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

University of Kansas | Drew Davidson

EECS 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

<u>You Should Know</u>

- 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!

Same language
Different grammars

Grammar 1

❶ S ::= **a b**

❷ | **a c**

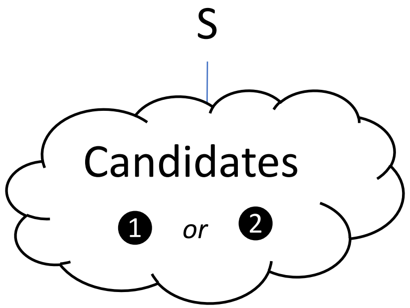
Grammar 2

❶ S ::= **a** X

❷ X ::= **b**

❸ X ::= **c**

Predicted Parse Tree

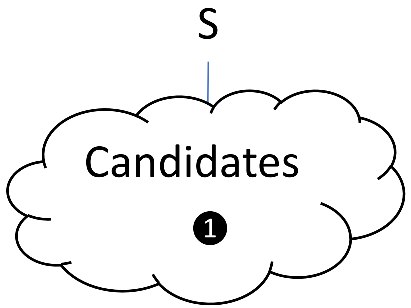


Input String

a

look

Predicted Parse Tree



Lookahead

a

look

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 <u>not</u> LL(1):

- The grammar **is** left-recursive

  X ::= X a | 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 $X \overset{+}{\Rightarrow} X\ \alpha$ 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

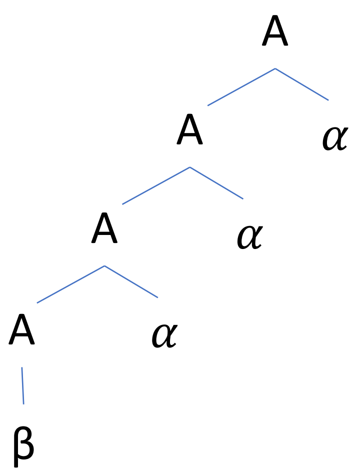
(Predictive) Parsing - LL(1) Transformations

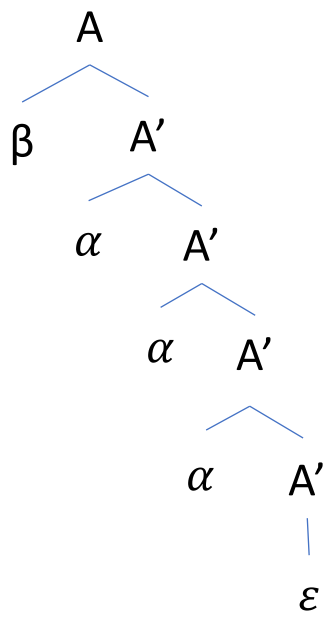
(for a single immediately left-recursive rule)

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

Arbitrary Strings
(nonterminal or terminal)

$$A \rightarrow \beta\ A'$$
$$A' \rightarrow \alpha\ A' \mid \varepsilon$$





Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

$$\underbrace{A}_{Exp} \rightarrow \overbrace{A}^{Exp}\ \overbrace{\alpha}^{-\ Factor} \mid \underbrace{\beta}_{Factor}$$

$$\overbrace{A}^{Exp} \rightarrow \overbrace{\beta}^{Factor}\ \overbrace{A'}^{Exp'}$$
$$\overbrace{A'}^{Exp'} \rightarrow \alpha\ \overbrace{A'}^{Exp'}$$
$$\mid\ \underbrace{\varepsilon}_{-\ Factor}$$

$$\overbrace{\boxed{Exp}}^{A} ::= \overbrace{\boxed{Exp}}^{A}\ \overbrace{\boxed{-\ Factor}}^{\alpha}$$
$$\mid\ \underbrace{\boxed{Factor}}_{\beta}$$

Factor ::= **intlit** | **(** Exp **)**

$$\overbrace{\boxed{Exp}}^{A} ::= \overbrace{\boxed{Factor}}^{\beta}\ \overbrace{\boxed{Exp'}}^{A'}$$
$$\underbrace{\boxed{Exp'}}_{A'} ::= \underbrace{\boxed{-\ Factor}}_{\alpha}\ \boxed{Exp'}\ A'$$
$$\mid\ \varepsilon$$

Factor ::= **intlit** | **(** Exp **)**

Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

(general rule)

