Build Selector Table

Review: LL(1) SDT

Build the selector table for this grammar

Grammar

❶ *S* ::= **lpar** *X* **rpar**

❷ *X* ::= **id comma** *X*

❸ *X* ::= ε

FIRST(S) = { **lpar** }

FIRST(X) = { ε , **id** }

FOLLOW(S) = { **eof** }

FOLLOW(*X*) = { **rpar** }

Building Selector Tables

for each production X ::= α
    for each t in FIRST(α)
        put X ::= α in Table[X][t]
    if ε is in FIRST(α)
        for each t in FOLLOW(X)
            put X ::= α in Table[X][t]

	lpar	id	comma	rpar	eof
S					
X					

Build Selector Table

Review: LL(1) SDT

Build the selector table for this grammar

Grammar

❶ *S* ::= **lpar** *X* **rpar**

❷ *X* ::= **id comma** *X*

❸ *X* ::= ε

FIRST(S) = { **lpar** }

FIRST(X) = { ε , **id** }

FOLLOW(S) = { **eof** }

FOLLOW(*X*) = { **rpar** }

Building Selector Tables

for each production X ::= α
    for each t in FIRST(α)
        put X ::= α in Table[X][t]
    if ε is in FIRST(α)
        for each t in FOLLOW(X)
            put X ::= α in Table[X][t]

	lpar	id	comma	rpar	eof
S					
X					

Administrivia

Housekeeping

Review

Catch-Up

Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

E ::= *E* **minus** *T* #1
| *T*

T ::= #2 **intlit**

	intlit	**minus**	**EOF**
E	*T E'*		
E'		**minus** *T* #1 *E'*	ε
T	#2 **intlit**		

Eval Stack Actions

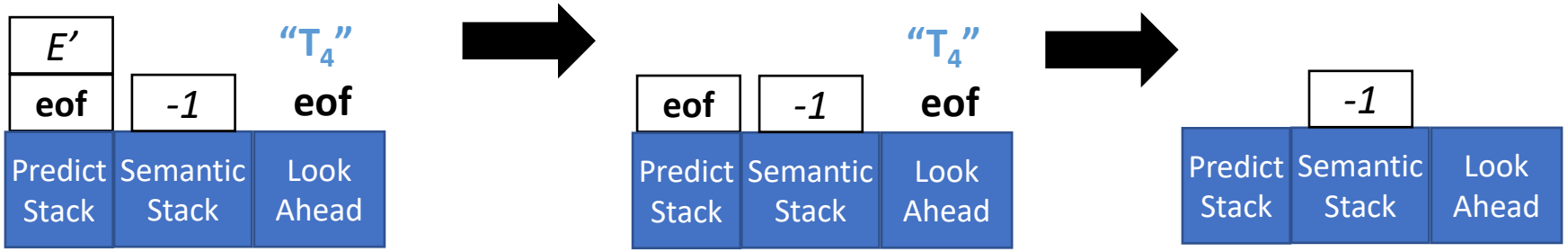
#1 tTrans = sem.pop() ;
eTrans =sem. pop() ;
LTrans = eTrans - tTrans;
sem.push(LTrans)

#2 sem.push(**look**.value)

Token Stream: **intlit:7** **minus** **intlit:8** **eof**

T_1 T_2 T_3 T_4

Accept!



Neat Trick! So What?

SDT for Top-Down Parsing

Expression evaluation is great and all…

- But what we really want is a translation to an AST
- Let's see a similar example

What About Producing ASTs?

SDT for Top-Down Parsing

Augmented CFG

E ::= *E* + *T* #1
| *T*

T ::= #2 **intlit**

	intlit	+	**EOF**
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

Take this out

Eval Stack Actions

#1 tTrans = sem.pop() ;
eTrans =sem. pop() ;
LTrans = eTrans + tTrans;
sem.push(LTrans)

#2 sem.push(**intlit**.value)

Put this in

AST Building Stack Actions

#1 tTrans = sem.pop() ;
eTrans = sem.pop() ;
LTrans = PlusNode(eTrans,tTrans)
sem.push(LTrans)

#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

What About the AST?

SDT for Top-Down Parsing

Selector Table

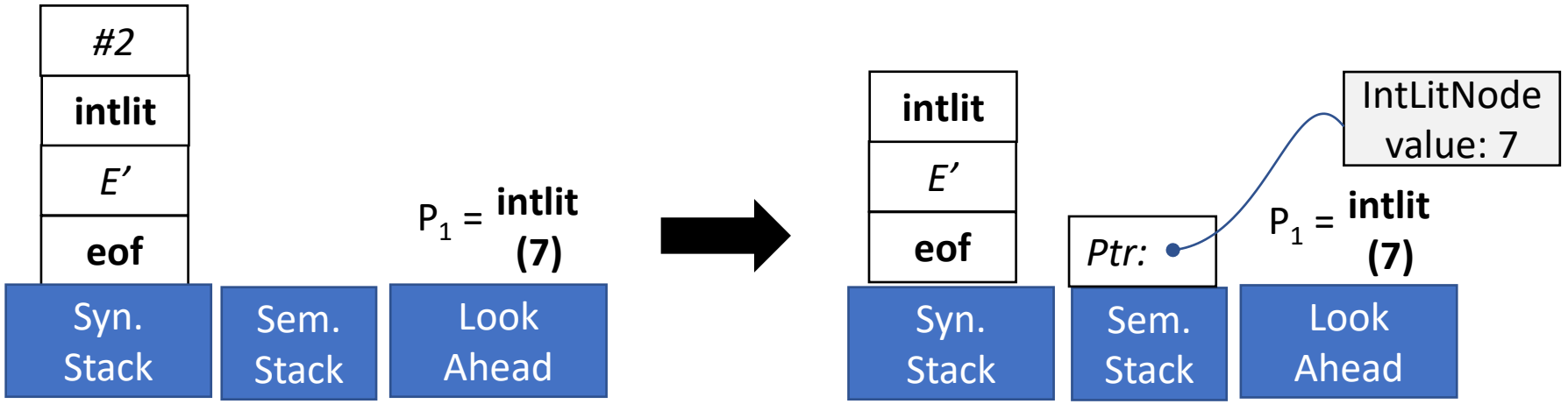
	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

AST Building Stack Actions

#1 tTrans = sem.pop() ;
eTrans = sem.pop() ;
LTrans = PlusNode(eTrans,tTrans)
sem.push(LTrans)

#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

Token Stream: 7 + 8 + 9



Transitions

What About the AST?

SDT for Top-Down Parsing

Selector Table

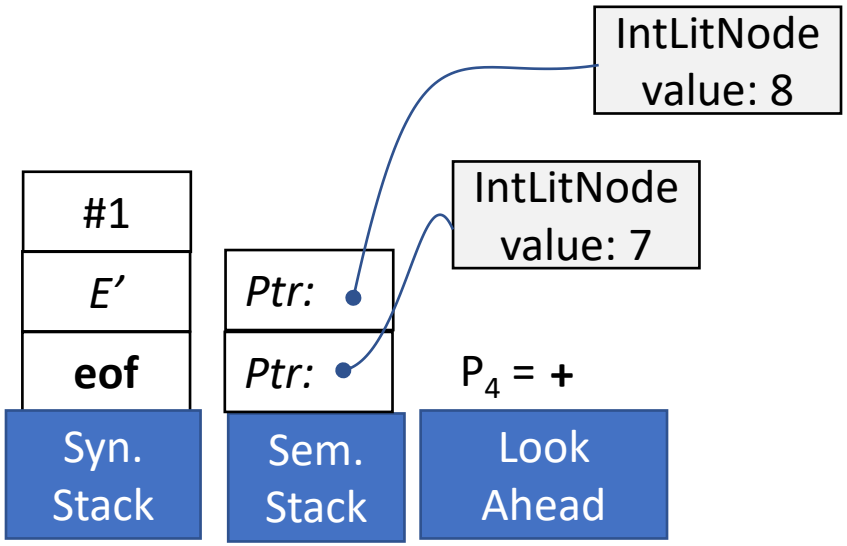
	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

AST Building Stack Actions

#1 tTrans = sem.pop() ;
eTrans = sem.pop() ;
LTrans = PlusNode(eTrans,tTrans)
sem.push(LTrans)

#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

Input: 7 + 8 + 9



What About the AST?

