

Check-In 10

Review – FOLLOW Sets

Compute FIRST and FOLLOW sets for the below grammar

$S ::= \text{lpar } X \text{ rpar}$

$X ::= \text{id comma } X$

$\quad \mid \epsilon$

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

ECS 665

COMPILER *CONSTRUCTION*

LL(1) SDT

Last Lecture

Lecture Review – FOLLOW Sets

Building LL(1) Parsers

- Finished FIRST Sets
- FOLLOW Sets

Built the Selector Table

- Filled cells according to algorithm

You should know

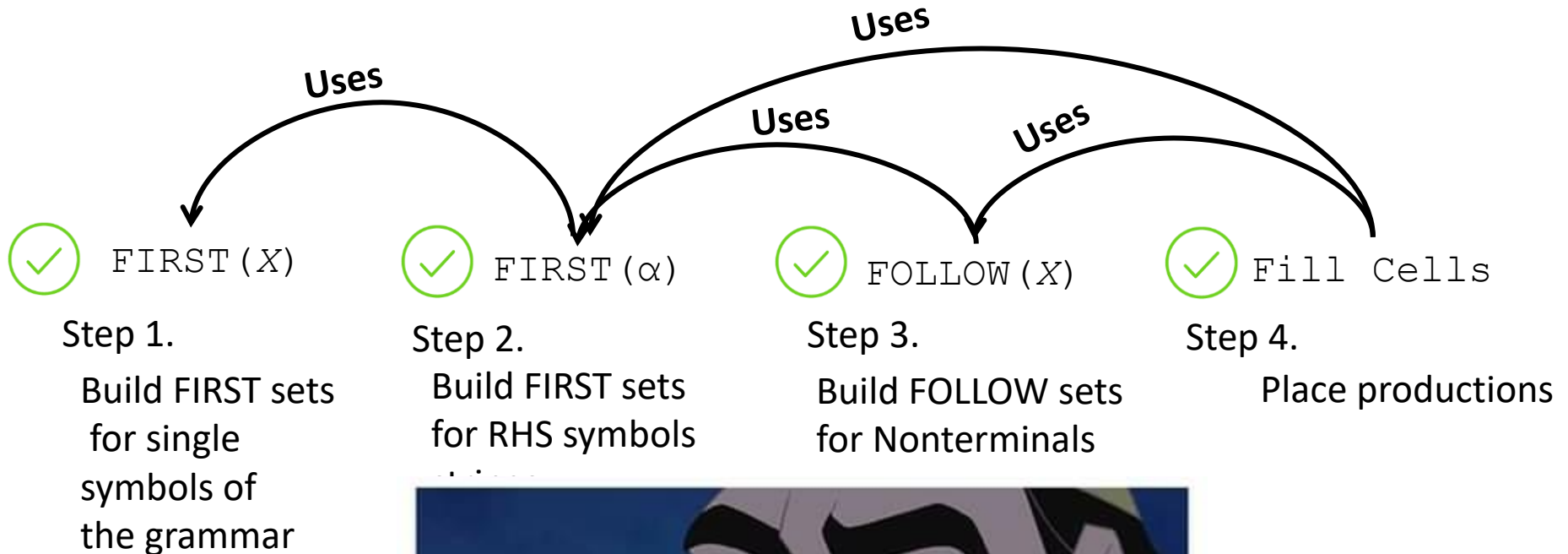
- How to build an LL(1) Selector table
 - Compute FIRST sets of a grammar
 - Compute FOLLOW sets of a grammar



Parsing

Building LL(1) Parser

Review LL(1) Concepts



Building Selector Table Example

CFG

P1 $S ::= \text{lpar } X \text{ rpar}$
P2 $X ::= \text{id comma } X$
P3 $\quad \mid \varepsilon$

$\text{FIRST}(S) = \{\text{lpar}\}$
 $\text{FIRST}(X) = \{\text{id}, \varepsilon\} = \{\text{lpar}\}$
 $\text{FIRST}(\text{lpar } X \text{ rpar}) = \{\text{lpar}\}$
 $\text{FIRST}(\text{id comma } X) = \{\text{id}\}$
 $\text{FIRST}(\varepsilon) = \{\varepsilon\}$
 $\text{FOLLOW}(S) = \{\text{eof}\}$
 $\text{FOLLOW}(X) = \{\text{rpar}\}$

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

For each production $X ::= \alpha$

P1 $S ::= \text{lpar } X \text{ rpar}$

For terminals in $\text{FIRST}(\alpha) = \{\text{lpar}\}$

Put P1 @ Table[S][lpar]

P2 $X ::= \text{id comma } X$

For terminals in $\text{FIRST}(\alpha) = \{\text{id}\}$

Put P2 @ Table[X][id]

P3 $X ::= \varepsilon$

No terminals in $\text{FIRST}(\alpha)$

ε **is** in $\text{FIRST}(\alpha)$

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

Put P3 @ Table[X][rpar]