Given Productions

$$
\begin{aligned}
A ::=\ & \beta_1 \\
\mid\ & \beta_2 \\
\mid\ & \beta_n \\
\mid\ & A\ \alpha_1 \\
\mid\ & A\ \alpha_2 \\
\mid\ & A\ \alpha_m
\end{aligned}
$$

Convert to

$$
\begin{aligned}
A\ ::=\ & \beta_1\ A' \\
\mid\ & \beta_2\ A' \\
\mid\ & \beta_n\ A' \\
A'\ ::=\ & \alpha_1\ A' \\
\mid\ & \alpha_2\ A' \\
\mid\ & \alpha_m\ A' \\
\mid\ & \varepsilon
\end{aligned}
$$

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* **}**
| **( )**
| **a b**
| **b b**

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

$$\begin{array}{l} \text{X} ::= \overbrace{\boxed{\textbf{a b}}}^{\alpha}\ \overbrace{\boxed{\textbf{c d}}}^{\beta_1} \\ \text{X} ::= \boxed{\textbf{a b}}\ \underbrace{\boxed{\textbf{e f}}}_{\beta_2} \end{array}$$

X ::= **a b** *X'*
X' ::= **c d** | **e f**

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

$$\overset{A}{\boxed{Exp}} ::= \overset{\beta_1}{\boxed{(\ Exp\ )}}$$
$$|\ \overset{A}{\boxed{Exp}}\ \overset{\alpha_1}{\boxed{Exp}}$$
$$|\ \overset{\beta_2}{\boxed{(\ )}}$$

Remove immediate left-recursion

$$\overset{A}{\boxed{Exp}} ::= \overset{\beta_1}{\boxed{(\ Exp\ )}}\ \overset{A'}{\boxed{Exp'}}$$
$$|\ \overset{\beta_2}{\boxed{(\ )}}\ \overset{A'}{\boxed{Exp'}}$$
$$\overset{A'}{\boxed{Exp'}} ::= \overset{\alpha_1}{\boxed{Exp}}\ \overset{A'}{\boxed{Exp'}}$$
$$|\ \overset{\text{new } \varepsilon}{\boxed{\varepsilon}}$$

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

becomes

$$A \rightarrow \beta\ A'$$
$$A' \rightarrow \alpha\ A'$$
$$\mid\ \varepsilon$$

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

$$Exp ::= \textbf{(}\ Exp\ \textbf{)} \mid Exp\ \ Exp \mid \textbf{(}\ \textbf{)}$$

Remove immediate left-recursion ✔

$$\overset{A}{\boxed{Exp}} ::= \overset{\alpha}{\boxed{\textbf{(}}}\ \overset{\beta_1}{\boxed{Exp\ \textbf{)}\ \ Exp'}}$$
$$\mid \overset{\alpha}{\boxed{\textbf{(}}}\ \overset{\beta_2}{\boxed{\textbf{)}\ \ Exp'}}$$
$$Exp' ::= Exp\ \ Exp' \mid \varepsilon$$

Left-factored

$$\overset{A}{\boxed{Exp}} ::= \overset{\alpha}{\boxed{\textbf{(}}}\ \overset{A'}{\boxed{Exp''}}$$
$$\overset{A'}{\boxed{Exp''}} ::= \overset{\beta_1}{\boxed{Exp\ \textbf{)}\ Exp'}}$$
$$\mid \overset{\beta_2}{\boxed{\textbf{)}\ Exp'}}$$
$$\mathrm{Exp}' ::= Exp\ Exp' \mid \varepsilon$$

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

Attempt LL(1) Conversion

(Predictive) Parsing - LL(1) Transformations

$$
\begin{aligned}
Exp &::= \mathbf{(}\ Exp\ \mathbf{)} \\
&\mid Exp\ \ Exp \\
&\mid \mathbf{(\ )}
\end{aligned}
$$

Remove immediate left-recursion

$$
\begin{aligned}
Exp &::= \mathbf{(}\ Exp\ \mathbf{)}\ \ Exp' \\
&\mid \mathbf{(\ )}\ \ Exp' \\
Exp' &::= Exp\ \ Exp' \\
&\mid \varepsilon
\end{aligned}
$$

Left-factored

$$
\begin{aligned}
Exp &::= \mathbf{(}\ \ Exp'' \\
Exp'' &::= Exp\ \mathbf{)}\ Exp' \\
&\mid \mathbf{)}\ Exp' \\
Exp' &::= Exp\ Exp' \\
&\mid \varepsilon
\end{aligned}
$$

