Check-In 10

Review – FOLLOW Sets

Compute FIRST and FOLLOW sets for the below grammar

```
S ::= lpar X rpar

X ::= id comma X

    |  ε
```

Housekeeping

Administrivia

Projects

- P2 not accepted after midnight tonight
- P3 comes out at midnight tonight

Quizzes

- Planning to grade Q1 this weekend

University of Kansas| Drew Davidson

EECS 665

COMPILER CONSTRUCTION

LL(1) SDT

Last Lecture

Lecture Review – FOLLOW Sets

Building LL(1) Parsers

- Finished FIRST Sets
- FOLLOW Sets

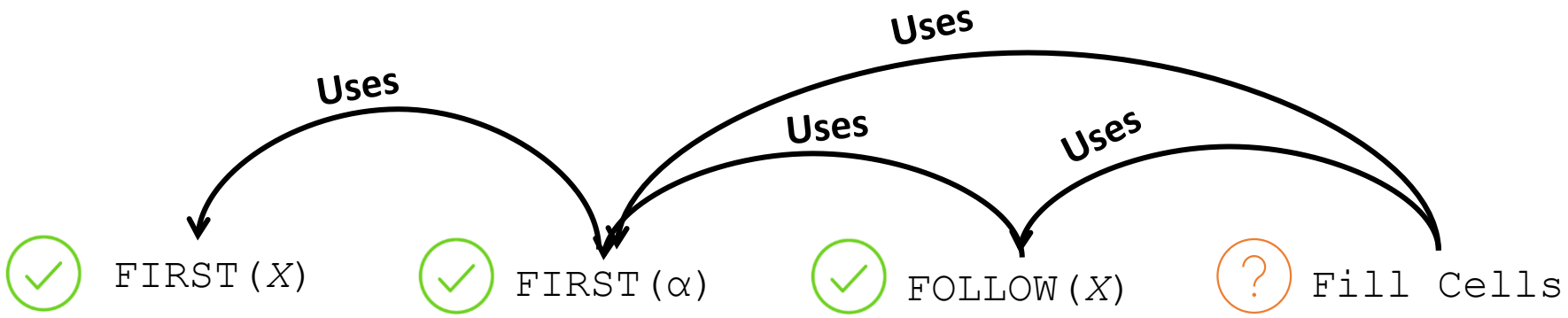
You should know

- How to build an LL(1) Selector table
  - Compute FIRST sets of a grammar
  - Computer FOLLOW sets of a grammar

Parsing

Building LL(1) Parser

Review LL(1) Concepts



Step 1.

Build FIRST sets for single symbols of the grammar

Step 2.

Build FIRST sets for RHS symbols

Step 3.

Build FOLLOW sets for Nonterminals

Step 4.

Place productions

Building Selector Table Example

CFG

P1 *S* ::= **lpar** *X* **rpar**

P2 *X* ::= **id comma** X

P3 | ε

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

FIRST(X) = { **id**, ε } = { **lpar** }

FIRST(**lpar** *X* **rpar**)

FIRST(**id comma** *X*) = { **id** }

FIRST(ε) = { ε }

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

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

	lpar	**rpar**	**id**	**comma**	**eof**
S	P1 **lpar** X **rpar**				
X		P3 ε	P2 **id comma** X		

For each production X ::= α

P1 S ::= **lpar** *X* **rpar**

For terminals in FIRST(α) = {**lpar**}

Put P1 @ Table[*S*][**lpar**]

P2 X ::= **id comma** *X*

For terminals in FIRST(α) = { **id** }

Put P2 @ Table[*X*][**id**]

P3 X ::= ε

No terminals in FIRST(α)

ε ***is*** in FIRST(α)

For **t** in FOLLOW(*X*) = {**rpar**}

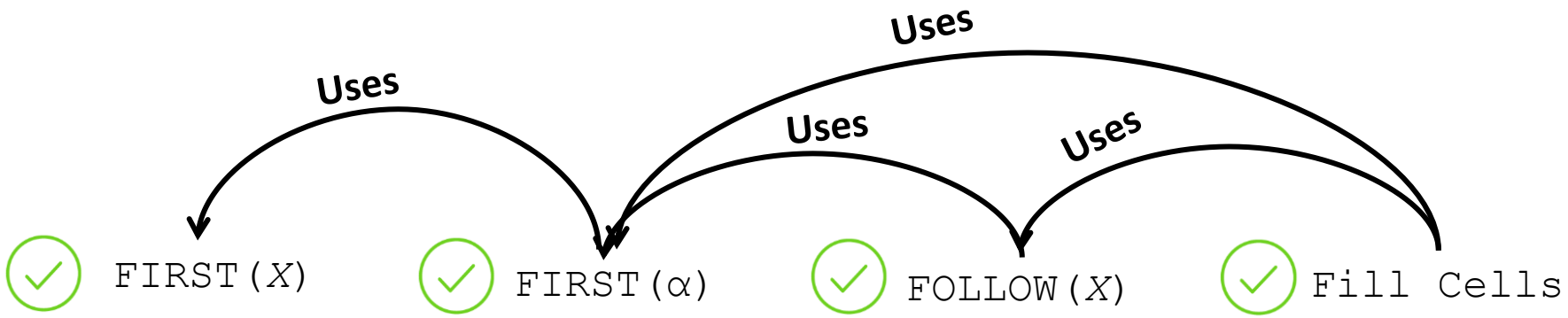
Put P3 @ Table[*X*][**rpar**]

Selector Table Building Procedure

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

Selector Table Dependencies

Building the Selector Table



Step 1.

Build FIRST sets for single symbols of the gramm

Step 2

Step 3.

ild FOLLOW sets onterminals

Step 4.

Place productions

But what does this do for a compiler?

This LL(1) parser only recognizes valid strings

Lecture Outline

Lecture – SDT

Beyond Parse Tree Recognition

- Connect LL(1) parser to compiler needs (SDT)