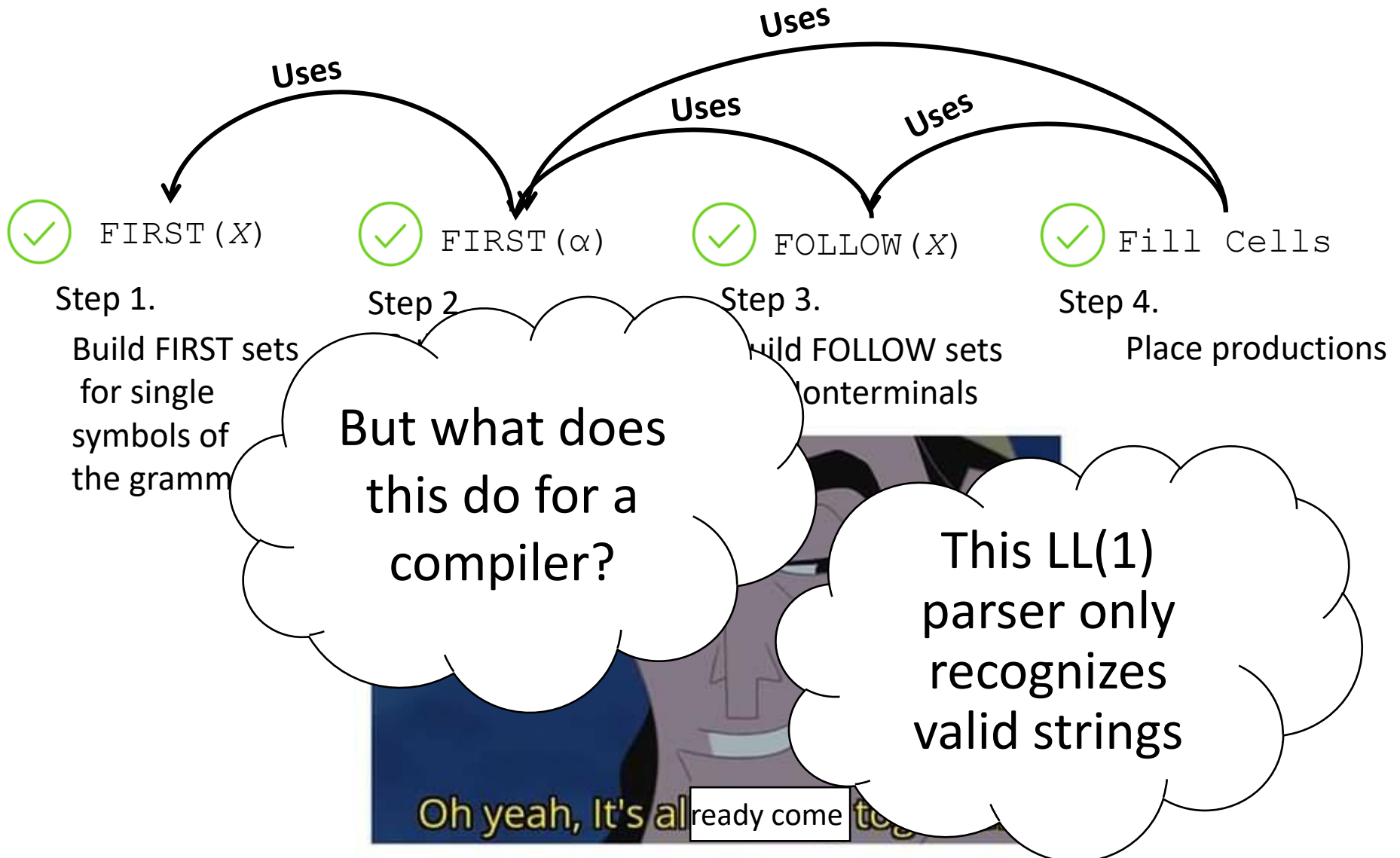
Selector Table Building Procedure

```

for each production  $X ::= \alpha$ 
  for each terminal t in  $\text{FIRST}(\alpha)$ 
    put  $X ::= \alpha$  in Table[X][t]
  if  $\varepsilon$  is in  $\text{FIRST}(\alpha)$ 
    for each terminal t in  $\text{FOLLOW}(X)$ 
      put  $X ::= \alpha$  in Table[X][t]
    
```

Selector Table Dependencies

Building the Selector Table



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

What About the Parse Tree?

Building LL(1)Selector Table

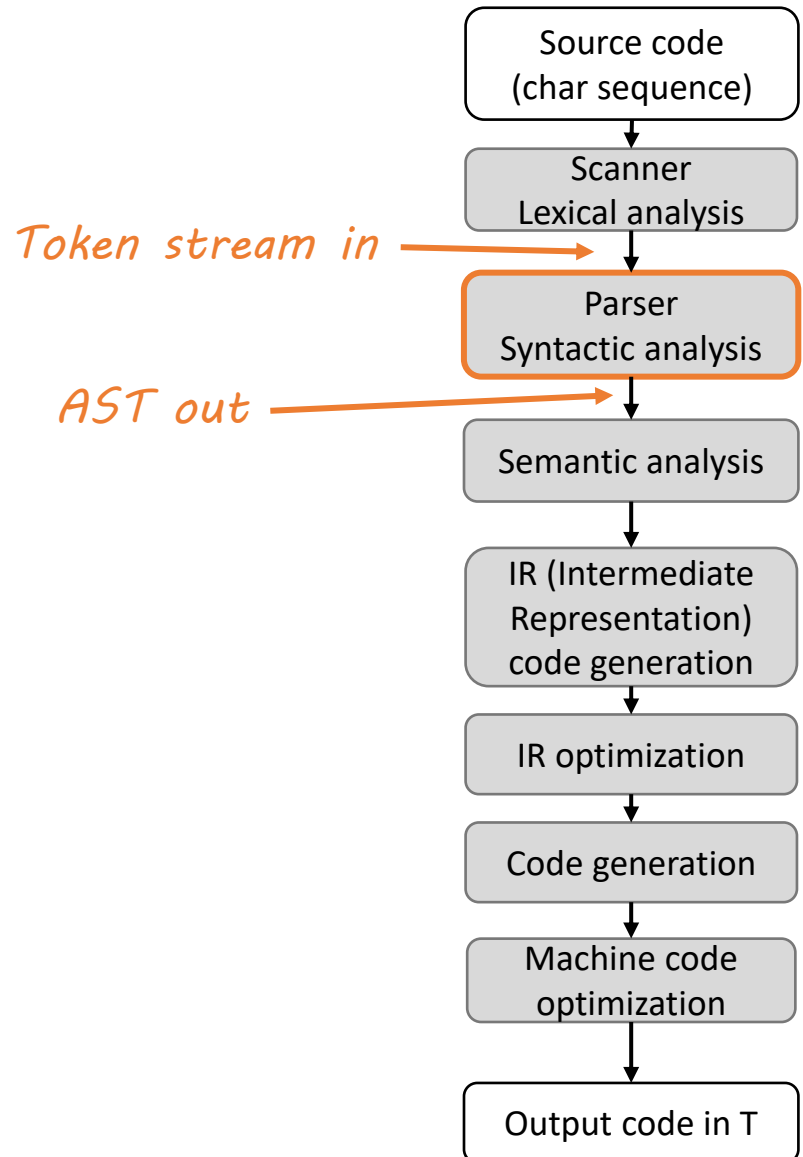


Compiler Just Wants Parse Translation!

Syntax-Directed Translation

Parse tree completely internal to syntactic analysis

- Logical intermediary to Syntax Directed Translation
- As long as we get *the SDT result* we don't need a parse tree



A Quick SDT Recap

Syntax-Directed Translation

Conceptual/Logical workflow:

Grammar

$E ::= E \text{ sub } T$
| T
 $T ::= T \text{ div } F$
| F
 $F ::= \text{intlit}$

Syntax Directed Definition (SDD)

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

Parser Generator

Input

1 - 2

Parser

Recognition and
Parse tree build

Parse Tree

```
      E  
     /|\  
    E sub T  
   /|\  /|\  
  T F  T F  
 /|\  /|\  
F intlit intlit  
|  
intlit
```

Syntax-Directed
Translator

Syntax-Directed
Translation

AST

```
      Sub  
     /  \  
  IntLit IntLit  
   1     2
```

A Quick SDT Recap

Syntax-Directed Translation

Implemented

~~Conceptual/Logical workflow:~~

Grammar

$E ::= E \text{ sub } T$
| T
 $T ::= T \text{ div } F$
| F
 $F ::= \text{intlit}$

