Announcement: Midterm Prep

Midterm is Tuesday, 3/10 here

List of topics

- Up to and including lecture on Tuesday’s the 5th

Length 1hr 10min

No extra material allowed – just bring a pen

Sample midterm available online

- Recommended that you do this by Thursday
- We’ll review it in class

Syntax Directed Translation for Top-Down Parsing

Last Time: Built LL(1) Predictive Parser

FIRST and FOLLOW sets define the parse table

If the grammar is LL(1), the table is unambiguous

- i.e., each cell has at most one entry

If the grammar is not LL(1) we can attempt a transformation sequence:

1. Remove left recursion
2. Left-factoring

Grammar transformations affect the structure of the parse tree. How does this affect syntax-directed translation (in particular, parse tree → AST)?

Today

Review Parse Table Construction

- 2 examples

Show how to do Syntax-Directed Translation using an LL(1) parser

FIRST(α) for α = $Y_1 Y_2 \ldots Y_k$

Add FIRST(Y_1) - {ε}

If ε is in FIRST($Y_{1 \text{ to } i-1}$): add FIRST(Y_i) – {ε}

If ε is in all RHS symbols, add ε

FOLLOW(A) for *X* ⟶ α *A* β

If A is the start, add **eof**

Add FIRST(β) – {ε}

Add FOLLOW(*X*) if ε in FIRST(β) or β empty

Table[X][t]

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

CFG

S ⟶ B **c** | D B

B ⟶ **a b** | **c** S

D ⟶ **d** | ε

Not LL(1)

FIRST (S)	=	{ **a, c, d** }
FIRST (B)	=	{ **a**, **c** }
FIRST (D)	=	{ **d**, ε }
FIRST (B **c**)	=	{ **a**, **c** }
FIRST (D B)	=	{ **d**, **a**, **c** }
FIRST (**a b**)	=	{ **a** }
FIRST (**c** *S*)	=	{ **c** }
FIRST (**d**)	=	{ **d** }
FIRST (ε)	=	{ ε }

FOLLOW (S)	=	{ **eof, c** }
FOLLOW (B)	=	{ **c, eof** }
FOLLOW (D)	=	{ **a, c** }

	a	**b**	**c**	**d**	**eof**
S	B **c** D B		B **c** D B	D B	
B	**a b**		**c** *S*		
D	ε		ε	**d**	

FIRST(α) for $\alpha = Y_1\ Y_2 \ldots Y_k$

Add FIRST(Y_1) - {ε}

If ε is in FIRST($Y_{1\text{ to } i-1}$): add FIRST(Y_i) – {ε}

If ε is in all RHS symbols, add ε

FOLLOW(A) for $X \longrightarrow \alpha\ A\ \beta$

If A is the start, add **eof**

Add FIRST(β) – {ε}

Add FOLLOW(*X*) if ε in FIRST(β) or β empty

Table[X][t]

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

CFG

$$S \rightarrow \mathbf{(}\ S\ \mathbf{)}\ |\ \mathbf{\{}\ S\ \mathbf{\}}\ |\ \varepsilon$$

FIRST (*S*) = { **{** , **(** , **ε** }

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

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

FIRST (ε) = { ε }

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

	(	**)**	**{**	**}**	**eof**
S	(S)	ε	{ S }	ε	ε

FIRST(α) for $\alpha = Y_1 Y_2 \ldots Y_k$

Add FIRST(Y_1) - {ε}

If ε is in FIRST($Y_{1 \text{ to } i-1}$): add FIRST(Y_i) – {ε}

If ε is in all RHS symbols, add ε

FOLLOW(A) for $X \longrightarrow \alpha\ A\ \beta$

If A is the start, add **eof**

Add FIRST(β) – {ε}

Add FOLLOW(*X*) if ε in FIRST(β) or β empty

Table[X][t]