SDT for Top-Down Parsing

Selector Table

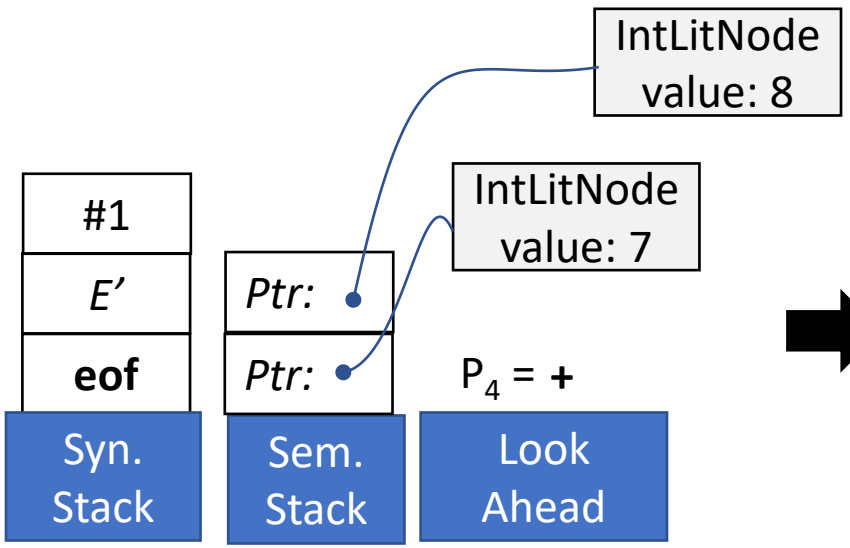
	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

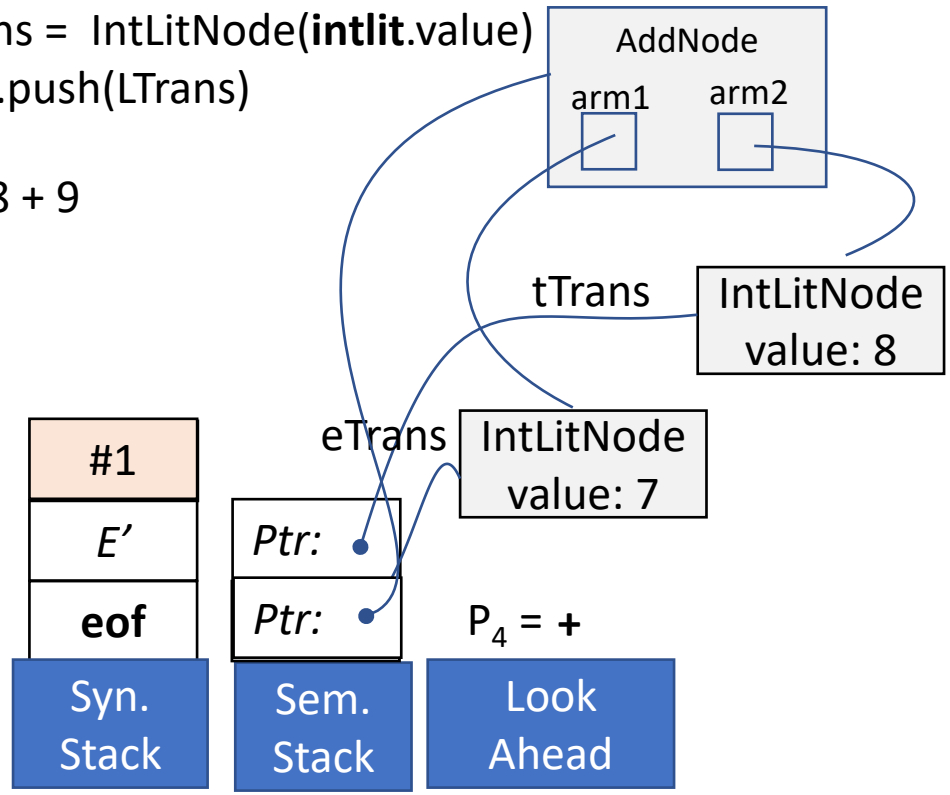
AST Building Stack Actions

#1 tTrans = sem.pop() ;
eTrans = sem.pop() ;
LTrans = PlusNode(eTrans,tTrans)
sem.push(LTrans)

#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

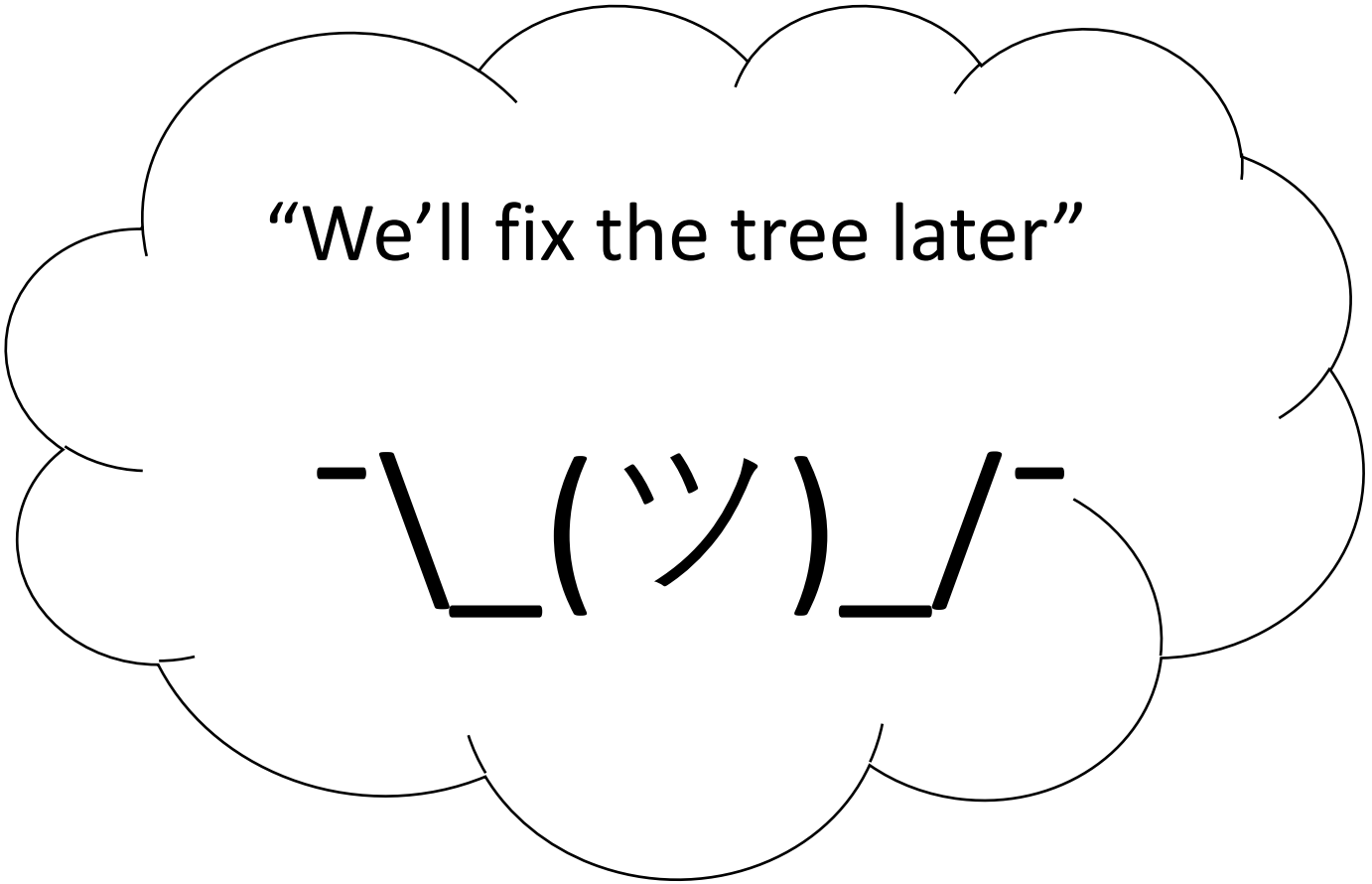
Input: 7 + 8 + 9





This is us “Fixing” the Tree

SDT for Top-Down Parsing



The parse tree is busted, but the AST comes out A-ok!