Syntax Directed Definition (SDD)

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

Parser Generator

Input

1 - 2

Parser

Recognition and
Parse Tree Build

Parse Tree

E
├── F
│ └── intlit
└── intlit

Syntax-Directed
Translation

Syntax-Directed
Translation

AST

Sub

IntLit
1

IntLit
2

Recognition and Syntax-Directed Translation

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); }
```



Parser Generator

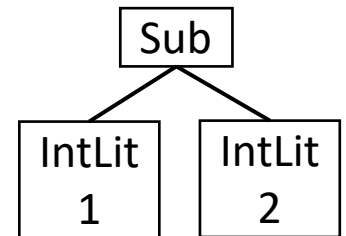


Input

1 - 2

Parser

AST



Recognition and Syntax-Directed Translation

SDT: It's not just for ASTs!

Syntax-Directed Translation

$E ::= E \text{ sub } T$	$\{ \text{LHS.trans} = \text{new SubNode}(E.\text{trans}, T.\text{trans}); \}$	$\{ \text{LHS.trans} = E.\text{trans} - T.\text{trans}; \}$
$\quad T$	$\{ \text{LHS.trans} = T.\text{trans} \}$	$\{ \text{LHS.trans} = T.\text{trans} \}$
$T ::= T \text{ div } F$	$\{ \text{LHS.trans} = \text{new DivNode}(T.\text{trans}, F.\text{trans}); \}$	$\{ \text{LHS.trans} = T.\text{trans} / F.\text{trans}; \}$
$\quad F$	$\{ \text{LHS.trans} = F.\text{trans} \}$	$\{ \text{LHS.trans} = F.\text{trans} \}$
$F ::= \text{intlit}$	$\{ \text{LHS.trans} = \text{new IntLitNode}(\text{intlit.value}); \}$	$\{ \text{LHS.trans} = \text{intlit.value}; \}$

Parser Generator

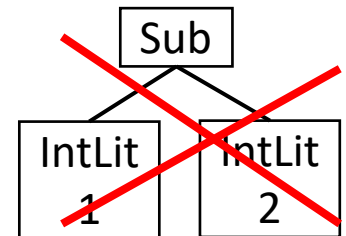
Parser

Input

1 - 2

Value: -1

~~AST~~



Recognition and Syntax-Directed Translation

Lecture Outline

Lecture 10 – SDT

Finish Building LL(1) Parsers

- Filling the Selector Table

Top-Down Syntax-Directed Translation

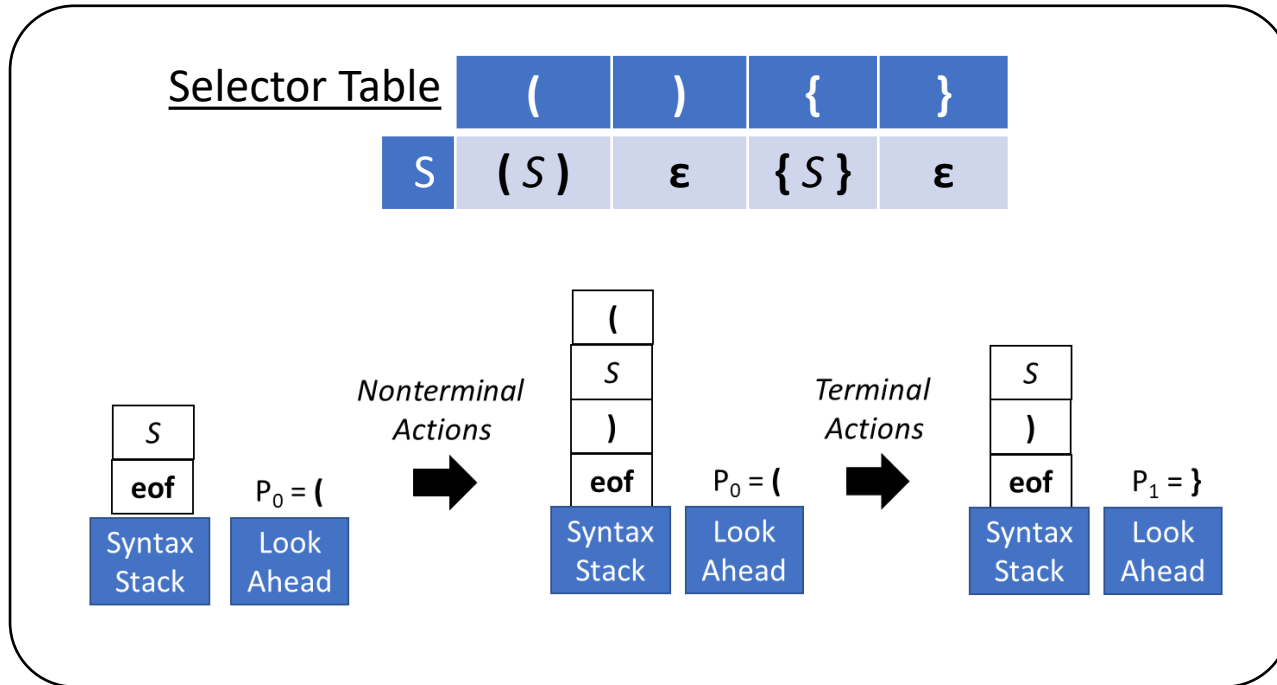
- Parser Tweaks
- Grammar Tweaks



Parsing

Wait *This* Thing can do SDT?

Top-Down Syntax-Directed Translation