Current Status

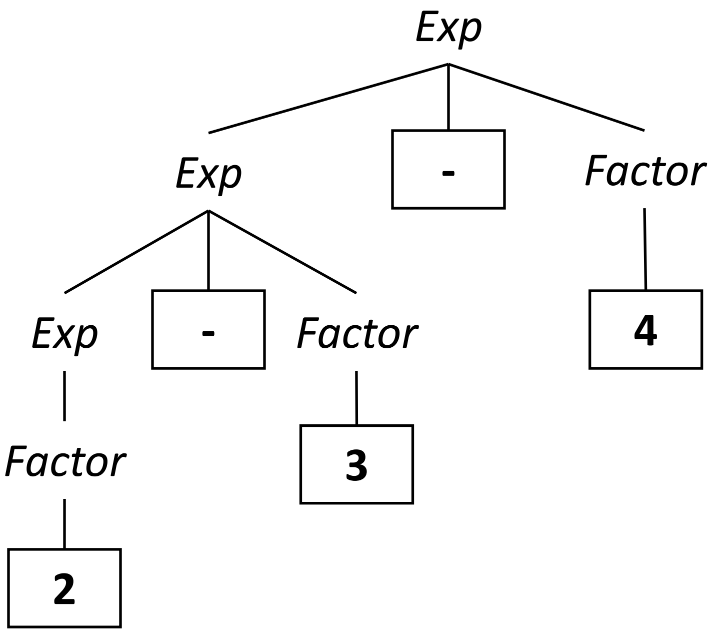
(Predictive) Parsing - LL(1) Transformations