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

CFG

$$S \rightarrow + S \mid \varepsilon$$

FIRST (*S*) = { +, **ε** }

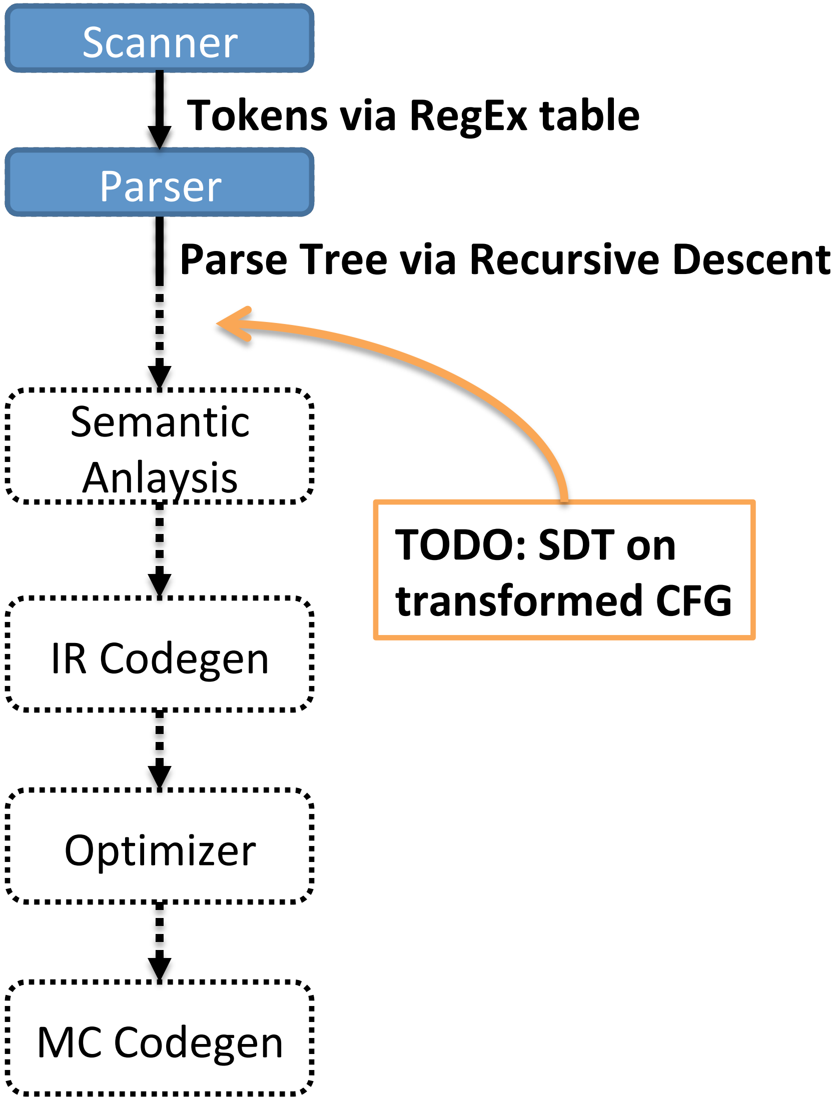
FIRST (+ *S*) = { + }

FIRST (ε) = { ε }

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

	+	**eof**
S	**+** S	ε

How's that Compiler Looking?



Scanner

Tokens via RegEx table

Parser

Parse Tree via Recursive Descent

Semantic Anlaysis

IR Codegen

Optimizer

MC Codegen

TODO: SDT on transformed CFG

Implementing SDT for LL(1) Parser

So far, SDT shown as second (bottom-up) pass over parse tree

The LL(1) parser never needed to explicitly build the parse tree (implicitly tracked via stack)

Naïve approach: build the parse tree explicitly

Semantic Stack

Instead of building the parse tree, give parser second, *semantic* stack

- Holds nonterminals' tra

SDT rules converted to a

- Pop translations of RHS non off
- Push computed transl ion of LHS nonterm on

<u>Translation goal:</u>

- Count the number of occurrences of matched pairs of rounded parens: "**(** … **)**"
- Ignore occurrences of matched pairs of square brackets: "**[** … **]**"