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

Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

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

But **when** should we invoke these actions?

CFG

$Expr ::= \epsilon$

| (Expr)

| [Expr]

SDT Rules

LHS.trans = 0

LHS.trans = Expr.trans + 1

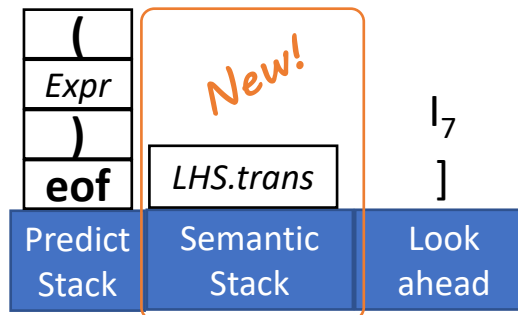
LHS.trans = Expr.trans

SDT Actions

push 0

ExprTrans = pop; push ExprTrans + 1

ExprTrans = pop; push ExprTrans



Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

But **when** should we invoke these actions?

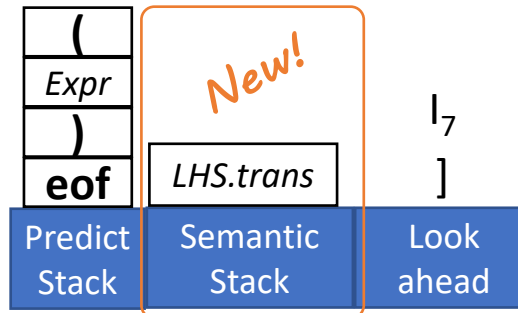
- Number the actions, embed numbers into production RHS
 - Push action numbers onto predict stack, like any other symbol

CFG

$Expr ::= \epsilon \quad \#1$
 | $(Expr) \quad \#2$
 | $[Expr] \quad \#3$

SDT Actions

#1 push 0
#2 $ExprTrans = pop; \text{push } ExprTrans + 1$
#3 $ExprTrans = pop; \text{push } ExprTrans$



Parser Tweaks for SDT

Top-Down Syntax-Directed Translation

But **when** should we invoke these actions?

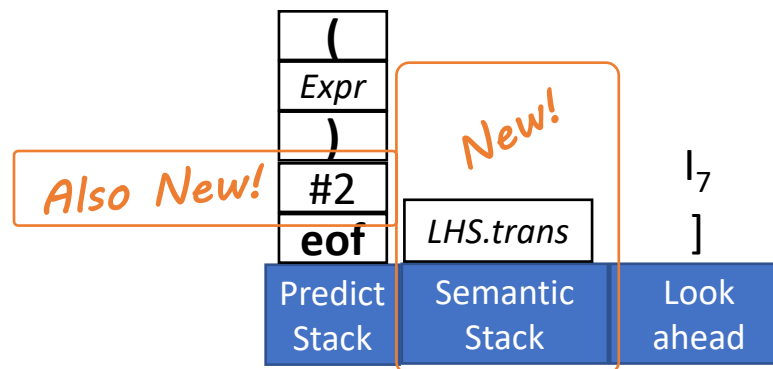
- Number the actions, embed numbers into production RHS
 - Push action numbers onto predict stack, like any other symbol
 - Execute the action when it makes it to the top of the predict stack
- By parse's end, semantic stack reduced to root translation

CFG

$Expr ::= \epsilon \quad \#1$
 | $(Expr) \quad \#2$
 | $[Expr] \quad \#3$

SDT Actions

#1 push 0
#2 $ExprTrans = pop; \text{push } ExprTrans + 1$
#3 $ExprTrans = pop; \text{push } ExprTrans$



Action Numbers, Example 1

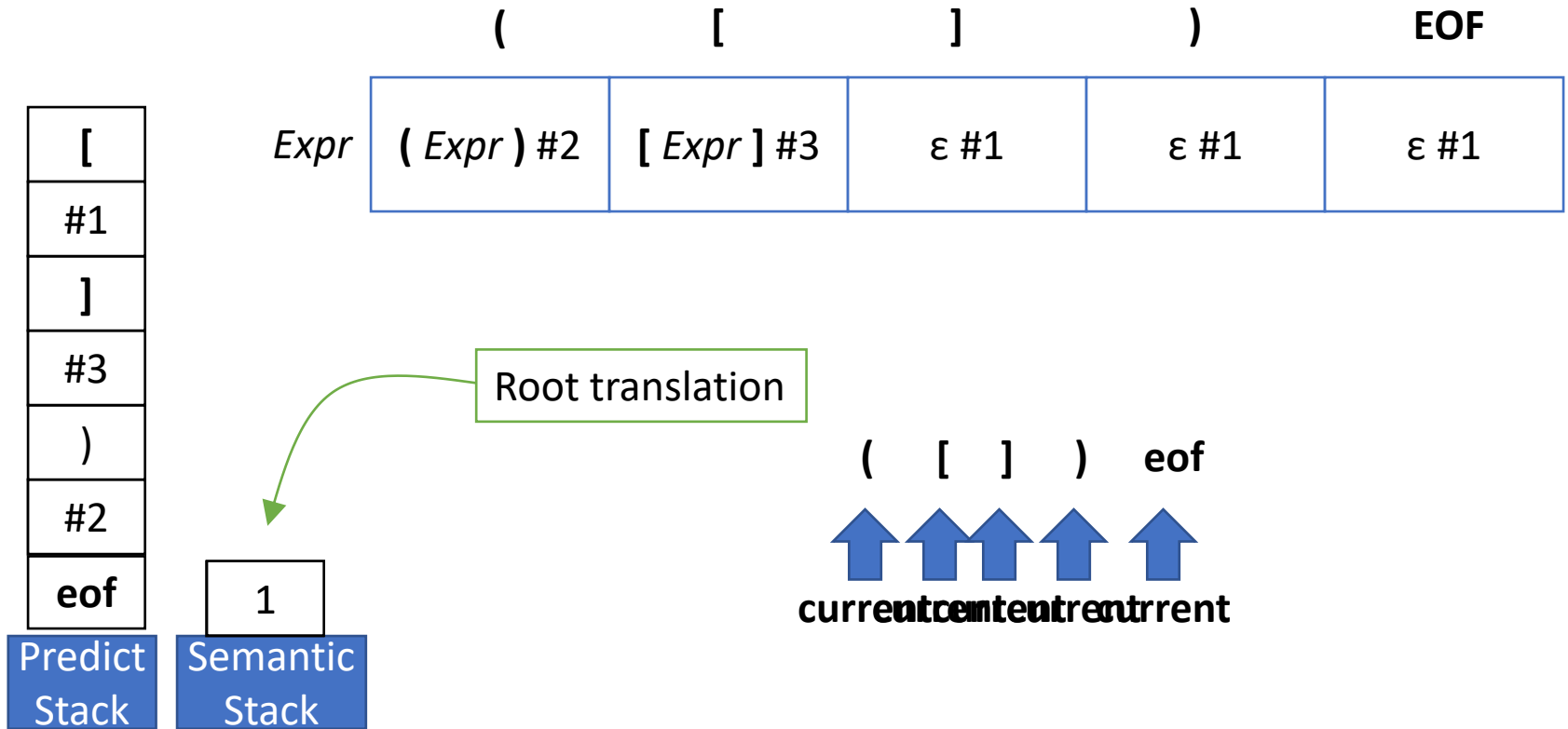
SDT for Top-Down Parsing

CFG

$Expr ::= \epsilon \quad \#1$
 $\quad | (Expr) \quad \#2$
 $\quad | [Expr] \quad \#3$