Summary

SDT for Top-Down Parsing

We've Completed one (1) way of deriving a parse tree:

- Added action numbers to grammar
- Converted SDD rules to stack actions
- Built an LL(1) parser from the grammar

Next Time

SDT for Top-Down Parsing

Handling more complicated languages

- Moving beyond the limitations of LL(1)

University of Kansas | EECS | Drew Davidson

LR Parsing

Last Time

Review LL(1) SDT

LL(1) SDT

- Add 2nd stack for symbol translations
- Turn SDD to semantic stack actions
- Embed triggers into syntactic stack

You should know

How to convert from SDD to stack actions

How to run SDT for an LL(1) parser

Parsing

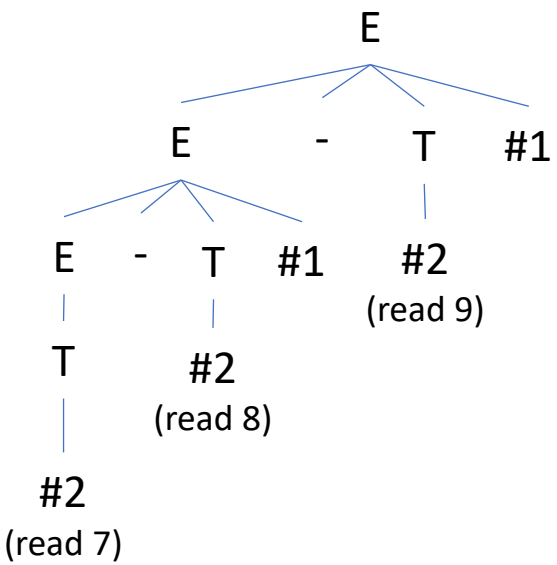
Actions Preserved Through Transforms

3 Grammars for Subtraction Expression:

intlit(7) – intlit(8) – intlit(9)

Augmented Left-Assoc CFG

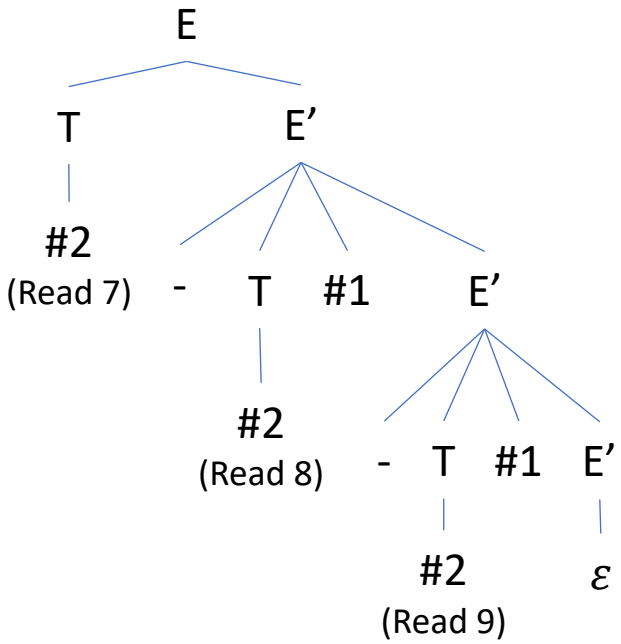
E ::= *E* - *T* #1
| *T*
T ::= #2 **intlit**



#2 #2 #1 #2 #1
7 8 -1 9 -10

LL(1) CFG

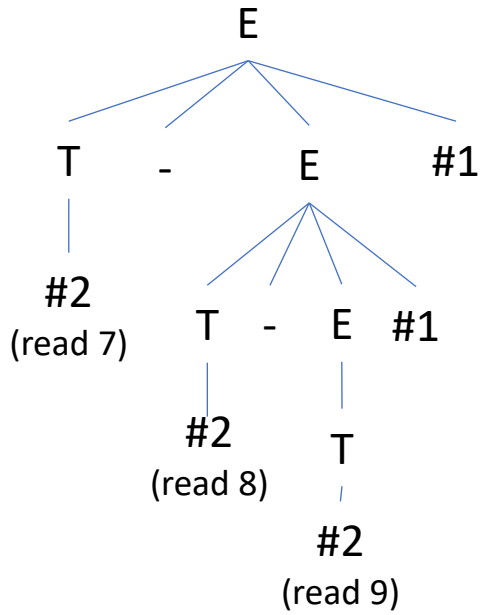
E ::= *T E'*
E' ::= - *T* #1 *E'* | ε
T ::= #2 **intlit**



#2 #2 #1 #2 #1
7 8 -1 9 -10

Augmented Right-Assoc CFG

E ::= *T* - *E* #1
| *T*
T ::= #2 **intlit**



#2 #2 #2 #1 #1
7 8 9 -1 -10

Today's Outline

Lecture 11 – LR Parsing

LL(1) Wrap-up

- Limitations

LR Parsers

- Concept
- Theory
- Operation

Parsing

LL(1) Summary

LL(1) Wrap-Up

Predictive Parser

- Commits to an entire production based on observed lookahead. For LL(1), predict a production based on 1 token of lookahead

Implementation

- Stack and a single streamable token

Many Useful Languages are NOT LL(1)

LL(1) Wrap-Up

Could increase lookahead

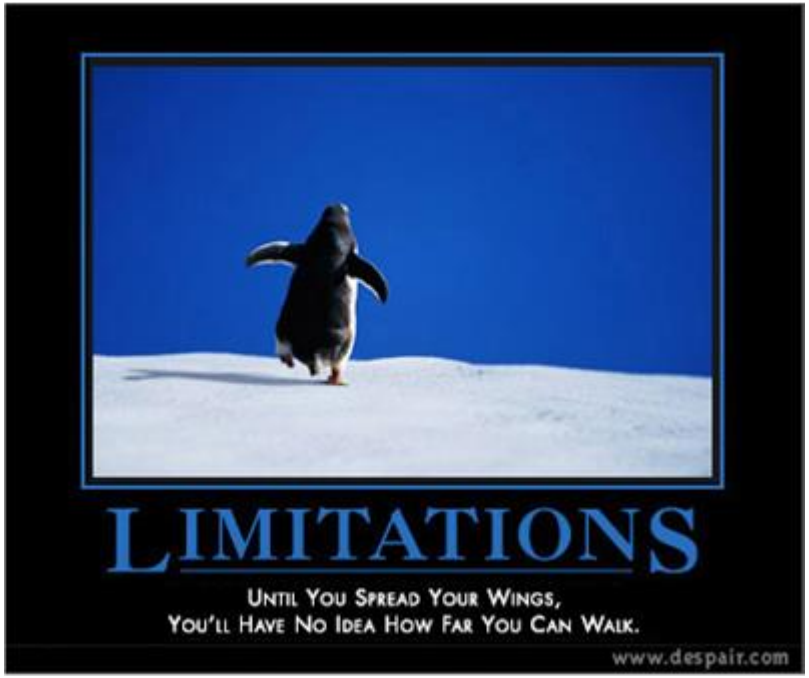
- Up until the mid 90s, this was considered impractical

Could add runtime complexity

- CYK has us covered there

Could add memory complexity

- i.e. more elaborate parse table



LR Parsers

LR Parsers - Concept

Advantages

- Suitable for most programming languages
- Runtime: time and space O(n) in the input size
- Subsumes corresponding LL parser i.e. LL(1) < LR(1)
- LR(1) parsers can recognize any Deterministic CFG

Disadvantages

- More complex parser generation
- Larger memory state

Linear Parsers: Commitment Issues

LR Parsers - Concept

Linear Parser have to commit to their parse tree in a linear scan

LL(1) Parser:
commits ASAP

Backtracking Parsers:
Go back on commitments

LR Parsers:
Delay commitments

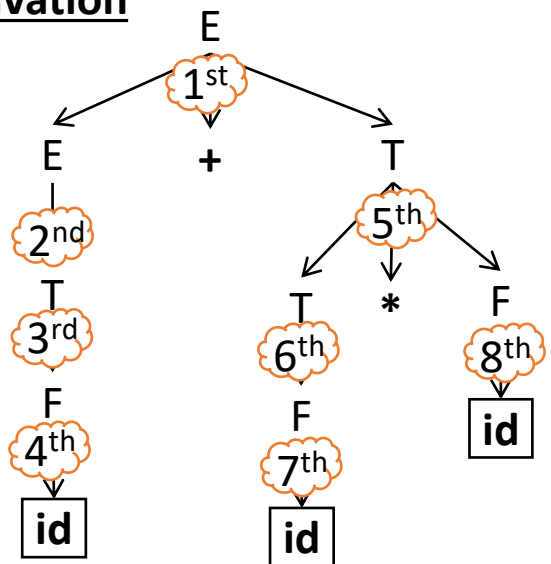
Delayed Commitment to a Rule

LR Parsers - Concept

Can we choose a production *after* we've seen it's subtree?