Coding up Syntax-Directed Translation

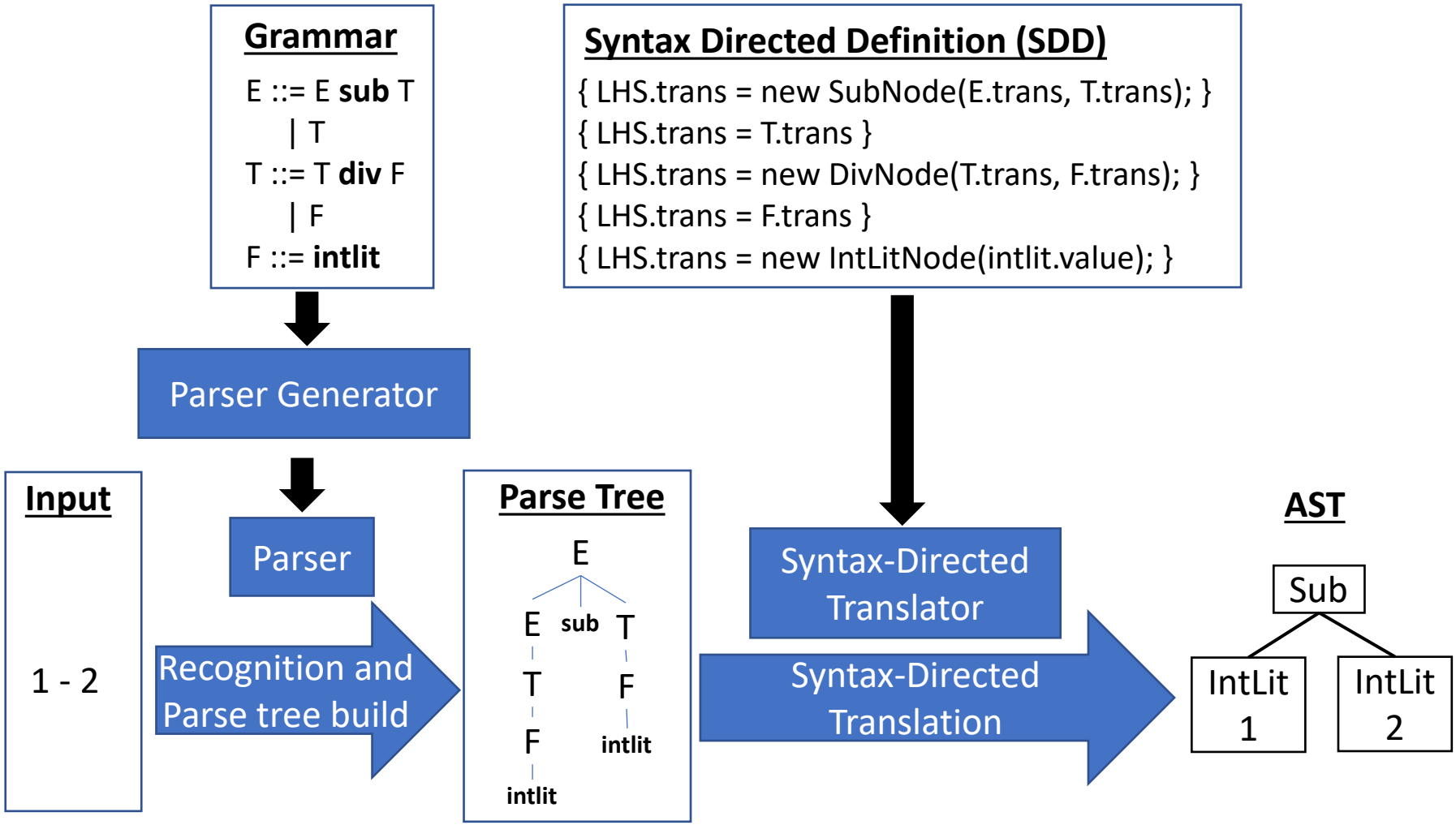
- Parser Tweaks
- Examples

Parsing

A Quick SDT Recap

Syntax-Directed Translation

Conceptual/Logical workflow:

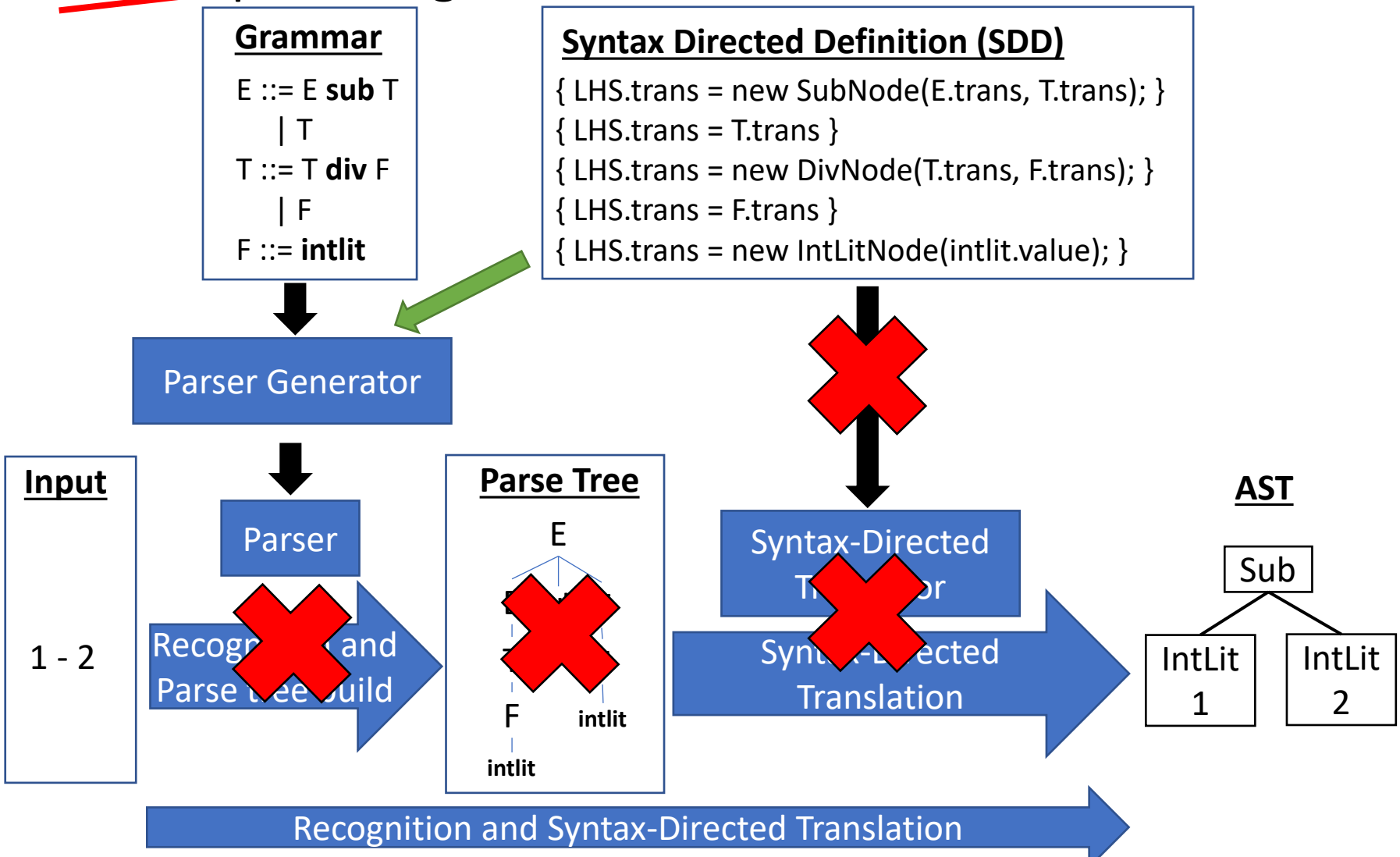


A Quick SDT Recap

Syntax-Directed Translation

Implemented

Conceptual/Logical workflow:



A Quick SDT Recap

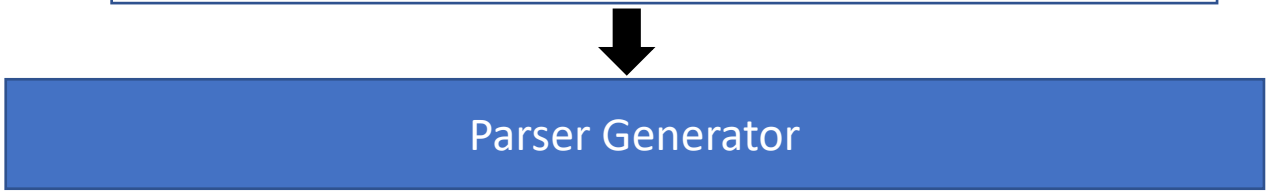
Syntax-Directed Translation

Implemented

Conceptual/Logical workflow:

Attribute Grammar

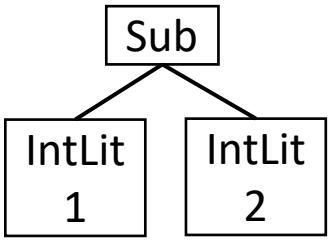
```
E ::= E sub T     { LHS.trans = new SubNode(E.trans, T.trans); }
    | T           { LHS.trans = T.trans }
T ::= T div F     { LHS.trans = new DivNode(T.trans, F.trans); }
    | F           { LHS.trans = F.trans }
F ::= intlit      { LHS.trans = new IntLitNode(intlit.value); }
```



Input

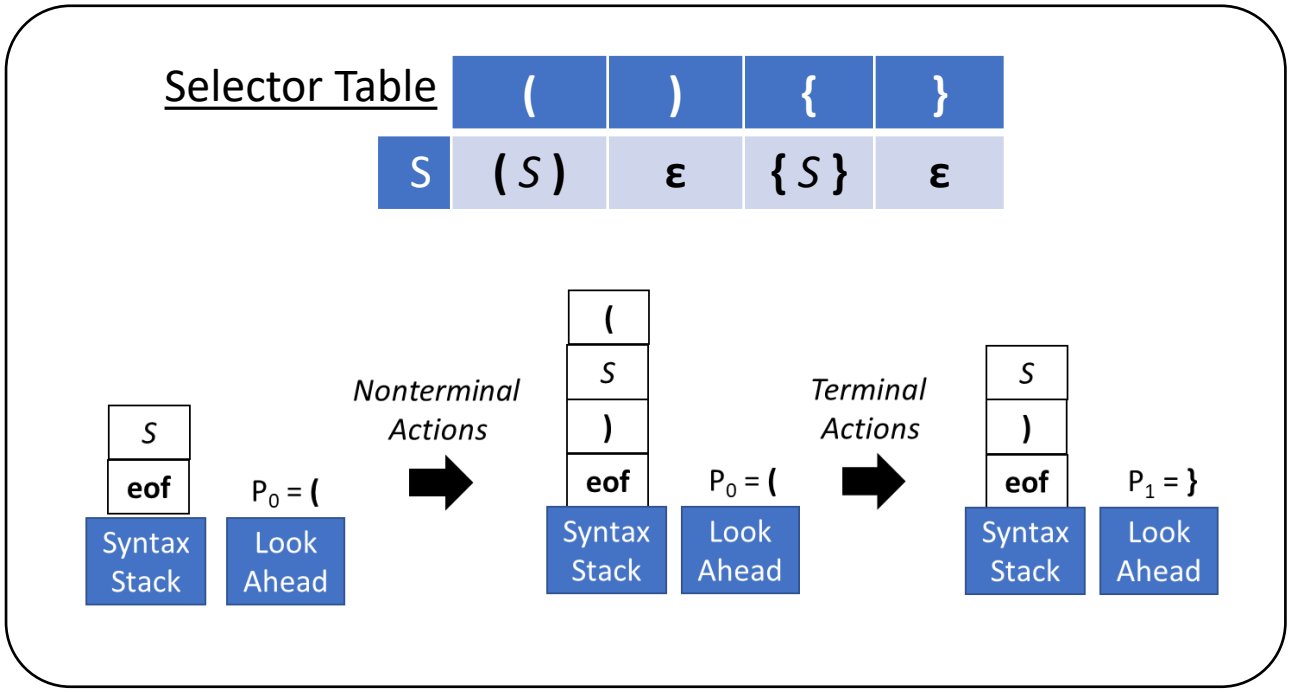
1 - 2

AST



Wait *This* Thing can do SDT?

Top-Down Syntax-Directed Translation



Pretty much! We just need to make a couple little tweaks

Lecture Outline

Lecture 10 – SDT

Finish Building LL(1) Parsers

- Filling the Selector Table

Top-Down Syntax-Directed Translation

- Parser Tweaks
- Grammar Tweaks

Parsing

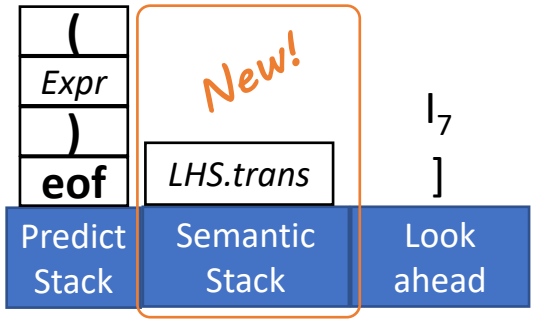
Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

How do we add SDT capability?

1. Add 2nd semantic stack to hold nonterminal translations
2. SDT rules become stack actions
   - Pop translations of RHS nonterms
   - Push computed translation of LHS nonterm

CFG	**SDT Rules**	**SDT Actions**
Expr ::= ε	LHS.trans = 0	push 0
| **(** *Expr* **)**	LHS.trans = Expr.trans + 1	ExprTrans = pop; push ExprTrans + 1
| **[** *Expr* **]**	LHS.trans = Expr.trans	ExprTrans = pop; push ExprTrans



Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

How do we add SDT capability?

3. Number the actions, embed numbers into production RHS

CFG

Expr ::= ε #1

| **(** *Expr* **)** #2

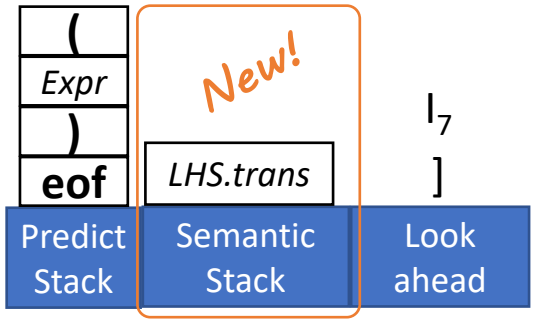
| **[** *Expr* **]** #3

SDT Actions

#1 push 0

#2 ExprTrans = pop; push ExprTrans + 1

#3 ExprTrans = pop; push ExprTrans



Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

How is SDT performed?