SDT Actions: Counting Max Parens Depth

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



No-op SDT Actions

SDT for Top-Down Parsing

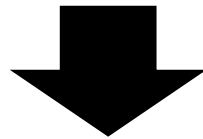
CFG

$Expr \rightarrow \varepsilon$ #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

$Expr \rightarrow \varepsilon$ #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

SDD for Top-Down Parsing

Template

Example

CFG Production

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

SDT Rule

$LHS.trans = \text{combined}$
 $\alpha_1.trans \dots \alpha_n.trans$



Enriched Production

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

Stack Action #1

$t_n = \Gamma.pop()$
 $\dots$
 $t_1 = \Gamma.pop()$
 $t_L = \text{combine } t_1 \dots t_n$
 $\Gamma.push(t_L)$

*Where Γ is the
semantic stack*

CFG Production

$E ::= L \text{ div } R$

SDT Rule

$LHS.trans = L.trans / R.trans$



Enriched Production

$E ::= L \text{ div } R \#1$

Stack Action #1

$t_R = \Gamma.pop()$
 $t_L = \Gamma.pop()$
 $\Gamma.push(t_L / t_R)$

Why Put LHS action at end of Production?

SDT for Top-Down Parsing

Augmented Production

$W ::= X Y Z \#1$

...

Stack Actions

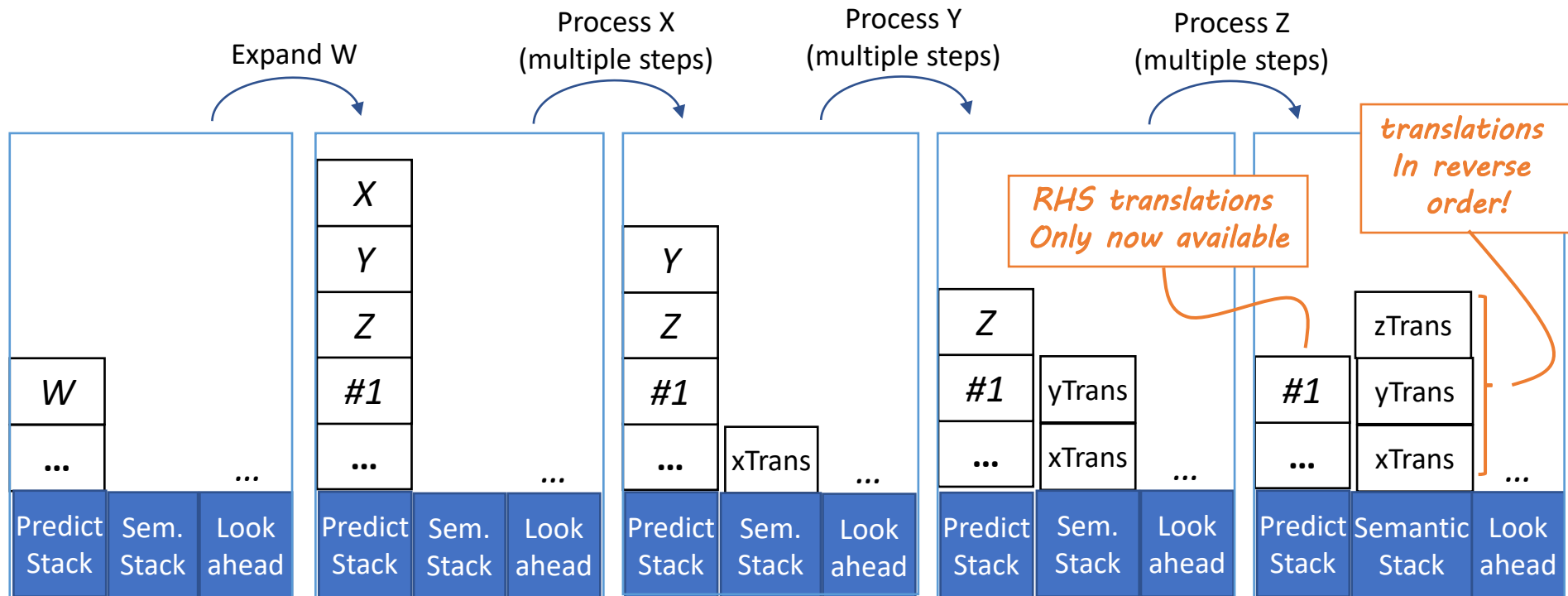
#1 $zTrans = sem.pop()$

$yTrans = sem.pop()$

$xTrans = sem.pop()$

... *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

$Y ::= \#3 \text{ intlit}$

Stack Actions Fragment

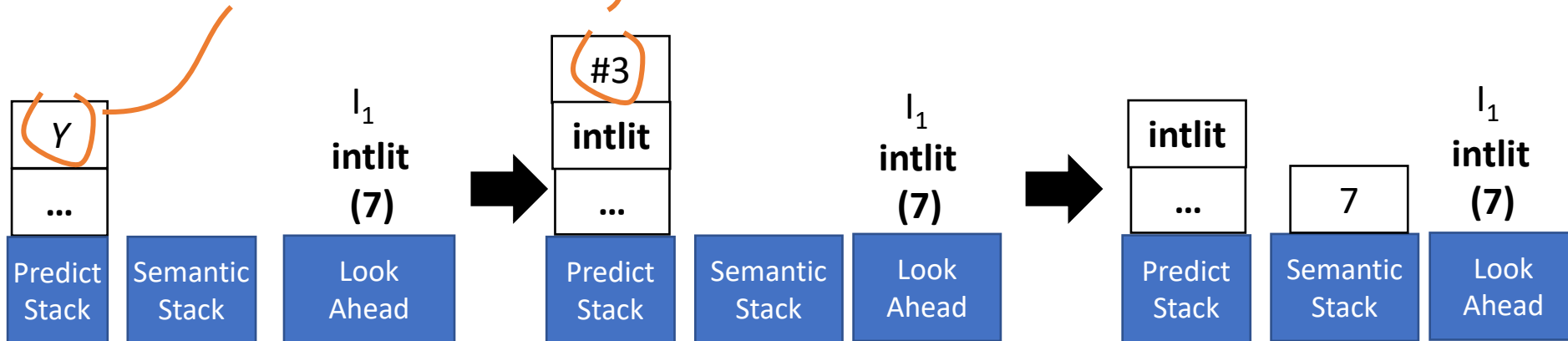
$\#3$ $yTrans = lookahead.value$
 $sem.push(yTrans)$

Input Stream

$7 + 8 + 9$

*Derive RHS
Via selector table
(not shown)*

*Put lookahead
value on sem stack*



Preparing SDT Schemes

SDT for Top-Down Parsing

CFG

$Expr ::= Expr \text{ minus } Term$
 | $Term$
 $Term ::= \text{intlit}$



Augmented CFG