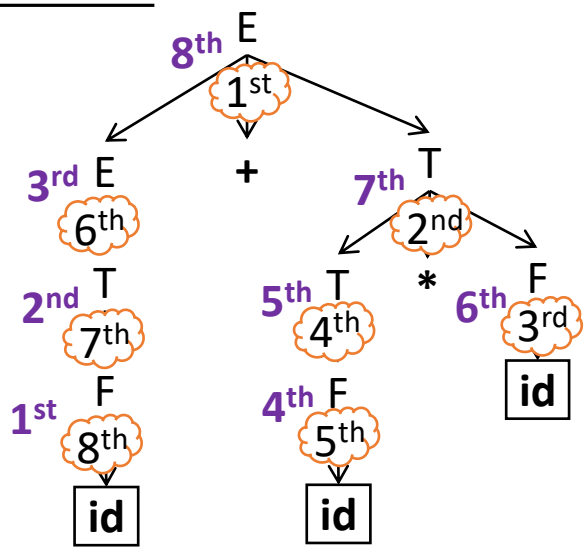
- Yes, if we build the tree *bottom-up*!
- What about derivation order?

Leftmost Derivation



Reverse

Rightmost Derivation



LR(k) Parsers

LR Parsers - Concept

Perform reverse rightmost derivation

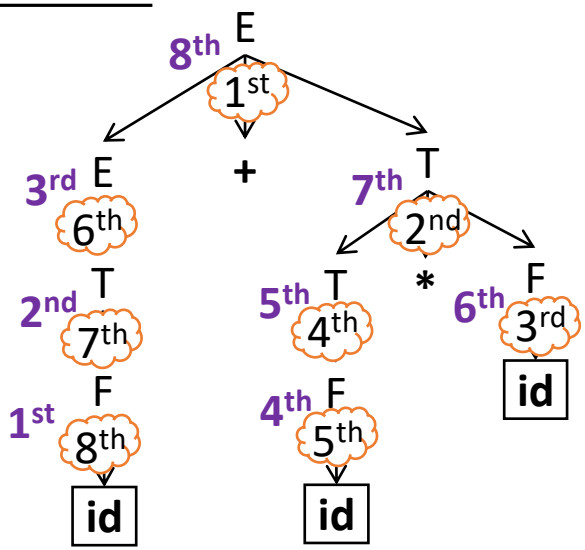
- Always move up branch before starting new branch
- Work on the LAST SYMBOL(S) of the derivation PREFIX

Reverse Rightmost derivation

$$
\begin{aligned}
E &\Longrightarrow E + T \\
&\Longrightarrow E + T * F \\
&\Longrightarrow E + T * id \\
&\Longrightarrow E + F * id \\
&\Longrightarrow E + id * id \\
&\Longrightarrow T + id * id \\
&\Longrightarrow F + id * id \\
&\Longrightarrow id + id * id
\end{aligned}
$$

Reverse

Rightmost Derivation



LR(k) Parsers

LR Parsers - Concept

Perform reverse rightmost derivation

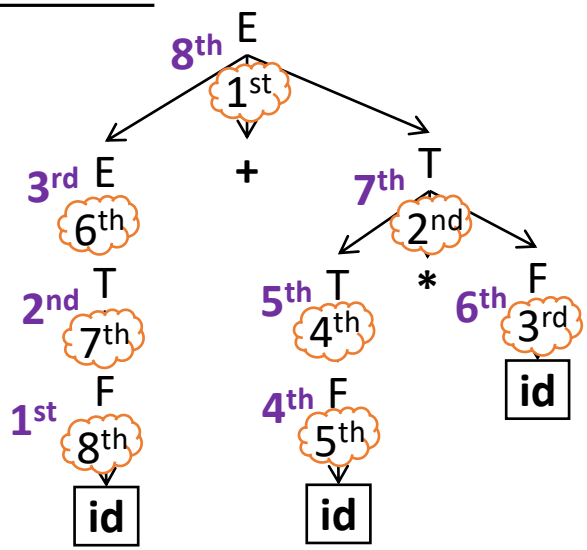
- Always move up branch before starting new branch
- Work on the LAST SYMBOL(S) of the derivation PREFIX

Reverse Rightmost derivation

id | + id * id
F | + id * id
T | + id * id
E + id | * id
E + F | * id
E + T * id |
E + T * F |
E + T | ⇐ E

Reverse

Rightmost Derivation



LL vs LR Parsers

Beyond LL(1)

LL Operations

- *Predict* the rule used to expand nonterminal
- *Verify* that a terminal was expected

LR Operations

- *Shift* to a new branch (add a new leaf)
- *Reduce* a string of symbols to the parent nonterminal

Bottom-Up Parsing – Example

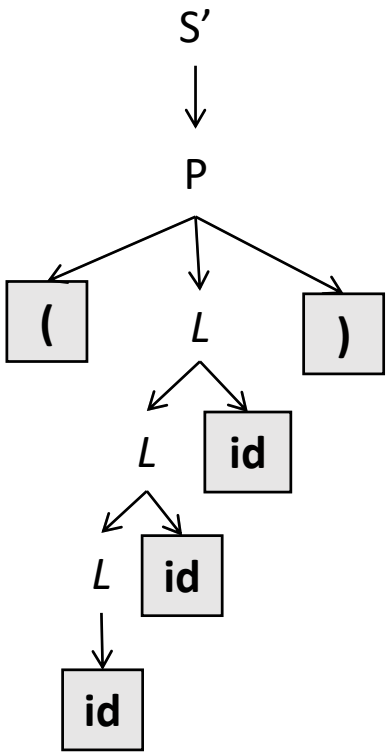
LR Parser Construction

Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❹ *L* ::= *L* id
(*L*	id) eof	shift id
(*L* id	) eof	Red. ❹ *L* ::= *L* id
(*L*	) eof	shift)
(*L*)	eof	Red. ❷ L ::= (*L*)
P	eof	Red. ❶ S' ⟶ P
S'	eof	Accept!!

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

Correct Parse Tree

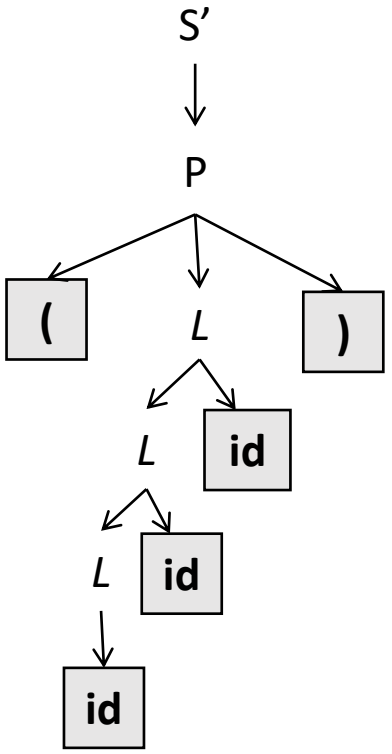


LR Parser – Towards Implementation

LR Parser Construction

Key Points

- Relies on *shift* and *reduce* actions
  - Shift – add a **terminal** leaf to parse tree
  - Reduce – add nonterminal to parse tree
- Builds the parse tree *bottom-up*
  - *Does as many reduce actions as possible before a shift*



Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

Bottom-Up Parsing – Implementation

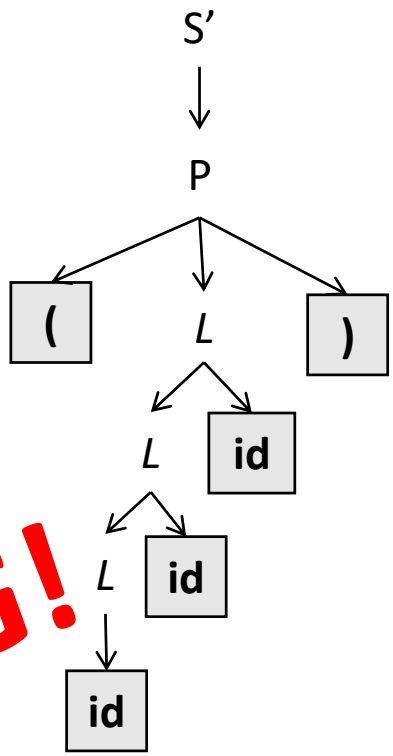
LR Parser Construction

Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❹ *L* ::= *L* id
(*L*	id) eof	shift id
(*L* id	) eof	Red. ❹ *L* ::= *L* id
(*L*	) eof	shift)
(*L*)	eof	Red. ❷ L ::= (*L*)
P	eof	Red. ❶ S' ⟶ P
S'	eof	Accept!!

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

Correct Parse Tree



So… works like an LL parser, right?

WRONG!

LL vs LR Parser – We are Not the Same

LR Parser Construction

LL(1) - Uses 1 token of lookahead and 1 nonterminal to pick production

"Oh, I'll just use that same info for the LR(1) parser"

LR Parser – Towards Implementation

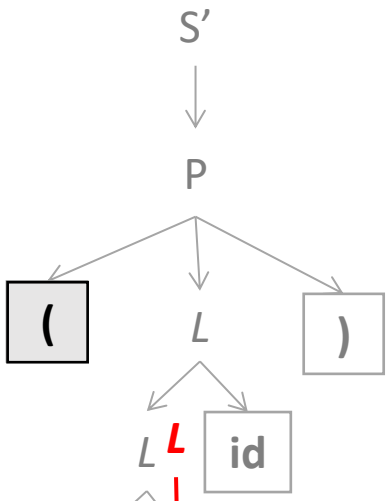
My *darkest timeline* fanfic:
Context: 1 grammar symbols x 1 lookahead

Prefix	Suffix	Action
	(id id id) eof	
(	id id id) eof	shift (
(id	id id) eof	shift id
(*L*	id id) eof	Red. ❸ *L* ::= id
(*L* id	id) eof	shift id
(*L* *L*	id) eof	Red. ❸ *L* ::= *L* id

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

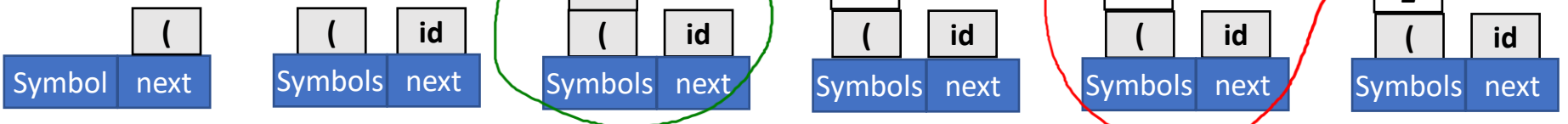
Correct Parse Tree



This prefix will never reduce to the root

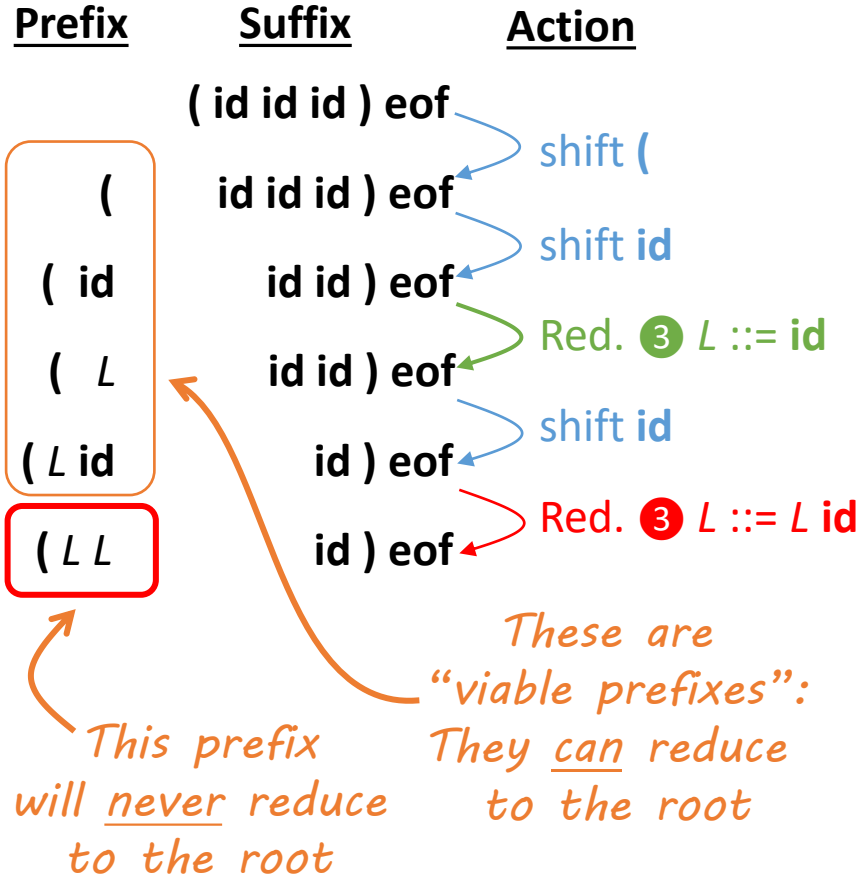
These are "viable prefixes": They can reduce to the root

Naïve / bad idea:



LR Parser – Towards Implementation

LR Parser Construction



The parser must maintain a viable prefix

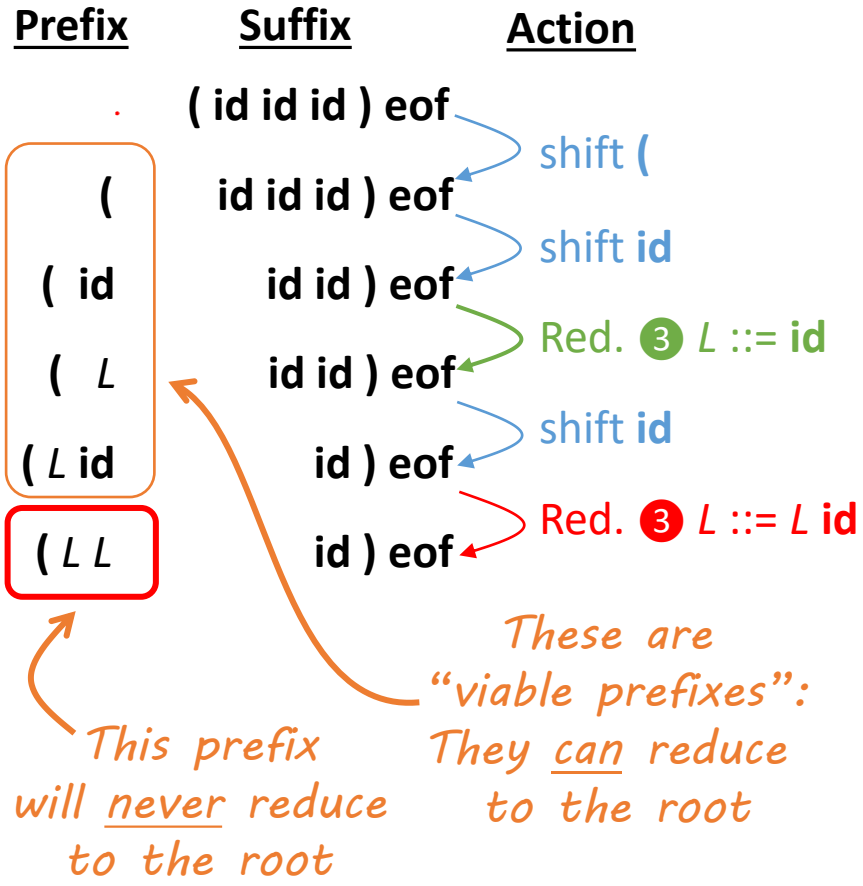
- Ensures no misstep taken
- How to capture the (infinite) viable prefixes?