1. When parsing, put numbers on stack with symbols

CFG

Expr ::= ε #1

| **(** *Expr* **)** #2

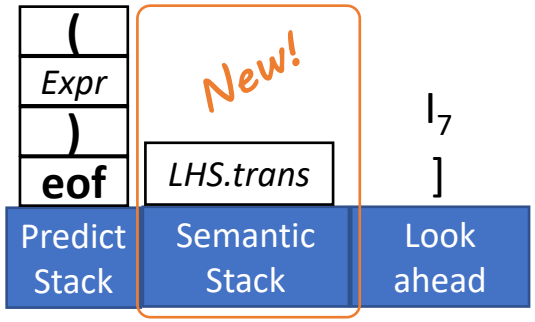
| **[** *Expr* **]** #3

SDT Actions

#1 push 0

#2 ExprTrans = pop; push ExprTrans + 1

#3 ExprTrans = pop; push ExprTrans



Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

How is SDT performed?

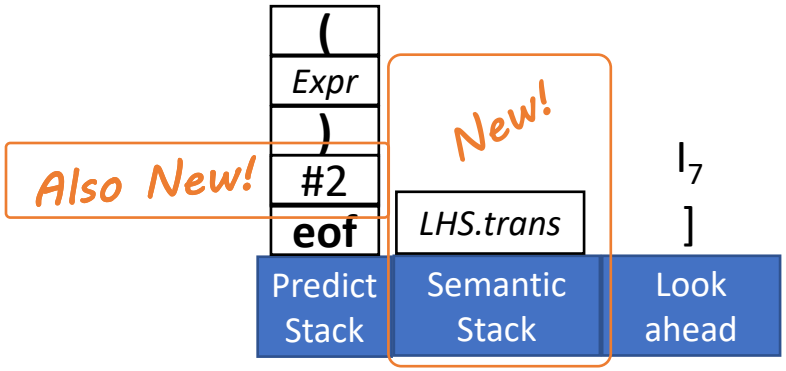
1. When parsing, put numbers on stack with symbols
2. When we encounter a number, run the corresponding code
   - By parse's end, semantic stack will contain the root translation

CFG

Expr ::= ε #1
| **(** *Expr* **)** #2
| **[** *Expr* **]** #3

SDT Actions

#1 push 0
#2 ExprTrans = pop; push ExprTrans + 1
#3 ExprTrans = pop; push ExprTrans



Action Numbers, Example 1

SDT for Top-Down Parsing

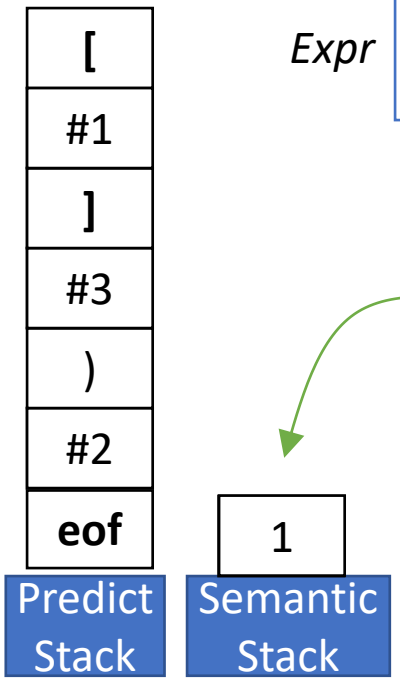
CFG

Expr ::= ε #1
| **(** *Expr* **)** #2
| **[** *Expr* **]** #3

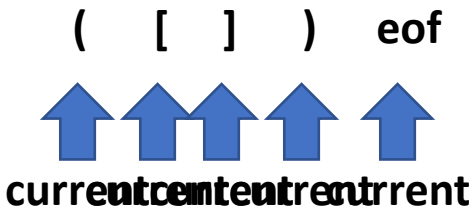
SDT Actions: Counting Max Parens Depth

#1 push 0
#2 ExprTrans = pop; push(ExprTrans + 1)
#3 ExprTrans = pop; push(ExprTrans)

	(	**[**	**]**	**)**	**EOF**
Expr	**(** *Expr* **)** #2	**[** *Expr* **]** #3	ε #1	ε #1	ε #1



Root translation



No-op SDT Actions

SDT for Top-Down Parsing

CFG

Expr ⟶ ε #1

| **(** *Expr* **)** #2

| **[** *Expr* **]** #3

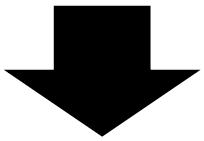
SDT Actions: Counting Max Parens Depth

#1 push 0

#2 ExprTrans = pop; push(ExprTrans + 1)

#3 ExprTrans = pop; push(ExprTrans)

Useless rule



CFG

Expr ⟶ ε #1

| **(** *Expr* **)** #2

| **[** *Expr* **]**

SDT Actions: Counting Max Parens Depth

#1 push 0

#2 ExprTrans = pop; push(ExprTrans + 1)

Lecture Outline

Lecture 10 – SDT

Finish Building LL(1) Parsers

- Filling the Selector Table

Top-Down Syntax-Directed Translation

- Parser Tweaks
- Grammar Tweaks

Parsing

SDD Rules to (Semantic) Stack Actions

SDT for Top-Down Parsing

Template

CFG Production

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

SDT Rule

LHS.trans = combined α_1.trans ... α_n.trans

Enriched Production

$X ::= \alpha_1\ \alpha_2 \ldots \alpha_n$ #1

Stack Action #1

t_n = sem.pop()
…
t_1 = sem.pop()
t_L = combine $t_1 \ldots t_n$
sem.push(t_L)

Example

CFG Production

E ::= *L* **div** *R*

SDT Rule

LHS.trans = *L*.trans / *R*.trans

Enriched Production

E ::= *L* **div** *R* #1

Stack Action #1

t_R = sem.pop()
t_L = sem.pop()
sem.push(t_L / t_R)

Why Put LHS action at end of Production?