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

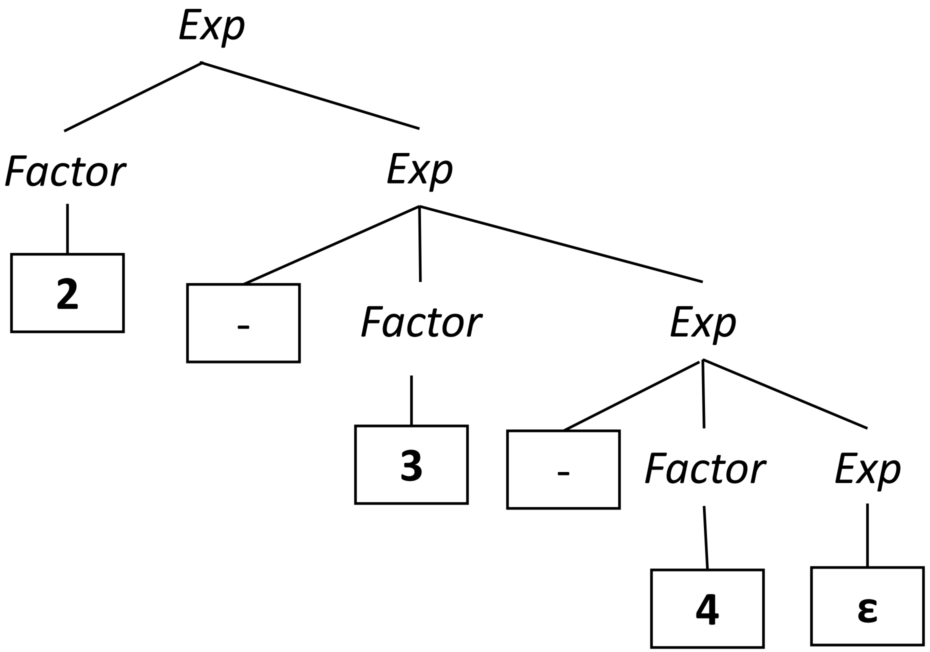
Let’s Check on the Parse Tree

LL(1) Grammar Transformations

Exp → *Exp* **–** *Factor*
| *Factor*
Factor → **intlit** | **(** Exp **)**

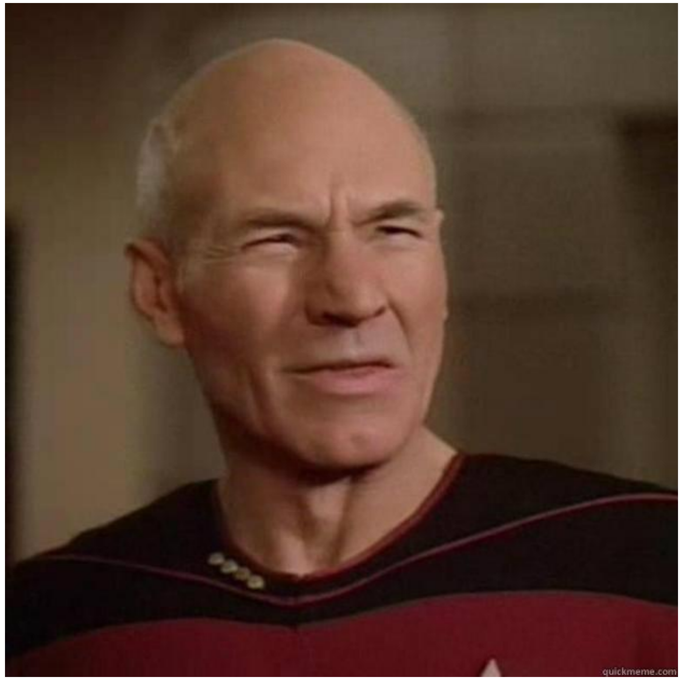


Exp → *Factor Exp’*
Exp’ → - *Factor Exp’*
| ε
Factor → **intlit** | **(** Exp **)**



Let’s Check on the Parse Tree

LL(1) Grammar Transformations

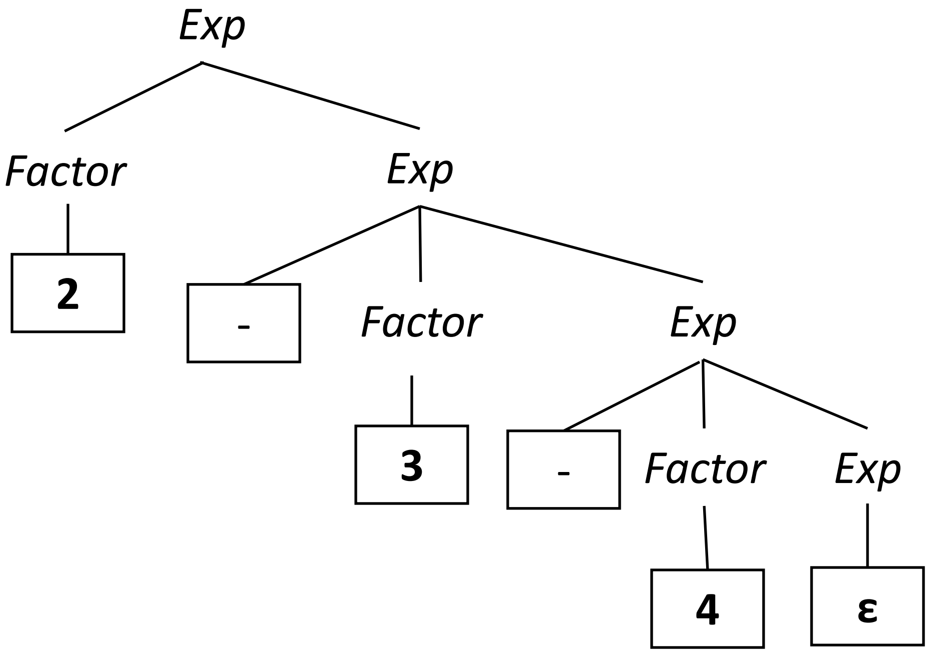


Exp → *Factor Exp’*

Exp’ → - *Factor Exp’*

| ε

Factor → **intlit** | **(** Exp **)**



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 **L**eft-to-right
- **L**eftmost 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*