AMAZING Fact:

- For a deterministic CFG the viable prefixes form a regular language
  - So there's a DFA for it!

LR Parser – Towards Implementation

LR Parser Construction



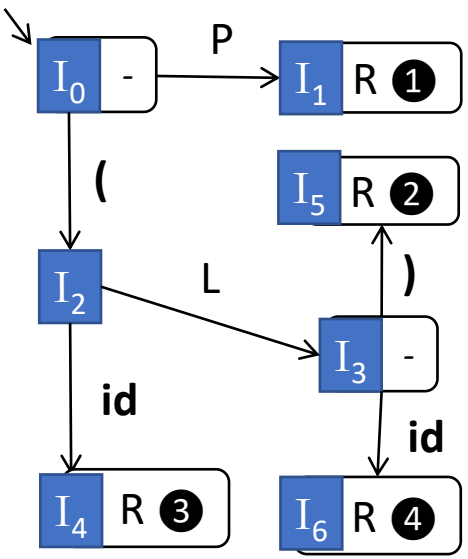
AMAZING Fact:

- For a deterministic CFG the viable prefixes form a regular language
  - So there's a DFA for it!

Grammar G

1. S' ::= P
2. P ::= (L)
3. L ::= id
4. L ::= L id

G Parser Automaton



LR Parser – Towards Implementation

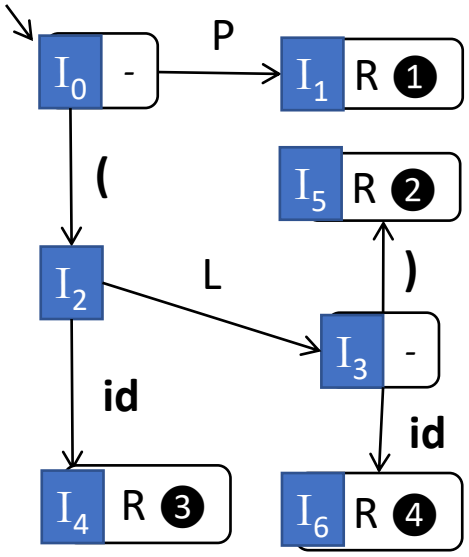
LR Parser Construction

Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❸ *L* ::= *L* id
(*L L*	id) eof	

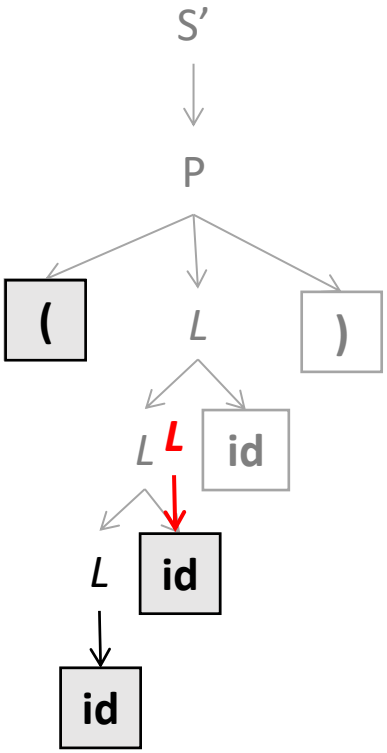
Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

G Parser Automaton

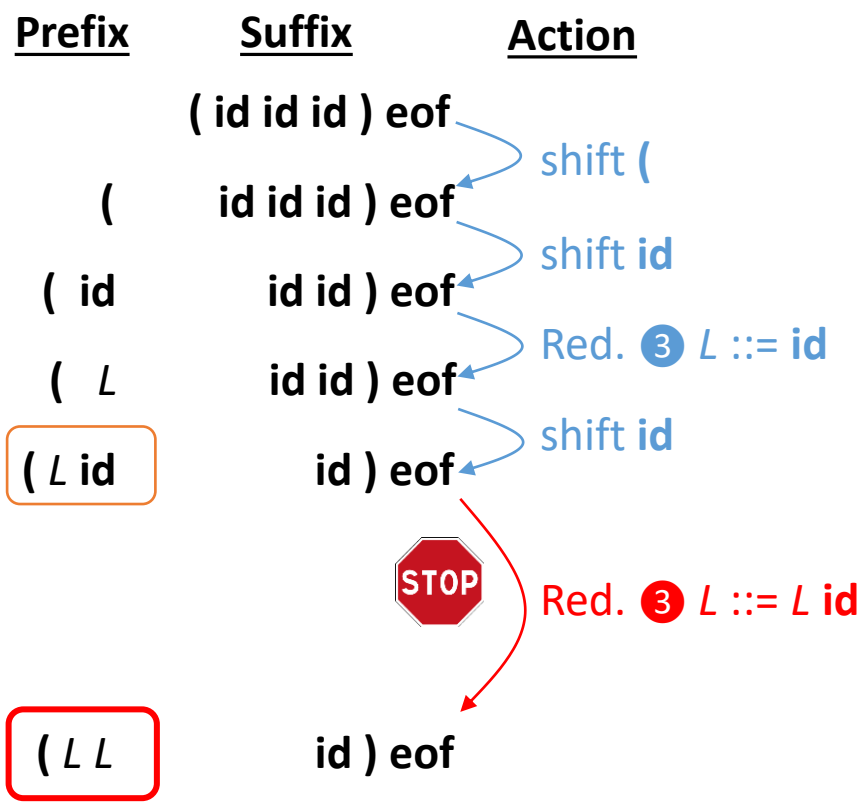


Correct Parse Tree



LR Parser – Towards Implementation

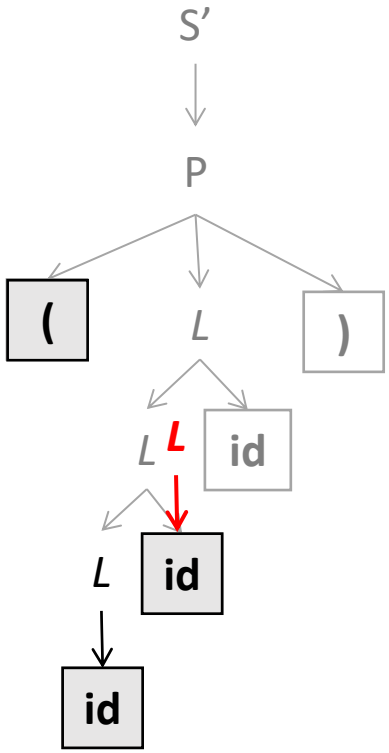
LR Parser Construction



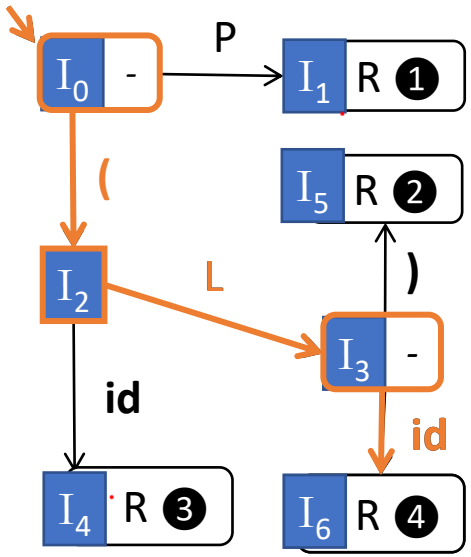
Grammar G

1. S' ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

Correct Parse Tree



G Parser Automaton



LR Parser – Towards Implementation

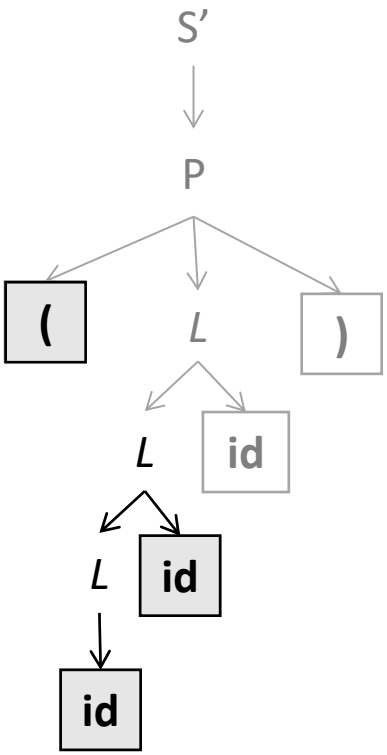
LR Parser Construction

Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❹ *L* ::= *L* id
(*L*	id) eof	