SDT for Top-Down Parsing

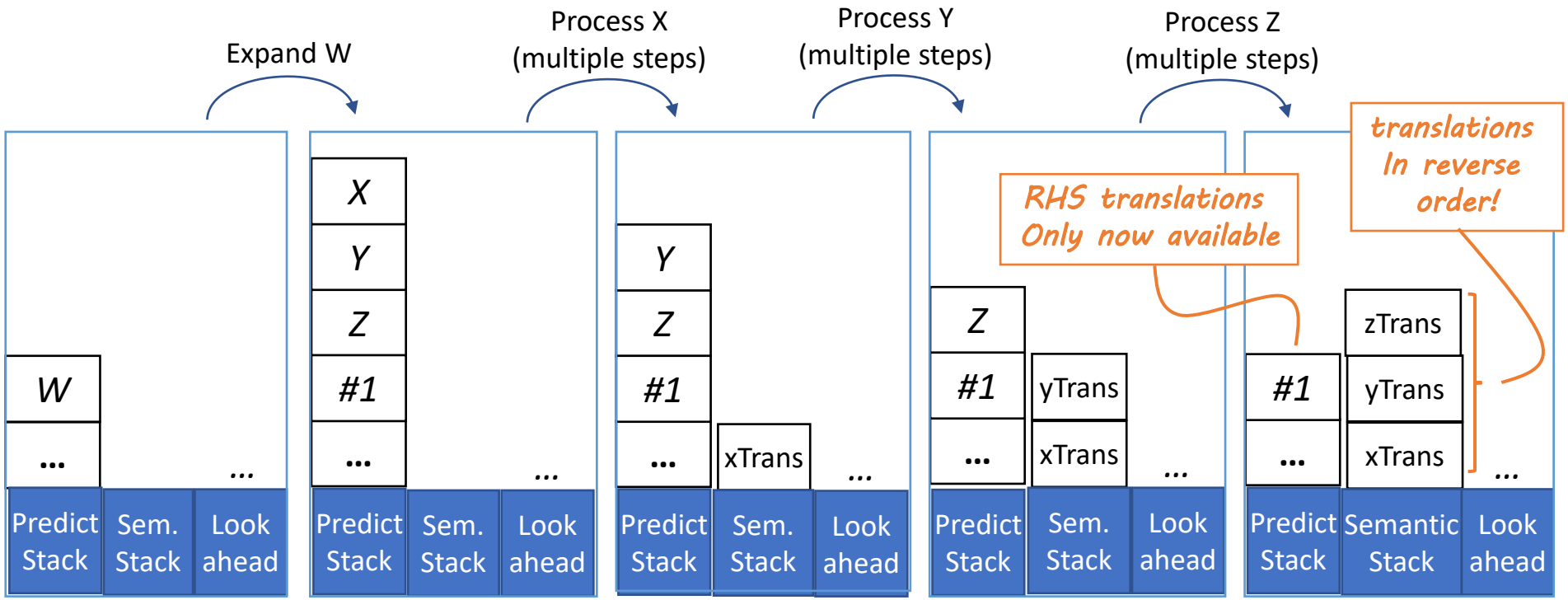
Augmented Production

W ::= *X Y Z* #1

…

Stack Actions

#1 zTrans = sem.pop()
yTrans = sem.pop()
xTrans = sem.pop()
… *combine xTrans, yTrans, zTrans into xTrans* …
sem.push(xTrans)



Embedding Action Numbers

SDT for Top-Down Parsing

Put the action # at the end of the production, unless…

- Rules is a strict pass-through of the translation
  - No action needed (it would have just gotten popped then pushed right back on)

X ::= *Y*

X ::= *Y*

LHS.trans = *Y.trans*

No stack action

- You're using a lexeme value
  - Put the action *before* the terminal (ensures it's available at lookahead position)

X ::= **t**

X ::= #1 **t**

LHS.trans = *terminal.value*

#1

tmpVal = lookahead.value
push tmpVal

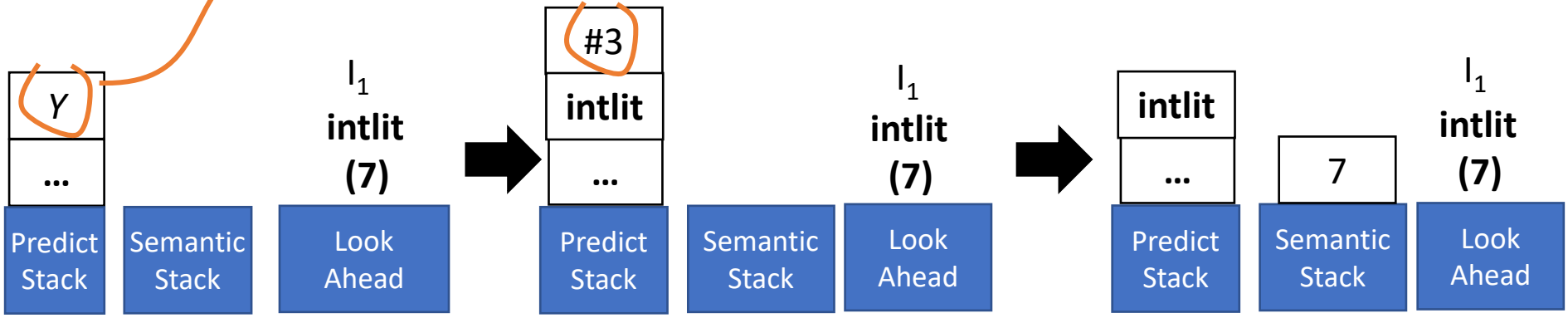
Why Put Terminals Actions before symbol?

SDT for Top-Down Parsing

Augmented CFG Fragment	**Stack Actions Fragment**	**Input Stream**
Y ::= #3 **intlit**	#3 yTrans = lookahead.value sem.push(yTrans)	7 + 8 + 9

Put lookahead value on sem stack

Derive RHS Via selector table (not shown)



Preparing SDT Schemes

SDT for Top-Down Parsing

CFG

Expr ::= *Expr* **minus** *Term*
| *Term*
Term ::= **intlit**

SDD Rules for Evaluation

LHS.trans = *Expr*.trans - *Term*.trans

LHS.trans = *Term*.trans

LHS.trans = **intlit**.value

Augmented CFG

Expr ::= *Expr* **minus** *Term* #1
| *Term*
Term ::= #2 **intlit**

Stack Actions

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

#2 sem.push(lookahead.value)

Preparing SDT Schemes

SDT for Top-Down Parsing

CFG

Expr ::= *Expr* **minus** *Term*
| *Term*
Term ::= **intlit**

SDD Rules for Evaluation

LHS.tr ... *Term*.trans
LHS ...

Hey! This grammar isn't LL(1)

Augmented CFG

Expr ::= *Expr* **minus** *Term* #1
| *Term*
Term ::= #2 **intlit**

...ns

...tTrans = sem.pop();
eTrans = sem.pop();
sem.push(eTrans - tTrans)

#2 sem.push(lookahead.value)

Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

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

T ::= #2 **intlit**

LL(1) CFG

E ::= *T E'*

E' ::= **minus** *T* #1 *E'* | ε

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(**intlit**.value)

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(**intlit**.value)

Time to see the
parser in action!

