$, add $\text{FIRST}(\alpha_k) - \varepsilon$

C_3 : If ε is in every FIRST set $\alpha_1 \dots \alpha_n$, add ε

Rules for Single Symbols

Building LL(1) Parsers

Building FIRST(X) for nonterminal X

For each $X ::= \alpha_1 \alpha_2 \dots \alpha_n$

C_1 : add FIRST(α_1) - ε

C_2 : If ε could “prefix” FIRST(α_k), add FIRST(α_k) - ε

C_3 : If ε is in every FIRST set $\alpha_1 \dots \alpha_n$, add ε

Rules for Single Symbols

Building LL(1) Parsers

Building FIRST(X) for nonterminal X

For each $X ::= \alpha_1 \alpha_2 \dots \alpha_n$

C_1 : add FIRST(α_1) - ϵ

C_2 : If ϵ could “prefix” FIRST(α_k), add FIRST(α_k) - ϵ

C_3 : If ϵ is in every FIRST set $\alpha_1 \dots \alpha_n$, add ϵ

Say there's a production

$X ::= Y Z R T$

and we know

FIRST(Y) = { ϵ , **a** }

FIRST(Z) = { ϵ , **b**, **m** }

FIRST(R) = { **c** }

FIRST(T) = { **d** }

By C_2 clause FIRST(X) includes **b**, **m** and **c**

b, m because FIRST of every symbol before the 2nd includes ϵ)

***Z** in this case* 

c because FIRST of every symbol before the 3rd includes ϵ)

***R** in this case* 

FIRST(X) does not add **d** in this clause
because not every FIRST set before the T
includes ϵ

Building FIRST Sets for Symbol Strings

Building LL(1) Parsers

Building FIRST(α)

Let α be composed of symbols $\alpha_1 \alpha_2 \dots \alpha_n$

C_1 : add FIRST(α_1) - ε

C_2 : If $\alpha_1 \dots \alpha_{k-1}$ is nullable, add FIRST(α_k) - ε

C_3 : If $\alpha_1 \dots \alpha_n$ is nullable, add ε

Base Cases:

α_i is a terminal t . Add t

α_i is a nonterminal X . Add every leaf symbol that could begin an X subtree
(this gets a bit complicated due to dependencies)

Summary: Explored the LL(1) Mindset

FIRST Sets

LL(1) “Parseability” Qualification

- Knowing the leftmost terminal of a parse (sub)tree is enough to pick the next derivation step

Elusive Conditions

- Two different rules could start with the same terminal (not left factored)
- The same rule(s) could be applied repeatedly (left recursive)

Began choosing matching productions to input

- What terminal could the production be the start of (FIRST)?