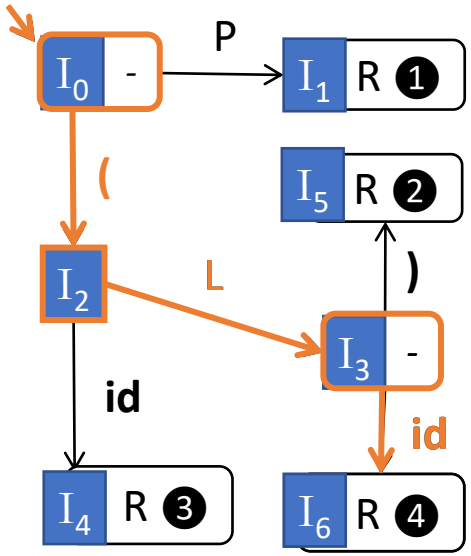
Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

Correct Parse Tree



G Parser Automaton



Viable Prefix Summary

LR Parser Construction

Any DCFG has a regular language of viable prefixes

- We can use the DFA for that language to prevent missteps

Advancing the automaton

- Terminal Edges – add a terminal to the prefix
- Nonterminal Edges – add nonterminal to the prefix
- Accepting states – modify the existing prefix

Key Idea: Run the DFA for viability

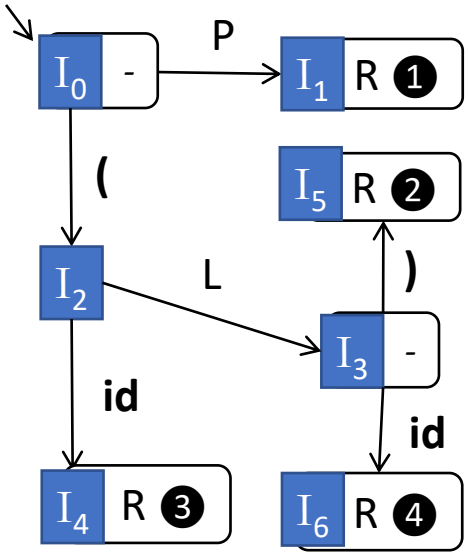
LR Parser Construction

Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❹ *L* ::= *L* id
(*L*	id) eof	shift id
(*L* id	) eof	Red. ❹ *L* ::= *L* id
(*L*	) eof	shift)
(*L*)	eof	Red. ❷ L ::= (*L*)
P	eof	Red. ❶ S' ⟶ P
S'	eof	

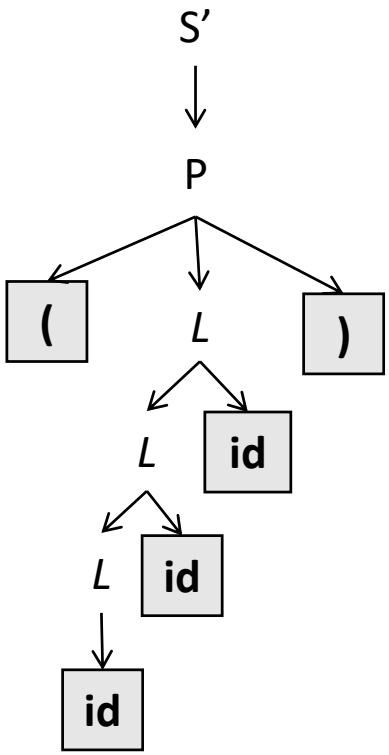
Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= id
4. *L* ::= *L* id

G Parser Automaton



Correct Parse Tree



Shortcut: Don't Restart DFA Every Time

LR Parser Construction

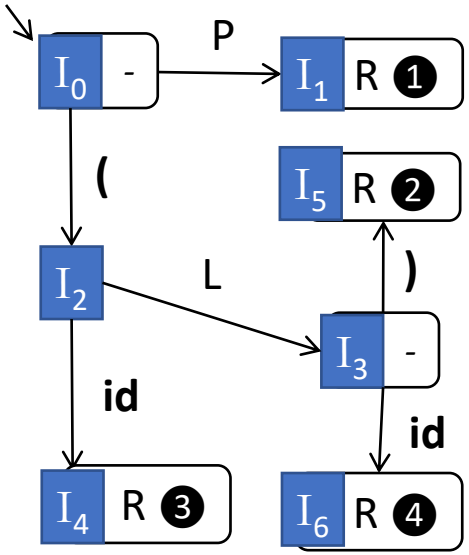
Prefix	Suffix	Action
	(id id id) eof	shift (
(	id id id) eof	shift id
(id	id id) eof	Red. ❸ *L* ::= id
(*L*	id id) eof	shift id
(*L* id	id) eof	Red. ❹ *L* ::= *L* id
(*L*	id) eof	shift id
(*L* id	) eof	Red. ❹ *L* ::= *L* id
(*L*	) eof	shift)
(*L*)	eof	Red. ❷ L ::= (*L*)
P	eof	Red. ❶ S' ⟶ P
S'	eof	

Only this stuff changes

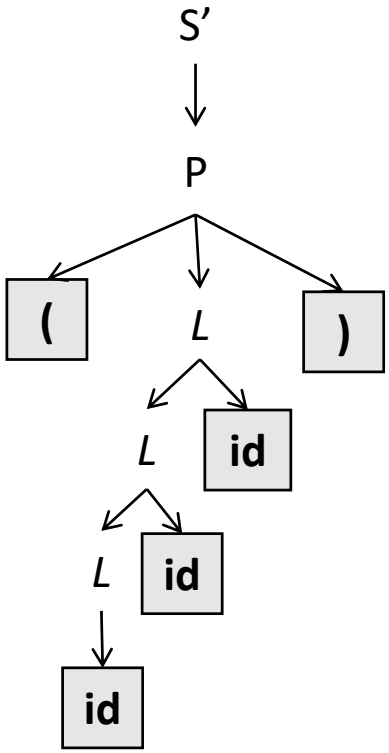
Grammar G

❶ *S'* ::= *P*
❷ *P* ::= (*L*)
❸ *L* ::= id
❹ *L* ::= *L* id

G Parser Automaton



Correct Parse Tree



LR Parser Stack

LR Parser Construction

Amazing fact #2: Can keep position in the parse tree *and* position in the automaton in a single stack

- The prefix becomes implicit in the stack
- Can go back to tracking a lookahead "next" token just beyond the prefix

Advancing the automaton

- Terminal Edges – move terminal ~~symbol into prefix~~ Item onto stack
- Nonterminal Edges – add nonterminal ~~symbol to prefix~~ Item onto stack
- Accepting states – ~~modify the existing prefix~~ Pop RHS stack items