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**		

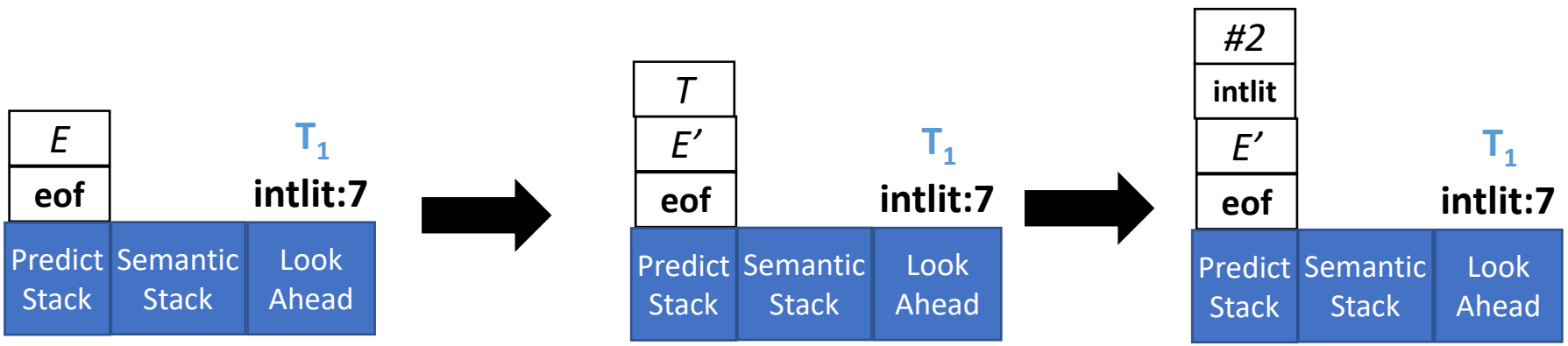
Eval Stack Actions

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

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

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

T_1 T_2 T_3 T_4



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**		

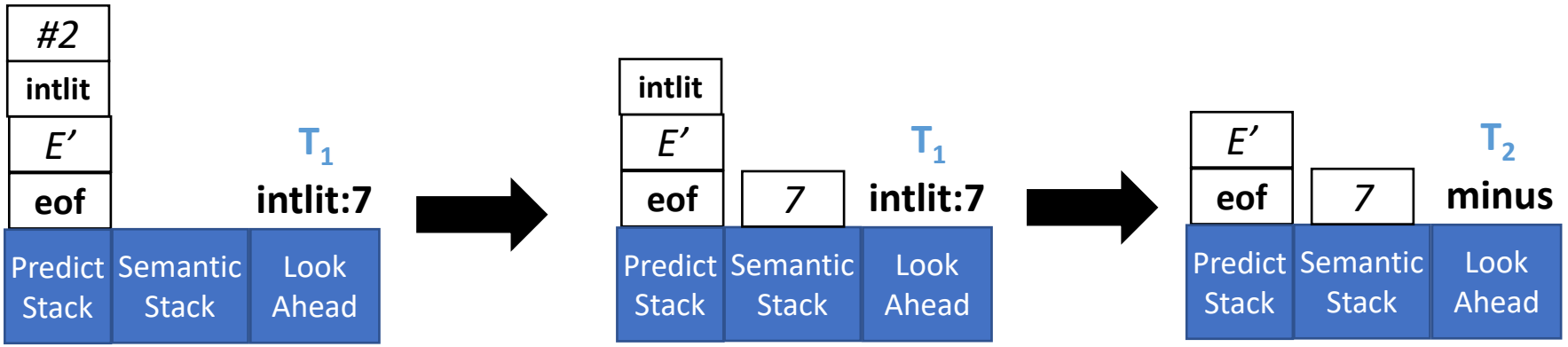
Eval Stack Actions

#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



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**		

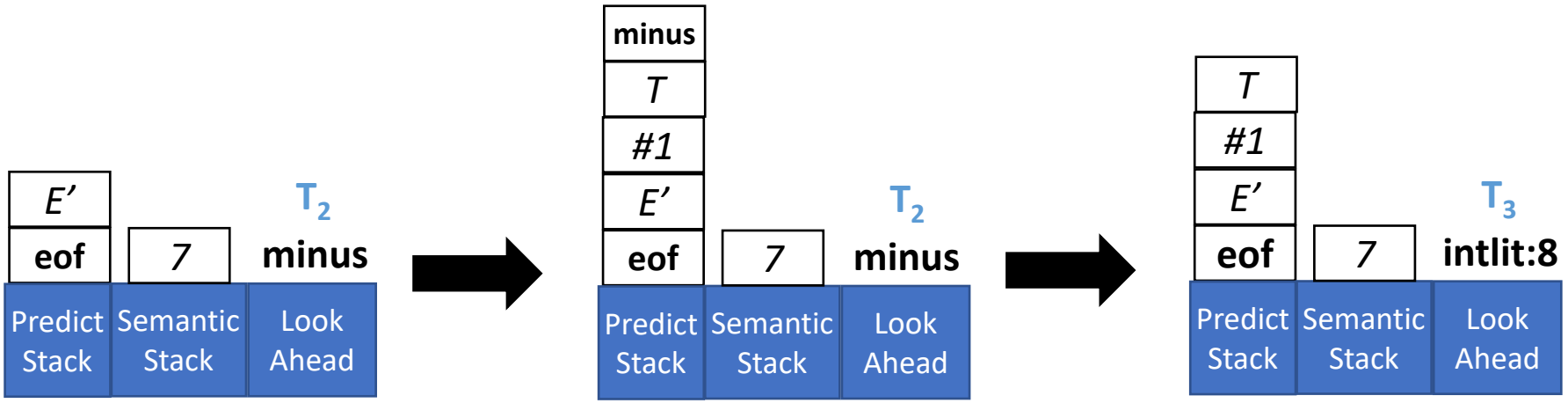
Eval Stack Actions

#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



Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

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

T ::= #2 **intlit**

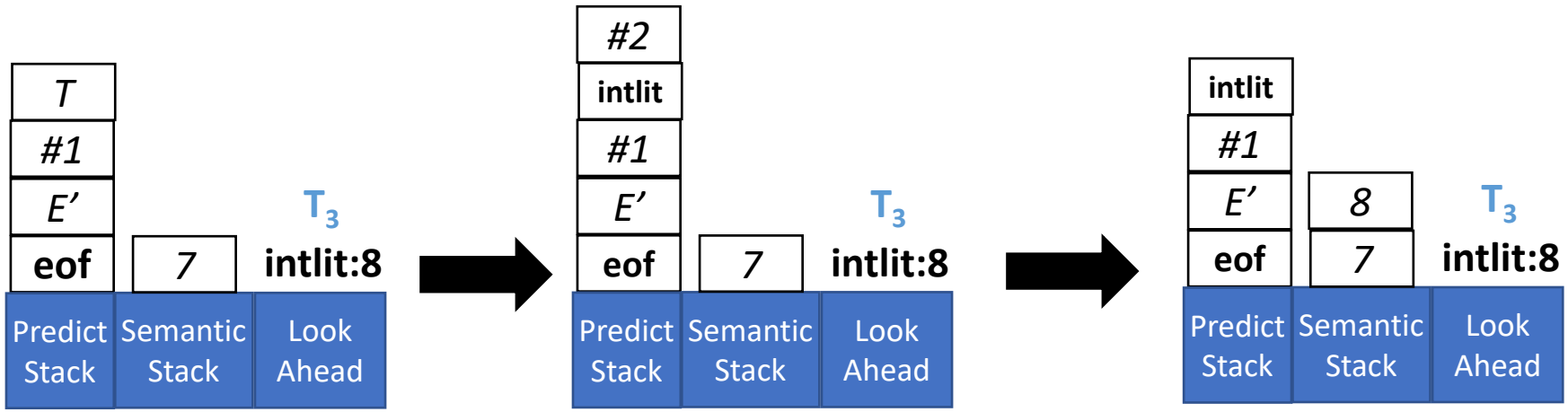
	intlit	**minus**	**EOF**
E	*T E'*		
E'		**minus** *T* #1 *E'*	ε
T	#2 **intlit**		