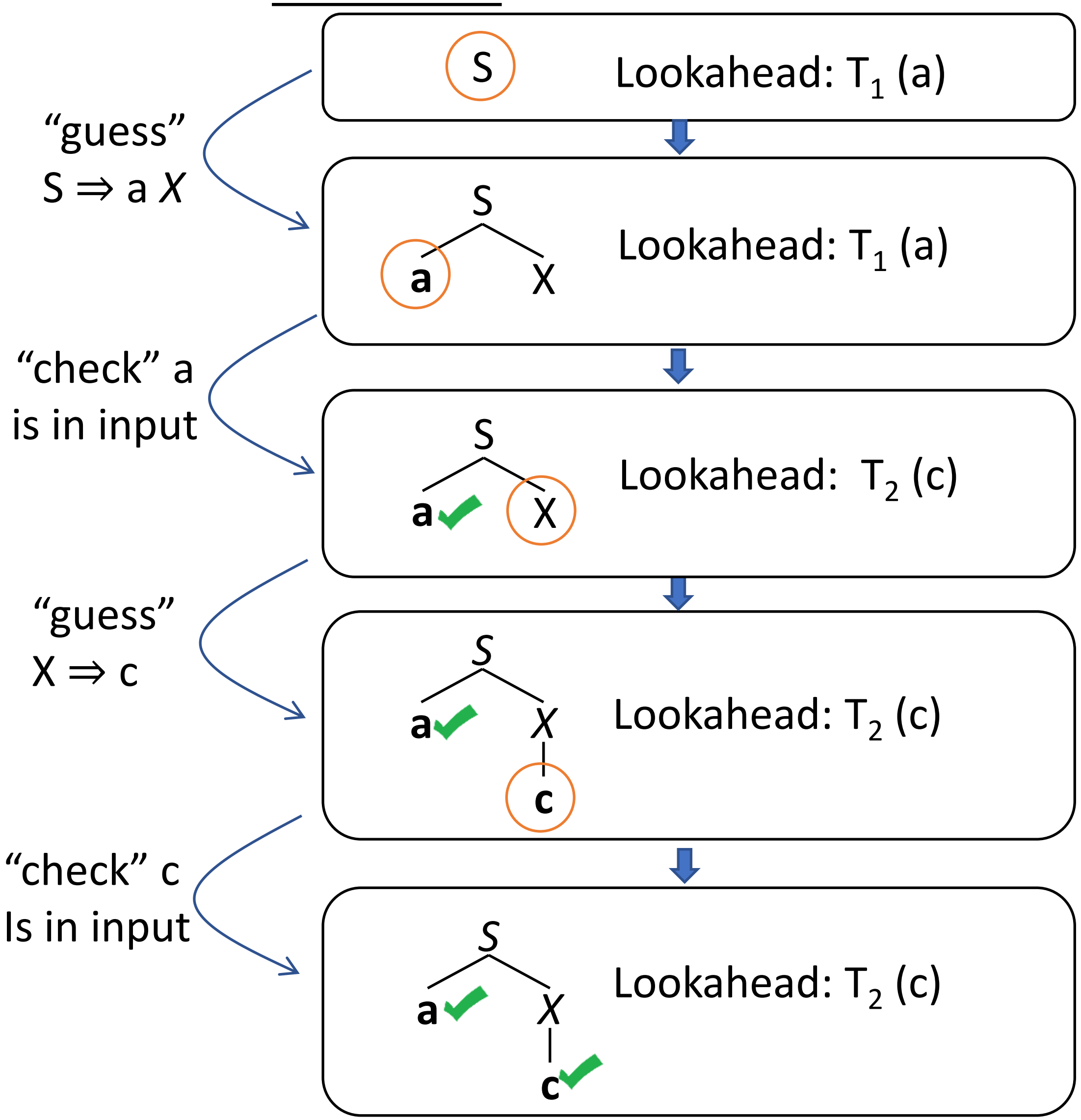
X ::= **b a** | **c**

Example Input:

a c

T_1 T_2

Parser state



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

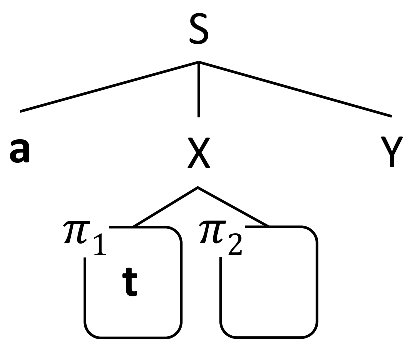
Building Parser Tables

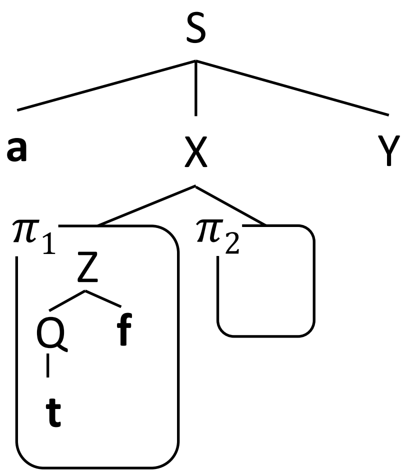
You need to unfold a nonterminal *X* with lookahead token **t**

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

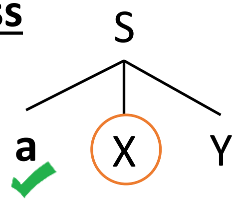
How do we know to guess this production?