CFG		**SDT Rules**	**SDT Actions**
Expr ⟶	ε	Expr.trans = 0	push 0
|	**(** Expr **)**	Expr.trans = Expr_2.trans + 1	Expr_2.trans = pop; push Expr_2.trans + 1
|	**[** Expr **]**	Expr.trans = Expr_2.trans	Expr_2.trans = pop; push Expr_2.trans

Action Numbers

Need to define *when* to fire the SDT Action

- Not immediately obvious since SDT is bottom-up

Solution

- Number actions and put them on the symbol stack!
- Add action number symbols at end of the productions

CFG	**SDT Actions**
Expr ⟶ ε #1	#1 push 0
| **(** Expr **)** #2	#2 Expr_2.trans = pop; push Expr_2.trans + 1
| **[** Expr **]** #3	#3 Expr_2.trans = pop; push Expr_2.trans

Action Numbers: Example 1

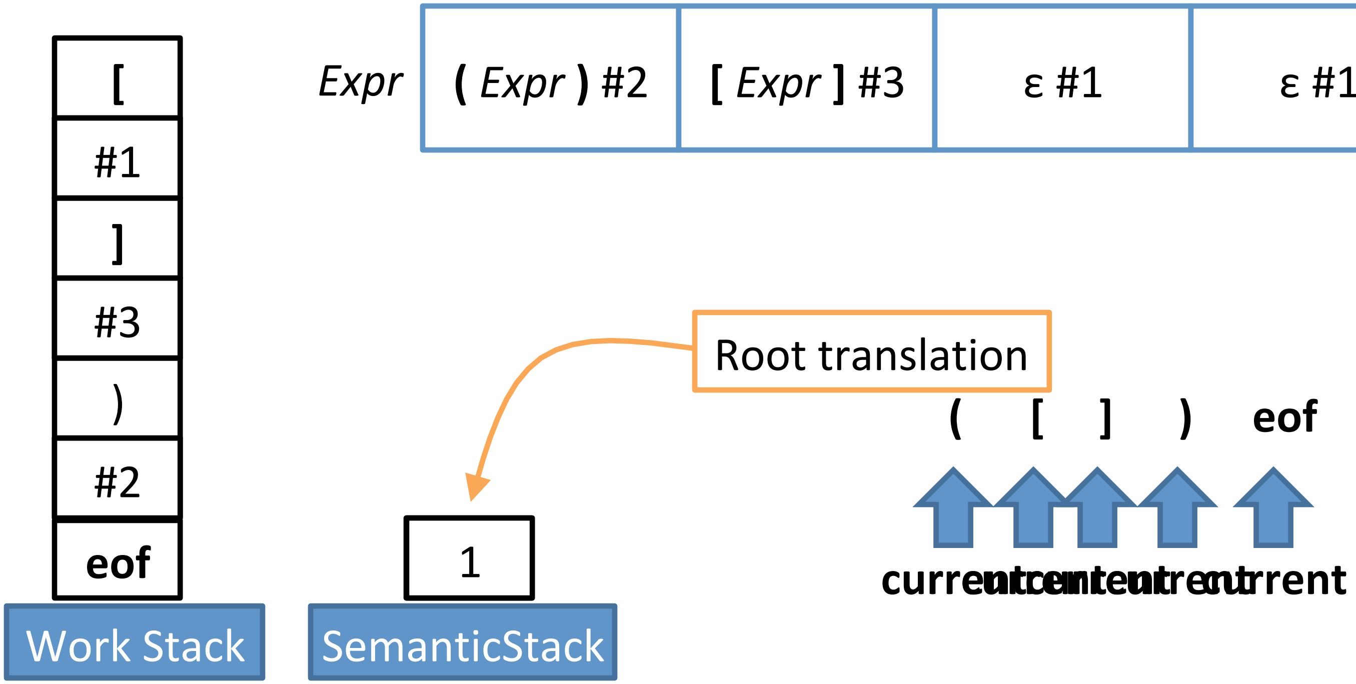
CFG

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

SDT Actions: Counting Max Parens Depth

#1 push 0
#2 $Expr_2$.trans = pop; push($Expr_2$.trans + 1)
#3 $Expr_2$.trans = pop; push($Expr_2$.trans)