$Expr ::= Expr \text{ minus } Term \#1$
 | $Term$
 $Term ::= \#2 \text{ intlit}$

SDD Rules for Evaluation

$LHS.trans = Expr.trans - Term.trans$
 $LHS.trans = Term.trans$
 $LHS.trans = \text{intlit.value}$



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

$Expr ::= Expr \text{ minus } Term$
 | $Term$
 $Term ::= \text{intlit}$



Augmented CFG

$Expr ::= Expr \text{ minus } Term \#1$
 | $Term$
 $Term ::= \#2 \text{intlit}$

SDD Rules for Evaluation

$LHS.trans = T.trans$
 $LHS.trans = T.trans$

*Hey! This
grammar
isn't LL(1)*

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

$\#2 \text{ sem.push(lookahead.value)}$

Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

$E ::= E \text{ minus } T \#1$
 $| T$
 $T ::= \#2 \text{ intlit}$



LL(1) CFG

$E ::= T E'$
 $E' ::= \text{minus } T \#1 E' \mid \epsilon$
 $T ::= \#2 \text{ intlit}$



	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

#1 $tTrans = \text{sem.pop}()$;
 $eTrans = \text{sem.pop}()$;
 $LTrans = eTrans - tTrans$;
 $\text{sem.push}(LTrans)$
#2 $\text{sem.push}(\text{intlit.value})$

Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$
 $\#2$ $\text{sem.push}(\text{intlit.value})$

Time to see the
parser in action!

Actions Preserved Through Transforms

SDT for Top-Down Parsing

Augmented CFG

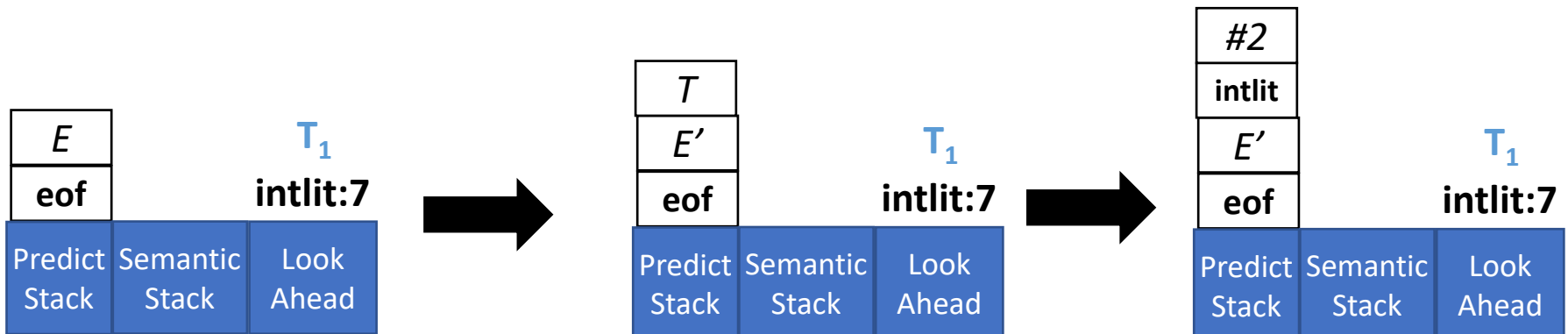
$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$
 $\#2$ $\text{sem.push}(\text{intlit.value})$

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

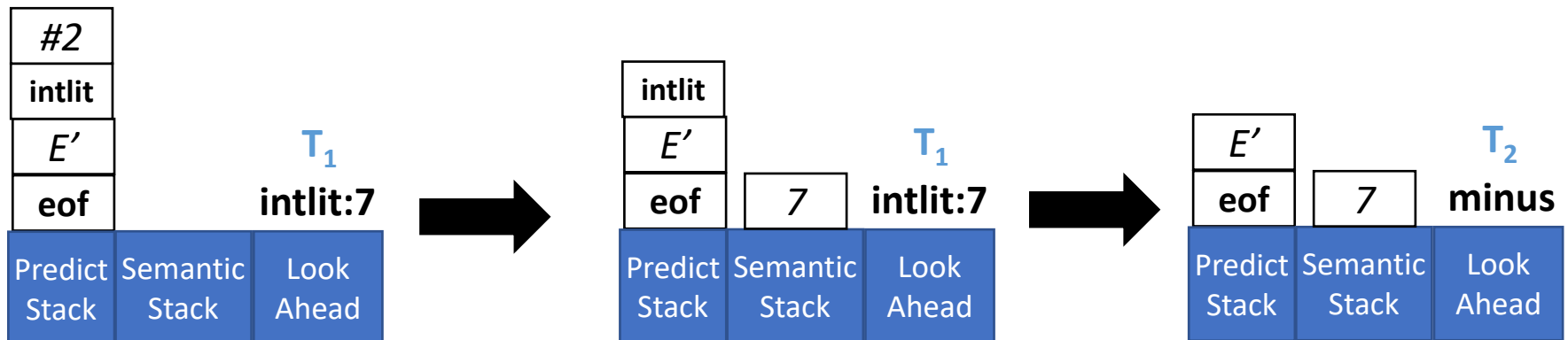
$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$
 $\#2$ $\text{sem.push}(\text{look.value})$

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

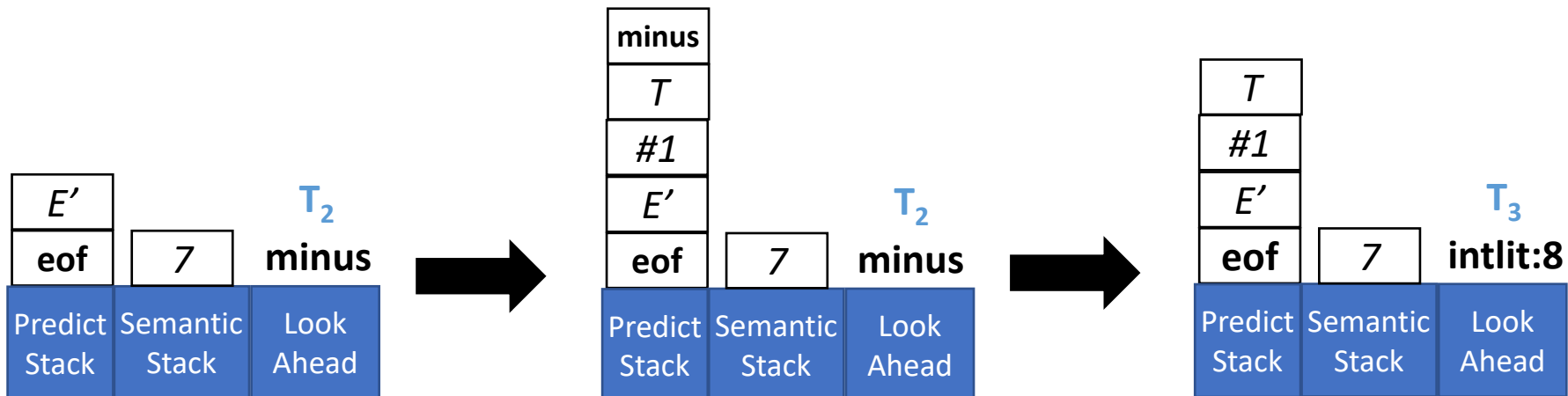