Case 1: π_1 subtree may start with **t**





Parse in Progress



Lookahead: T_2 (**t**)

Grammar Fragment

…

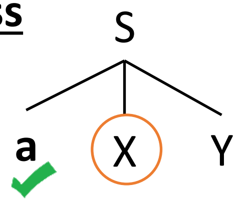
X ::= $\pi_1\ \pi_2$

…

Become the Parser

Building Parser Tables

Parse in Progress



Lookahead: T_2 (**t**)

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?

Grammar Fragment

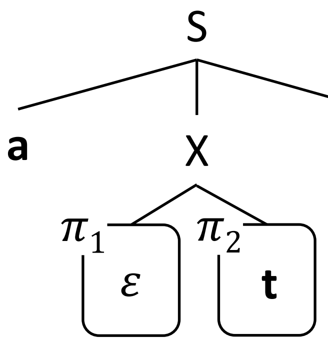
…

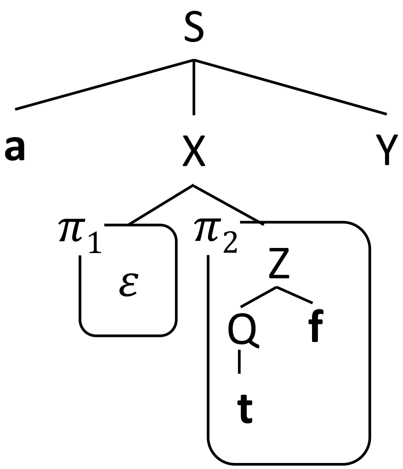
$X ::= \pi_1\ \pi_2$

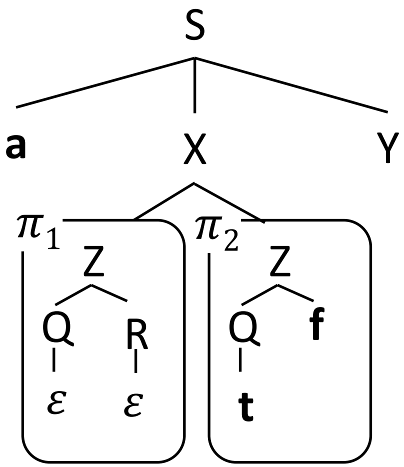
…

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

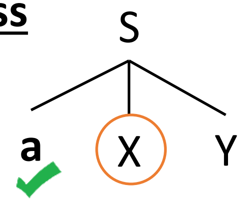
Building Parser Tables

You need to unfold a nonterminal *X* with lookahead token **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

…

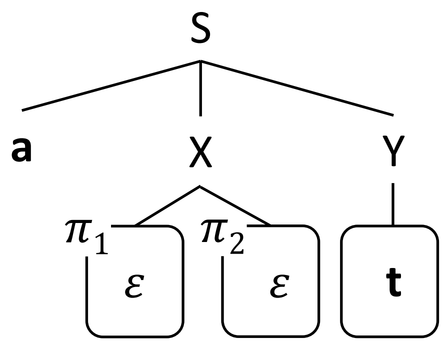
$X ::= \pi_1\ \pi_2$

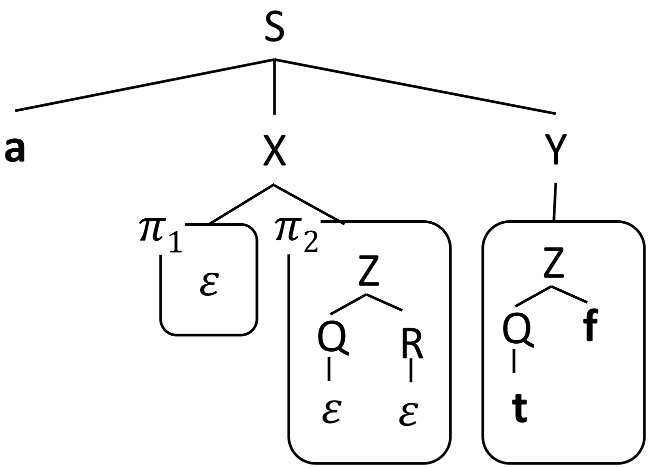
…

Case 1: π_1 subtree may start with **t**

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

Case 3: both π_1 and π_2 may be empty and the sibling may start with **t**





Become the Parser

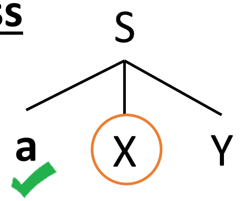
Building Parser Tables

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**

Case 3: both π_1 and π_2 may be empty and the sibling may start with **t**

How can we account for these cases when building the parser?