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



No-op SDT Actions

CFG

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

SDT Actions: Counting Max Parens Depth

#1 push 0
#2 Expr$_2$.trans = pop; push(Expr$_2$.trans + 1)
#3 Expr$_2$.trans = pop; push(Expr$_2$.trans)

Useless rule

CFG

Expr ⟶ ε #1
| **(** *Expr* **)** #2
| **[** *Expr* **]**

SDT Actions: Counting Max Parens Depth

#1 push 0
#2 Expr$_2$.trans = pop; push(Expr$_2$.trans + 1)

Placing Action Numbers

Action numbers go after their corresponding nonterminals, before their corresponding terminal

Translations popped right to left in action

CFG

Expr ⟶ *Expr + Term* #1
| *Term*
Term ⟶ *Term * Factor* #2
| *Factor*
Factor ⟶ #3 **intlit**

SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(tTrans + eTrans)

#2 fTrans = pop; tTrans = pop ; push(fTrans * tTrans)

#3 push(**intlit**.value)

Placing Action Numbers: Example

Write SDT Actions and place action numbers to get the **product** of a *ValList* (i.e. multiply all elements)

CFG

List ⟶ *Val List'* #1

List' ⟶ *Val List'* #2

| ε #3

Val ⟶#4 **intlit**

SDT Actions

The order matters. In the reverse order.

#1 LTrans = pop ; vTrans = pop ; push(LTrans * vTrans)

#2 LTrans = pop; vTrans = pop ; push(LTrans * vTrans)

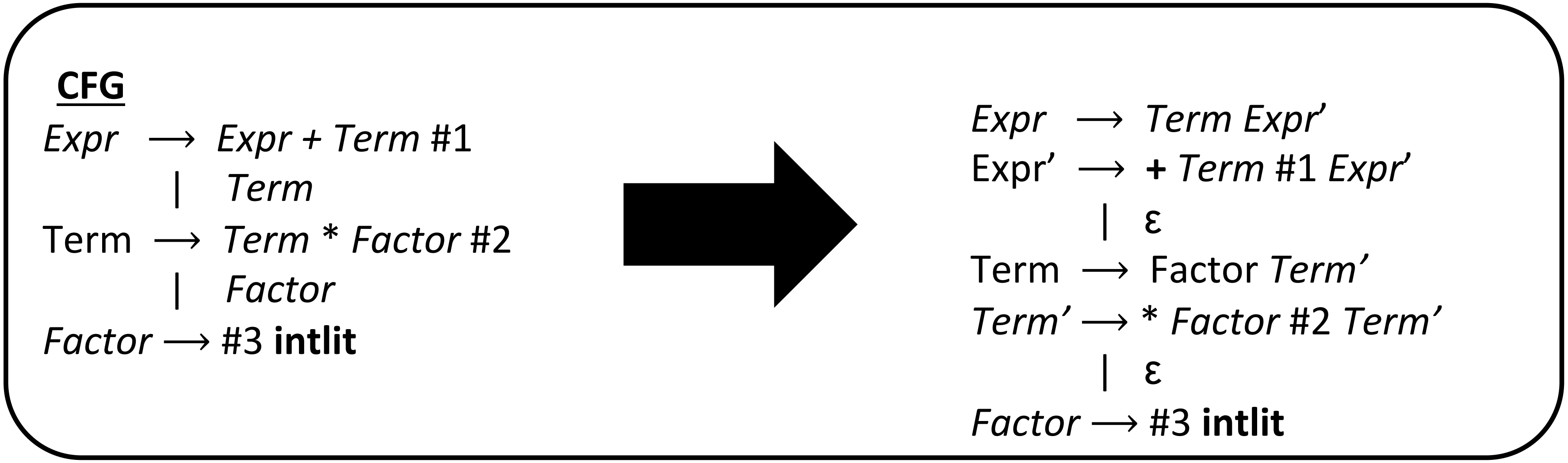
#3 push(1)