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



Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

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

T ::= #2 **intlit**

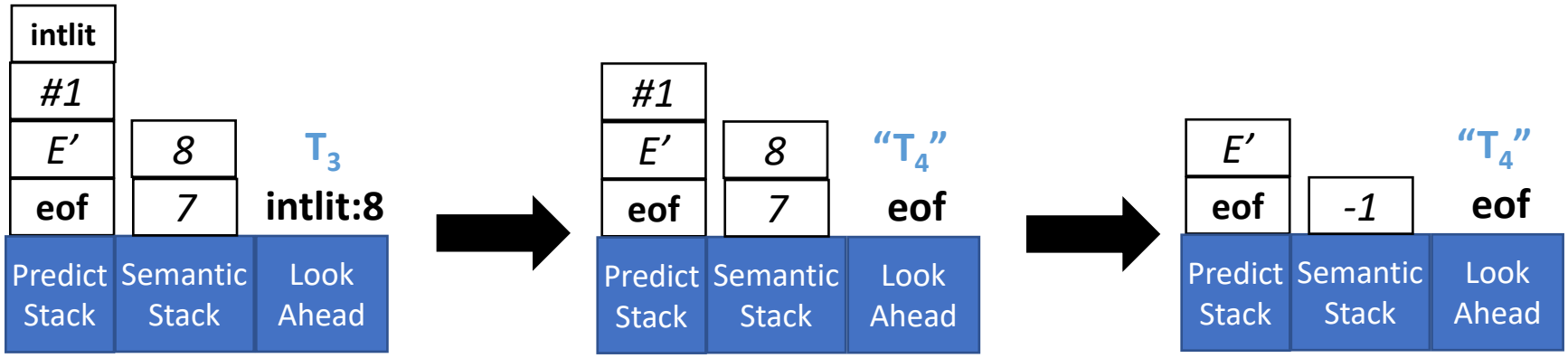
	intlit	**minus**	**EOF**
E	*T E'*		
E'		**minus** *T* #1 *E'*	ε
T	#2 **intlit**		

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



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

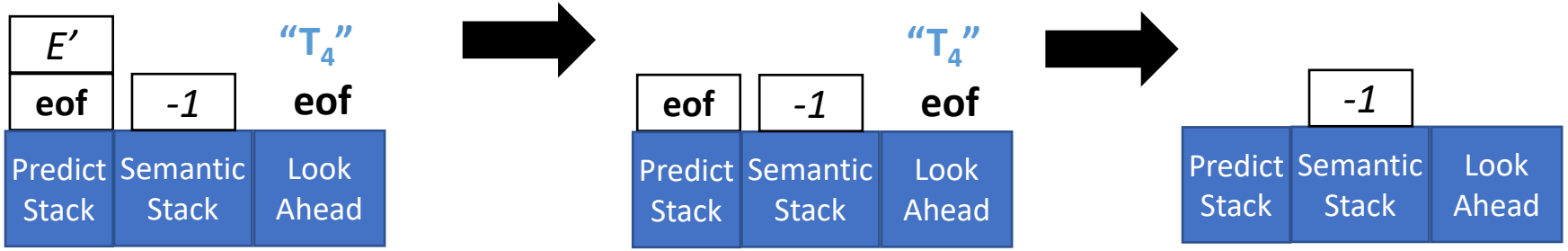
#1 tTrans = sem.pop() ;
eTrans =sem. pop() ;
LTrans = eTrans - tTrans;
sem.push(LTrans)

#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

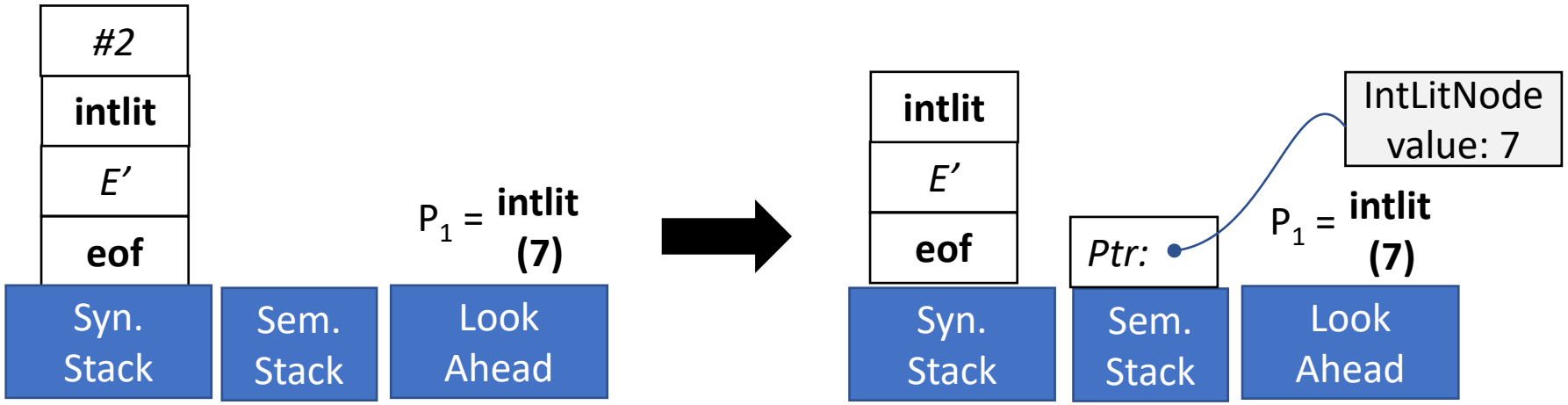
	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

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

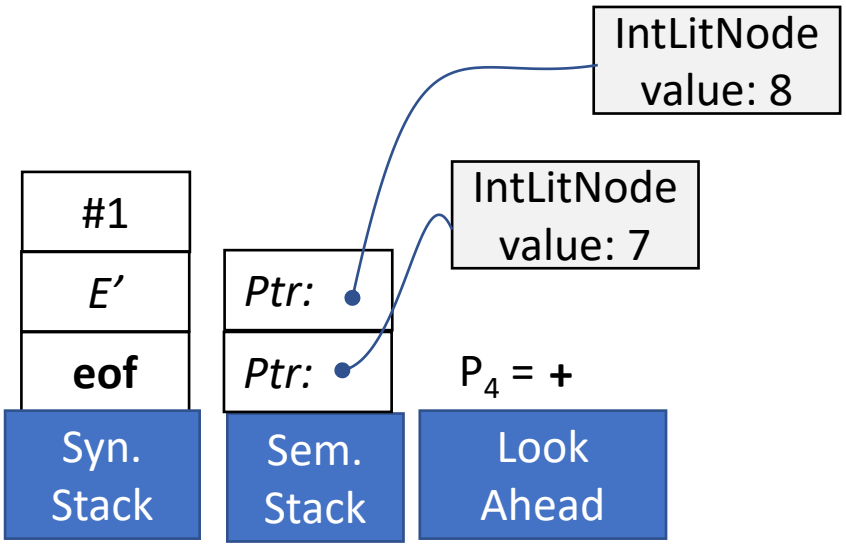
	intlit	**+**	**EOF**
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