Become the Parser

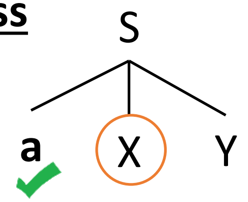
Building Parser Tables

You need to unfold a nonterminal *X* with lookahead token **t**

Assume there's an *X* production *X ::=* π_1 π_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 ::= π_1 π_2

…

Case 1: π_1 subtree may start with **t**

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

Case 3: both π_1 and π_2 may be empty and the sibling may start with **t**

Cases 1 and 2: **FIRST Sets**

Case 3: **FOLLOW Sets**

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

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

A Formal Definition

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

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

Formally, FIRST(α) =

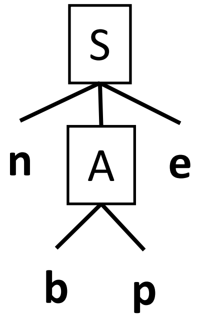
$$\left\{ \hat{\alpha} \;\middle|\; \left(\hat{\alpha} \in \Sigma \land \alpha \overset{*}{\Rightarrow} \hat{\alpha}\beta\right) \lor \left(\hat{\alpha} = \varepsilon \land \alpha \overset{*}{\Rightarrow} \varepsilon\right) \right\}$$

A Parse Tree Perspective

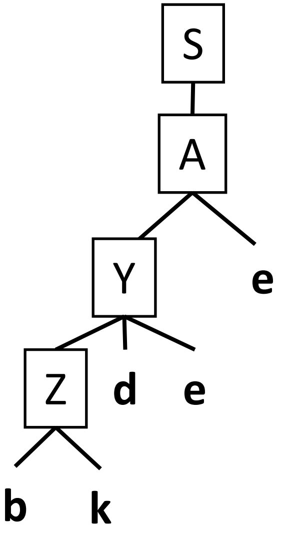
Building LL(1) Selector Table: FIRST sets, single symbol

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

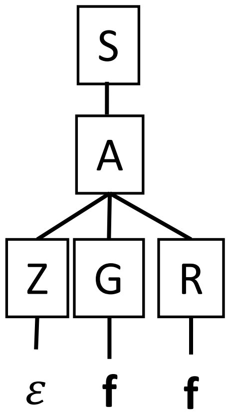
What does the parse tree say about FIRST(*A*)?



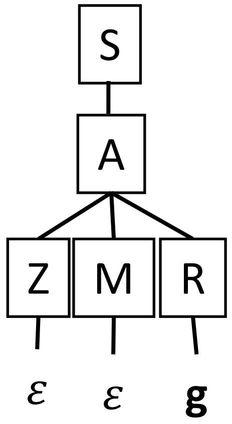
FIRST(A) includes { **b** }



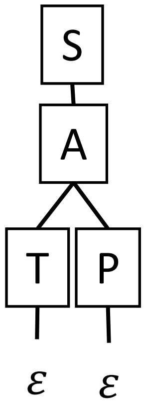
Again, FIRST(A) includes { **b** }



FIRST(A) Includes { **f** }



FIRST(A) includes { **g** }



FIRST(A) includes { ε }

If these were the only possible parse trees, then FIRST(*A*) = { **b**, f, **g,** ε }

A Parse Tree Perspective

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

FIRST(α) = The set of terminals that begin strings derivable from α, and also, if α can derive ε, then ε is in 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

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

Building FIRST for terminals

FIRST(**t**) = { **t** } for **t** in Σ

FIRST(ε) = { ε }

Building FIRST(*X*) for nonterminal *X*

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

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 $\alpha_1 \ldots \alpha_n$, add ε

Rules for Single Symbols

Building LL(1) Parsers

Building FIRST(*X*) for nonterminal *X*

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

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 $\alpha_1 \ldots \alpha_n$, 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

<u>Building FIRST(</u>α<u>)</u>

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

C_1: add FIRST(α_1) - ε

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

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

Base Cases:

α_i is is a terminal **t**. Add **t**

α_i is 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)?