#4 push(**intlit**.value)

Action Numbers: Benefits

Plans SDT actions using the work stack

Robust to previously introduced grammar transformations



CFG

Expr ⟶ *Expr + Term* #1
| *Term*
Term ⟶ *Term* * *Factor* #2
| *Factor*
Factor ⟶ #3 **intlit**

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ Factor *Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**

SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(tTrans + eTrans)

#2 fTrans = pop; tTrans = pop ; push(fTrans * tTrans)

#3 push(**intlit**.value)

Example: SDT on Transformed Grammar

CFG

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ *Factor Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)

#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)

#3 push(**intlit**.value)

Example: SDT on Transformed Grammar

CFG

Expr ⟶ *Term Expr'*
Expr' ⟶ **+** *Term* #1 *Expr'*
| ε
Term ⟶ *Factor Term'*
Term' ⟶ * *Factor* #2 *Term'*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

SD

#1 tTr rans + tTrans)

ans * fTrans)

First(Factor) = { intlit, (}
First(Term') = { *, }
First(Term) = { intlit, (}
First(Expr') = { +, }
First(Expr) = { intlit, (}

First(Term Expr') = { intlit, (}
First(+ Term #1 Expr') = { + }
First() = { }
First(Factor Term') = { intlit, (}
First(* Factor #2 Term) = { * }
First() = { }
First(#3 intlit) = { intlit }
First((Expr)) = { (}

Follow(Expr) = { eof,) }
Follow(Expr') = { eof,) }
Follow(Term) = { +, eof,) }
Follow(Term') = { +, eof,) }
Follow(Factor) = { *, +, eof,) }

Example: SDT on Transformed Grammar

CFG

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ *Factor Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)
#3 push(**intlit**.value)

	+	*****	**(**	**)**	**intlit**	**eof**
Expr			*Term Expr’*		*Term Expr’*	
Expr’	+ *Term* #1 *Expr’*			ε		ε
Term			*Factor Term’*		*Factor Term’*	
Term’	ε	* *Factor* #2 *Term’*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	

CFG

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ *Factor Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

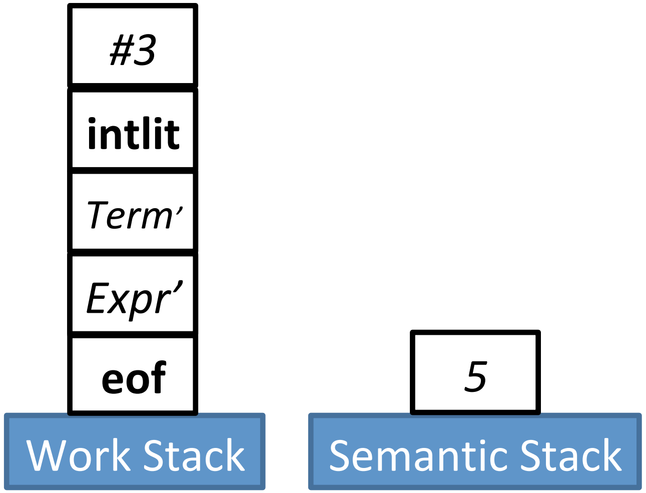
SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)

#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)

#3 push(**intlit**.value)

	+	*	(	)	intlit	eof
Expr			*Term Expr’*		*Term Expr’*	
Expr’	+ *Term* #1 *Expr’*			ε		ε
Term			*Factor Term’*		*Factor Term’*	
Term’	ε	* *Factor* #2 *Term’*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	