LR Parser Stack

LR Parser Construction

Input

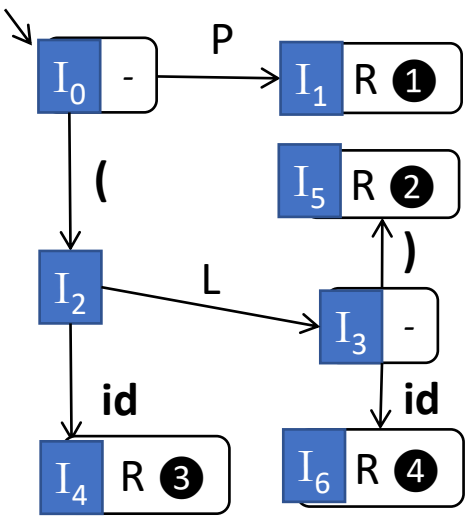
(id id) eof

T_1 T_2 T_3 T_4 T_5

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

G Parser Automaton

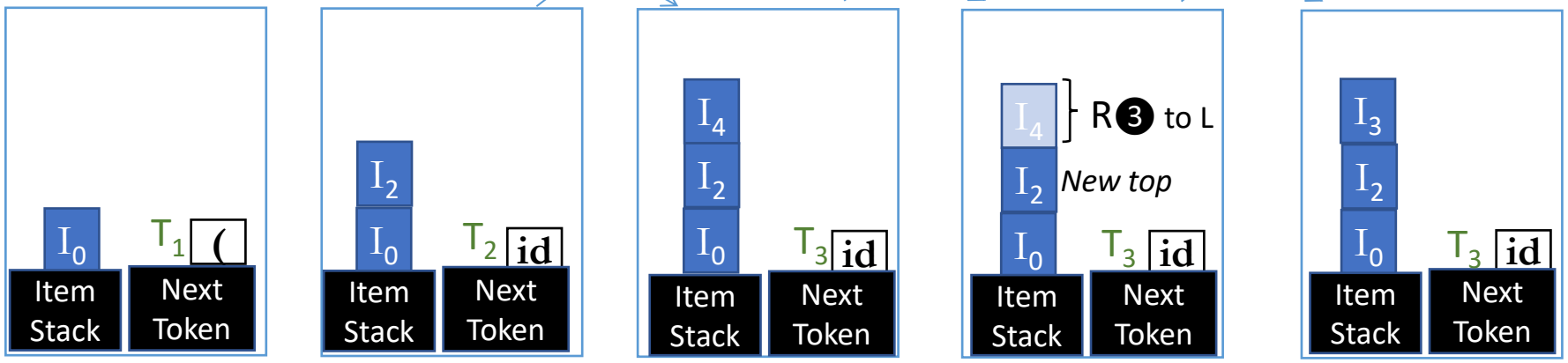


Terminal edge I_0,($\rightarrow I_2$

Terminal edge I_2,id $\rightarrow I_4$

Accepting state
Pop 1 stack item

Nonterminal edge I_2,L$\rightarrow I_3$



LR Parser Stack

LR Parser Construction

Input

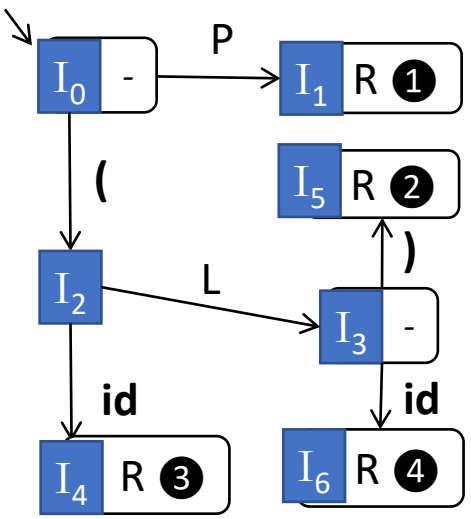
(id id) eof

T_1 T_2 T_3 T_4 T_5

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

G Parser Automaton

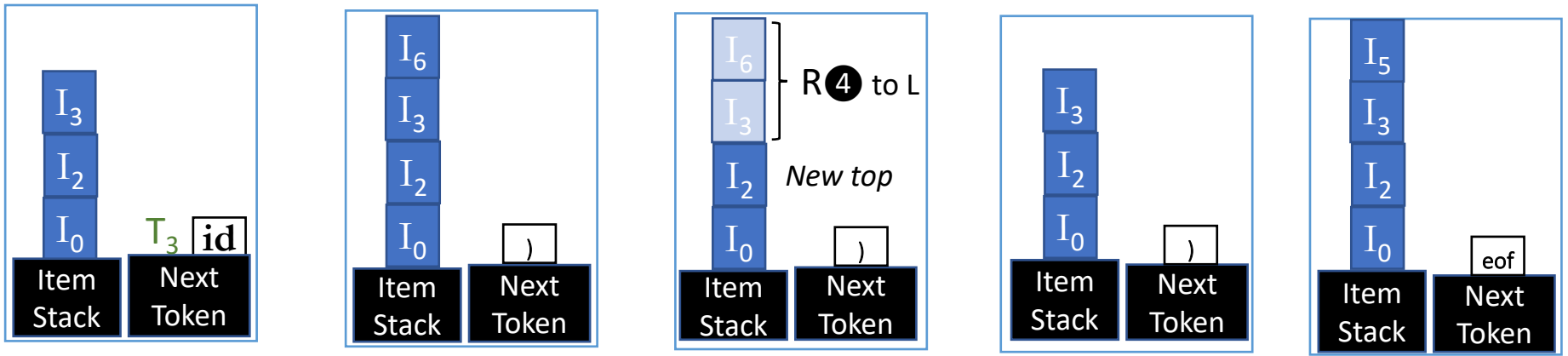


Terminal edge I_3,**id**→I_6

Accepting state 4 Pop 2 items

Terminal edge I_2,L →I_3

Terminal edge I_3,) →I_5



LR Parser Stack

LR Parser Construction

Input

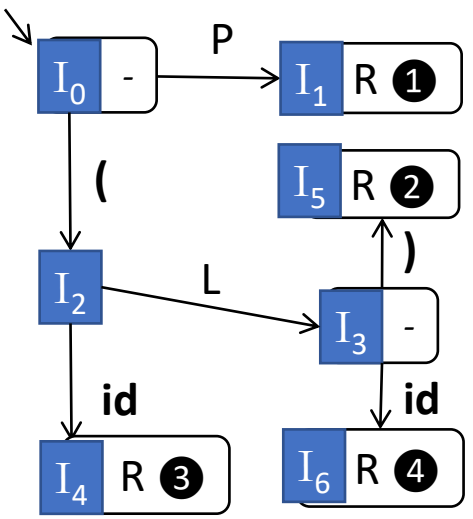
(id id) eof

T_1 T_2 T_3 T_4 T_5

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

G Parser Automaton

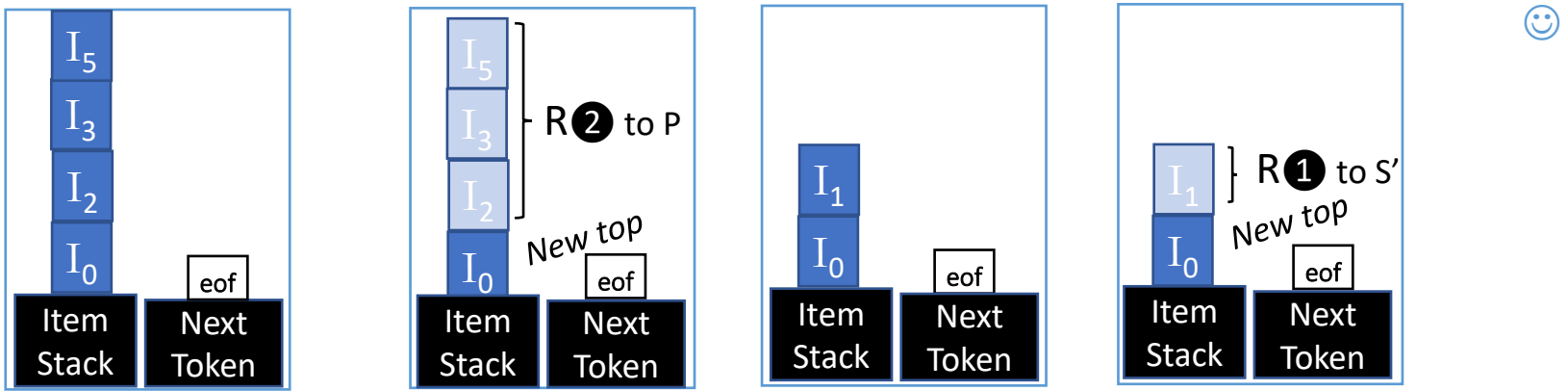


Accept state 2
Pop 3 items

Nonterminal edge
I_3,) →I_5

Accept state 1
Pop 1 item

I_0,S' → Accept!



Next: Building the Parser

LR Parser Construction

Final Tasks to complete Implementation:

- Building the “Prefix” Automaton
- Translating the Automaton into a table