$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$
 $\#2$ $\text{sem.push}(\text{look.value})$

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

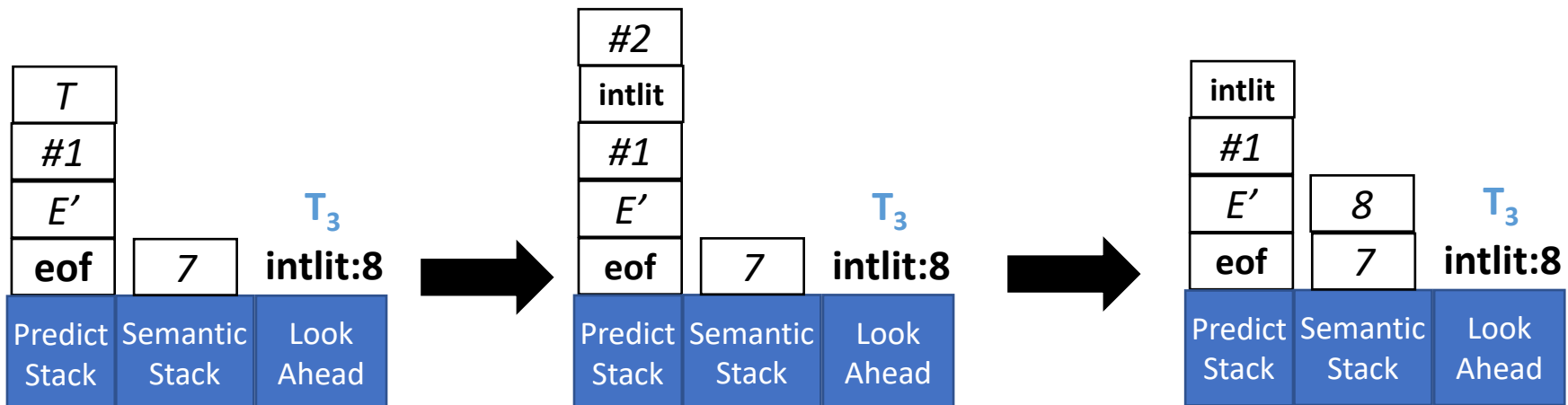
$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$;
 $\#2$ $\text{sem.push}(\text{look.value})$

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

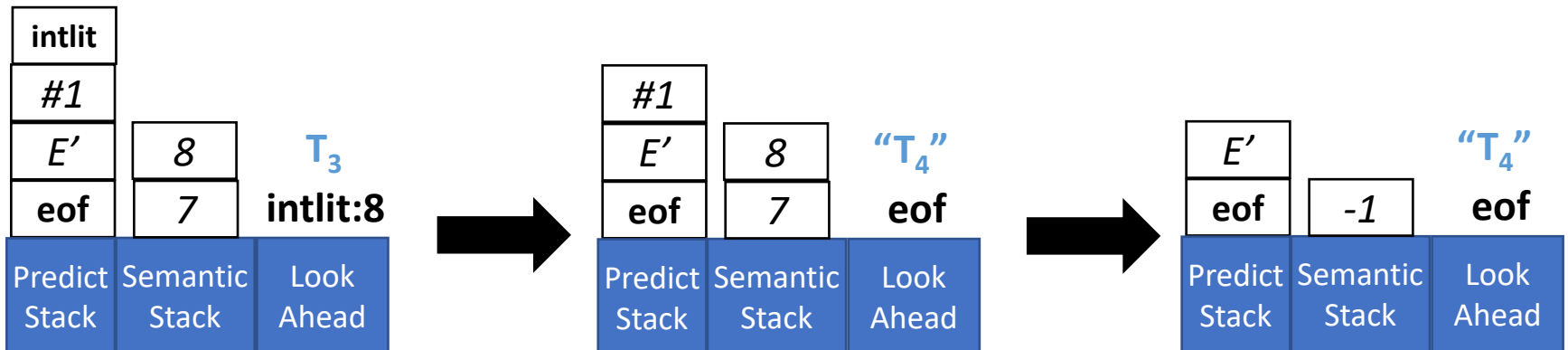
$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$;
 $\#2$ $\text{sem.push}(\text{look.value})$

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

$E ::= E \text{ minus } T \#1$
 $\quad \mid T$
 $T ::= \#2 \text{ intlit}$

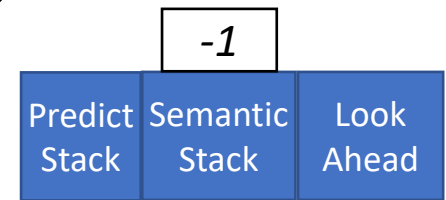
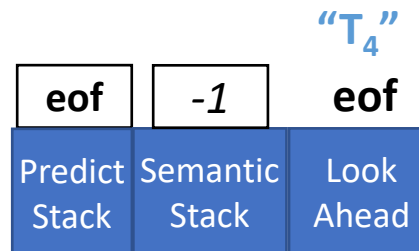
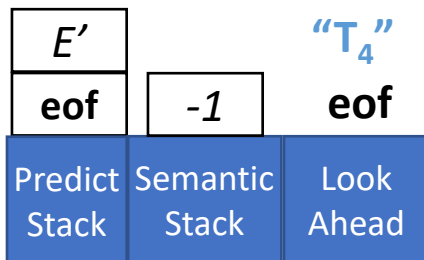
	intlit	minus	EOF
E	$T E'$		
E'		$\text{minus } T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

Eval Stack Actions

$\#1$ $tTrans = \text{sem.pop}()$;
 $\quad eTrans = \text{sem.pop}()$;
 $\quad LTrans = eTrans - tTrans$;
 $\quad \text{sem.push}(LTrans)$
 $\#2$ $\text{sem.push}(\text{look.value})$

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

$E ::= E + T \#1$
 $| T$
 $T ::= \#2 \text{ intlit}$

	intlit	+	EOF
E	$T E'$		
E'		$+ T \#1 E'$	ϵ
T	$\#2 \text{ intlit}$		

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

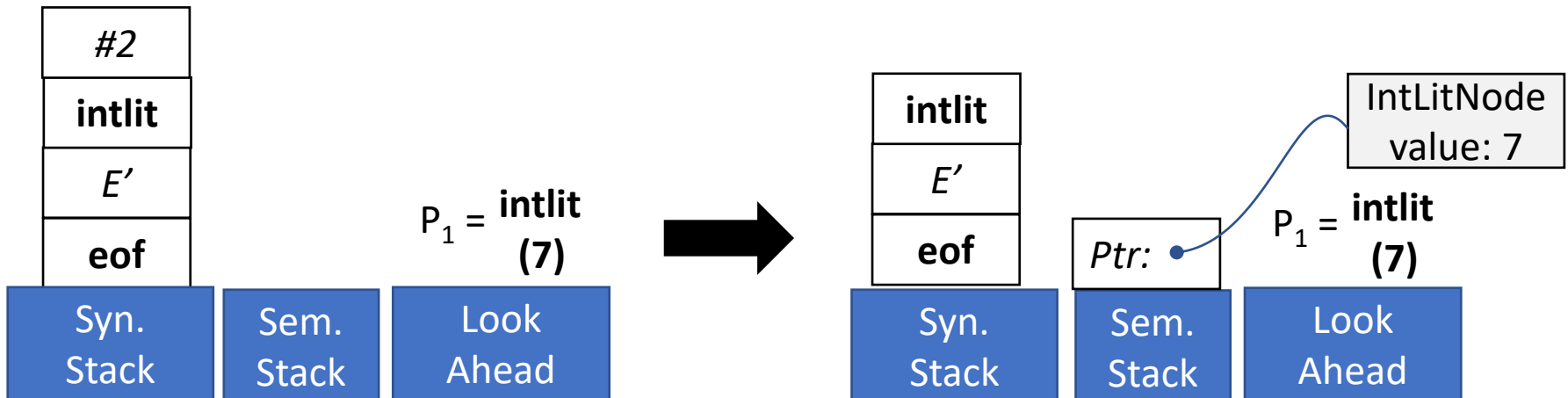
Selector Table

	intlit	+	EOF
E	$T E'$		
E'		$+ T \#1 E'$	ϵ
T	$\#2 \text{intlit}$		

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





A FEW

Transitions

LATER

What About the AST?

SDT for Top-Down Parsing

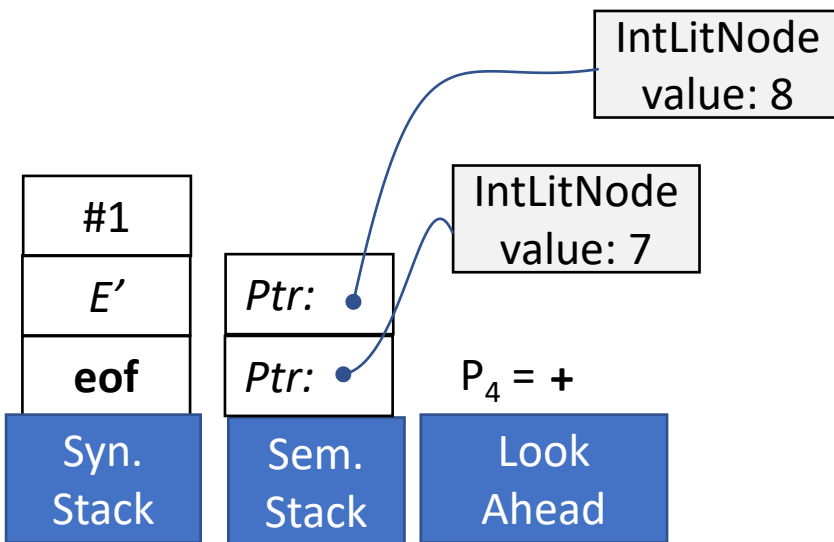
Selector Table

	intlrit	+	EOF
E	$T E'$		
E'		$+ T \#1 E'$	ϵ
T	$\#2 \text{intlrit}$		

AST Building Stack Actions