Input: 5 + 3 * 2 **eof**

current

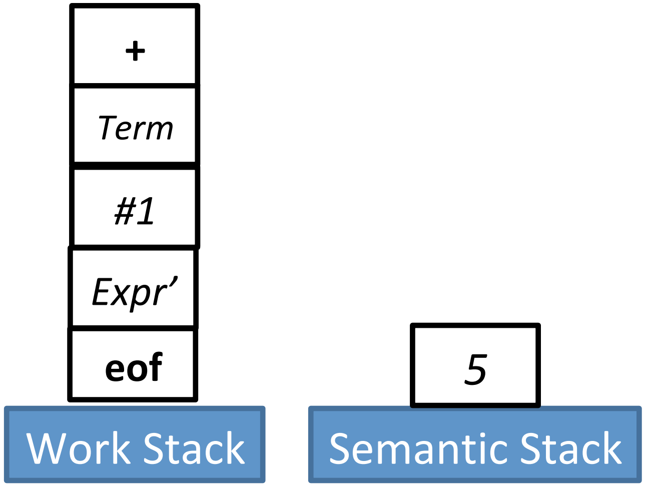
CFG

Expr ⟶ *Term Expr'*
Expr' ⟶ **+** *Term* #1 *Expr'*
| ε
Term ⟶ *Factor Term'*
Term' ⟶ * *Factor* #2 *Term'*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

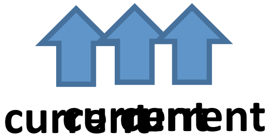
SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)
#3 push(**intlit**.value)

	+	*	(	)	intlit	eof
Expr			*Term Expr'*		*Term Expr'*	
Expr'	+ *Term* #1 *Expr'*			ε		ε
Term			*Factor Term'*		*Factor Term'*	
Term'	ε	* *Factor* #2 *Term'*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	



Input: 5 + 3 * 2 **eof**



CFG

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ *Factor Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

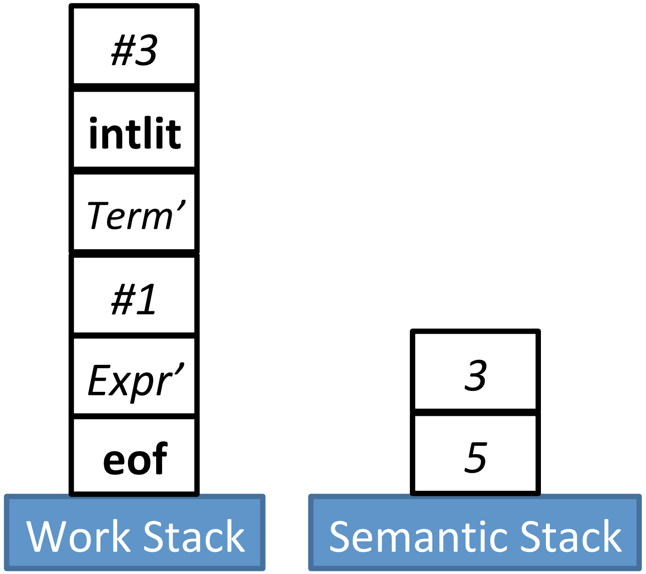
SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)

#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)

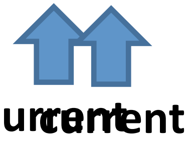
#3 push(**intlit**.value)

	+	*	(	)	intlit	eof
Expr			*Term Expr’*		*Term Expr’*	
Expr’	+ *Term* #1 *Expr’*			ε		ε
Term			*Factor Term’*		*Factor Term’*	
Term’	ε	* *Factor* #2 *Term’*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	


#3
intlit
Term’
#1
Expr’
eof
Work Stack

3
5
Semantic Stack

Input: 5 + 3 * 2 **eof**


current
current

CFG

Expr ⟶ *Term Expr’*
Expr’ ⟶ **+** *Term* #1 *Expr’*
| ε
Term ⟶ *Factor Term’*
Term’ ⟶ * *Factor* #2 *Term’*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