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

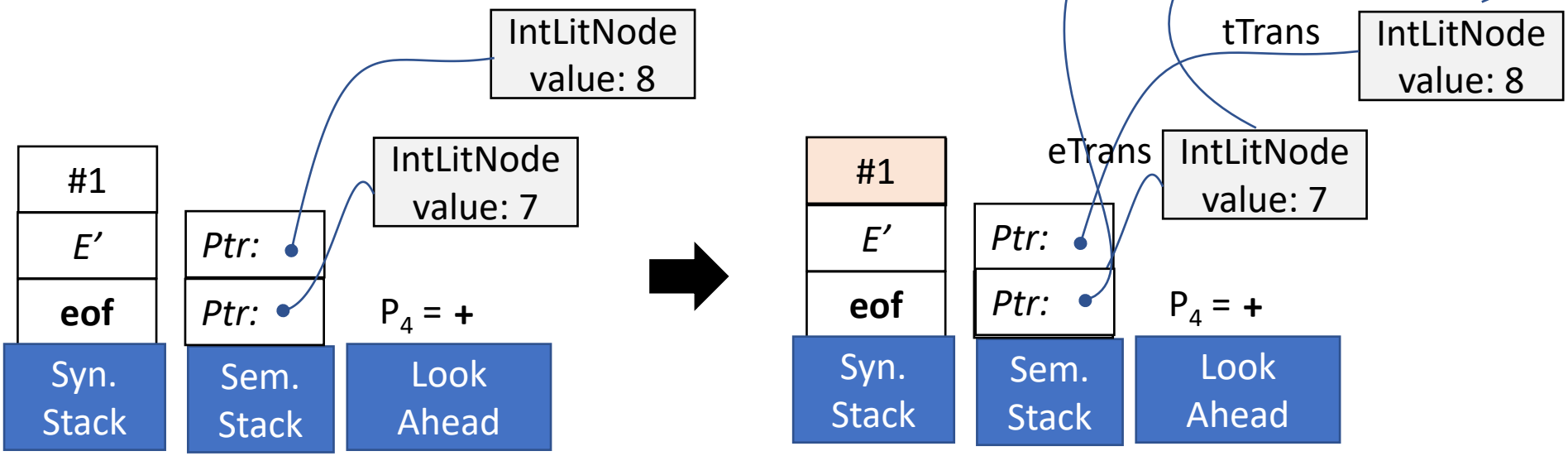
	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

AST Building Stack Actions

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

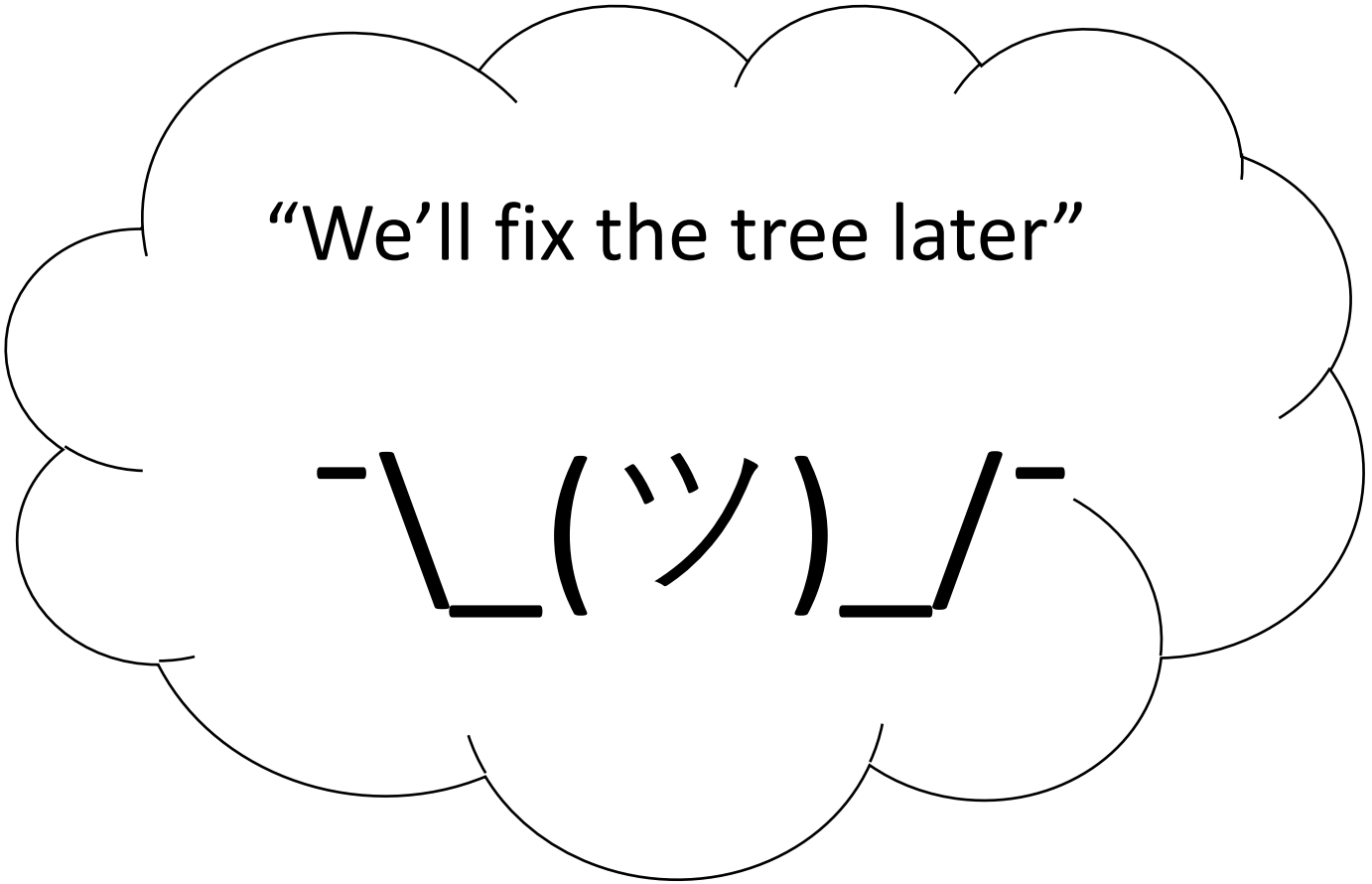
#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

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)