```
#1 tTrans = sem.pop() ;  
   eTrans = sem.pop() ;  
   LTrans = PlusNode(eTrans,tTrans)  
   sem.push(LTrans)  
  
#2 LTrans = IntLitNode(intlrit.value)  
   sem.push(LTrans)
```

Input: 7 + 8 + 9



What About the AST?

SDT for Top-Down Parsing

Selector Table

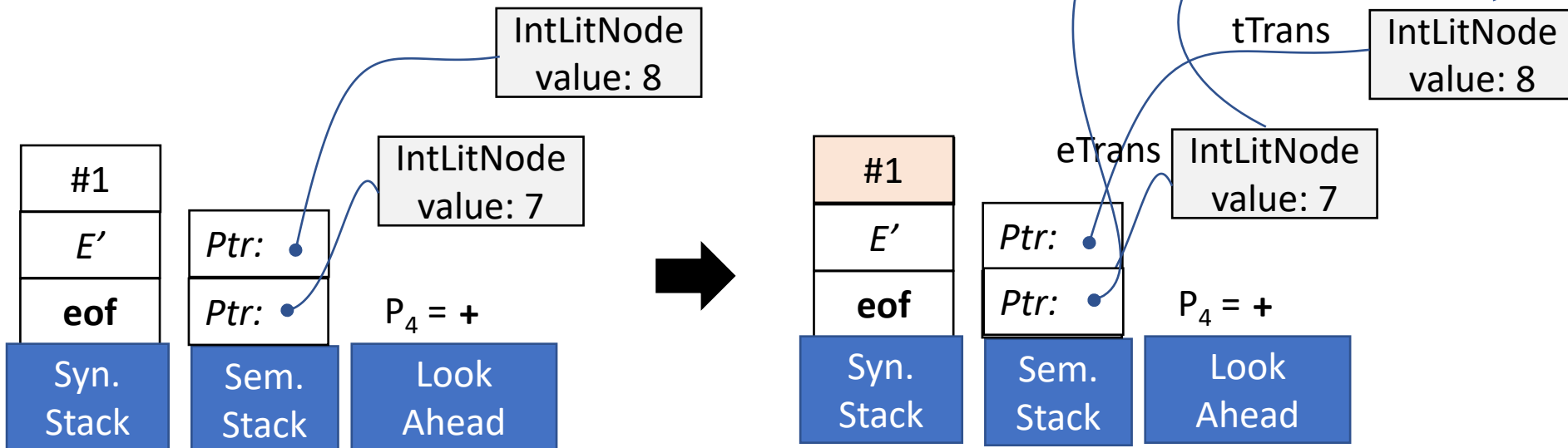
	intlrit	+	EOF
E	$T E'$		
E'		$+ T \#1 E'$	ϵ
T	$\#2 \text{intlrit}$		

AST Building Stack Actions

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

#2 LTrans = IntLitNode(intlrit.value)
   sem.push(LTrans)
```

Input: 7 + 8 + 9



This is us “Fixing” the Tree

SDT for Top-Down Parsing

“We’ll fix the tree later”

- _ (ツ) _ / -

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:

- Create attribute grammar (CFG + SDD)
- Convert SDD rules to stack actions
- Build 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)