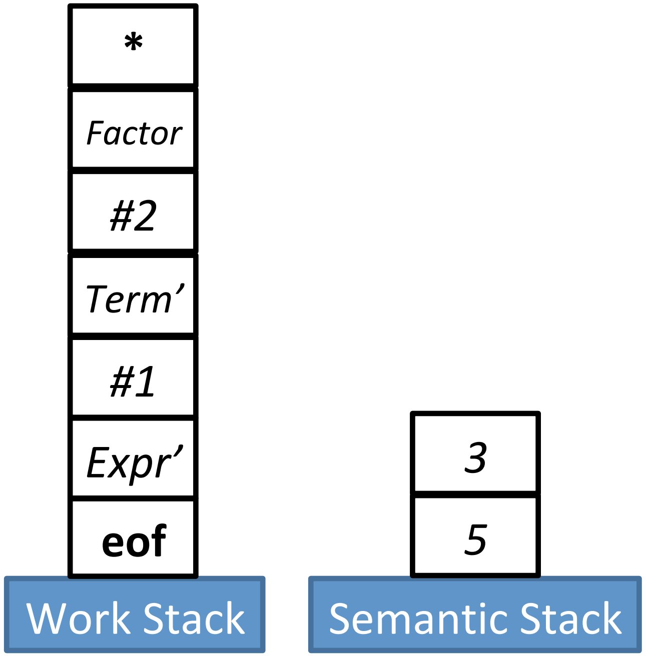
SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)

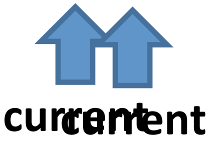
#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)

#3 push(**intlit**.value)

	+	*	(	)	intlit	eof
Expr			*Term Expr’*		*Term Expr’*	
Expr’	+ *Term* #1 *Expr’*			ε		ε
Term			*Factor Term’*		*Factor Term’*	
Term’	ε	* *Factor* #2 *Term’*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	



Input: 5 + 3 * 2 **eof**



CFG

Expr ⟶ *Term Expr'*
Expr' ⟶ **+** *Term* #1 *Expr'*
| ε
Term ⟶ *Factor Term'*
Term' ⟶ * *Factor* #2 *Term'*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

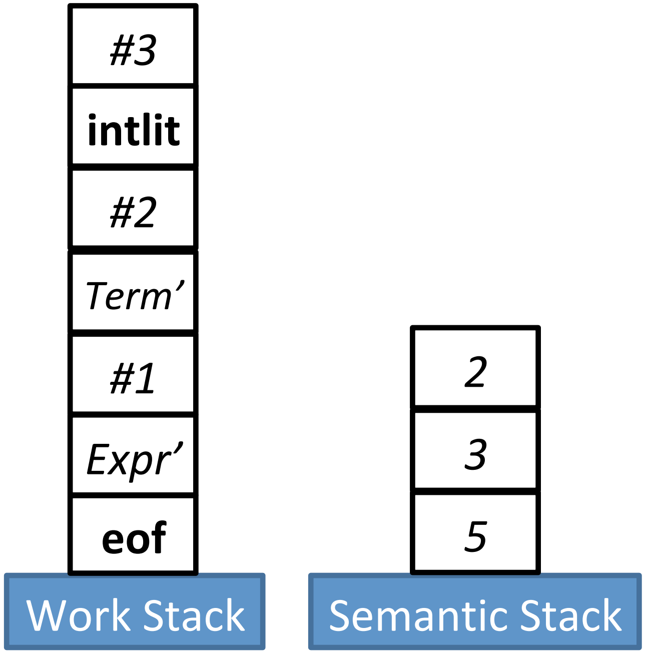
SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)

#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)

#3 push(**intlit**.value)

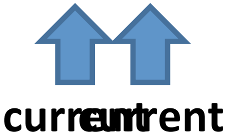
	+	*	(	)	intlit	eof
Expr			*Term Expr'*		*Term Expr'*	
Expr'	+ *Term* #1 *Expr'*			ε		ε
Term			*Factor Term'*		*Factor Term'*	
Term'	ε	* *Factor* #2 *Term'*		ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	



#3
intlit
#2
Term'
#1
Expr'
eof
Work Stack

2
3
5
Semantic Stack

Input: 5 + 3 * 2 **eof**



current
current

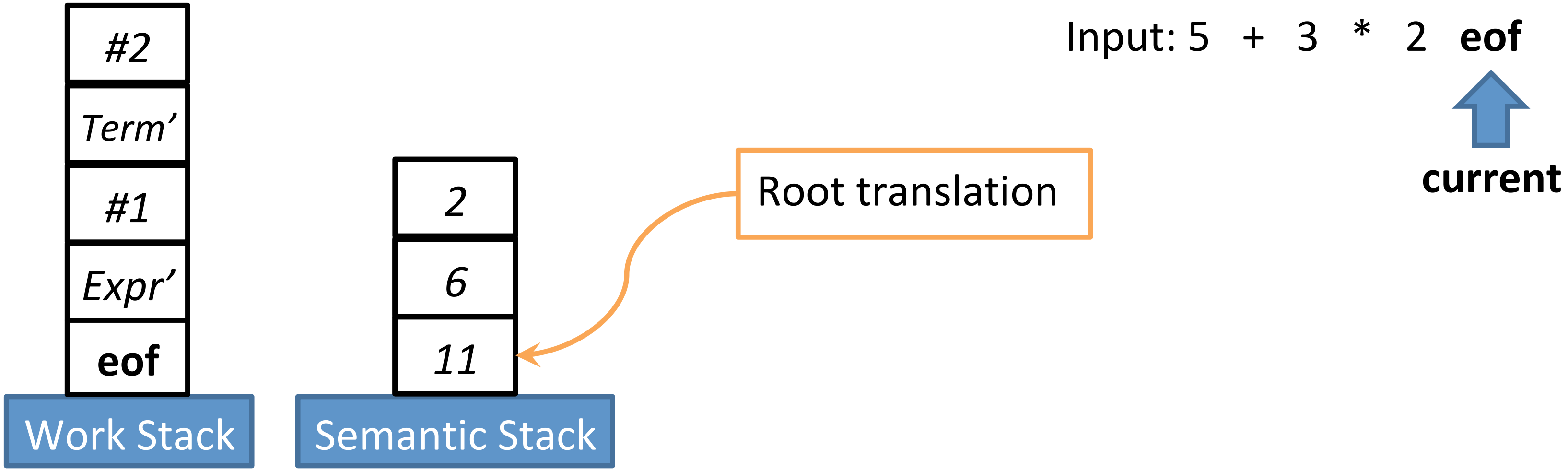
CFG

Expr ⟶ *Term Expr'*
Expr' ⟶ **+** *Term* #1 *Expr'*
| ε
Term ⟶ *Factor Term'*
Term' ⟶ * *Factor* #2 *Term'*
| ε
Factor ⟶ #3 **intlit**
| **(** *Expr* **)**

SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
#2 fTrans = pop; tTrans = pop ; push(tTrans * fTrans)
#3 push(**intlit**.value)

	+	*	(	)	intlit	eof
Expr			*Term Expr'*		*Term Expr'*	
Expr'	+ *Term* #1 *Expr'*			ε		ε
Term			*Factor Term'*		*Factor Term'*	
Term'	ε		* *Factor* #2 *Term'*	ε		ε
Factor			**(** *Expr* **)**		#3 **intlit**	


#2
Term'
#1
Expr'
eof
Work Stack
2
6
11
Semantic Stack
Root translation
Input: 5 + 3 * 2 **eof**
current

What about ASTs?

Push and pop AST nodes on the stack

Keep field references to nodes that we pop

CFG

Expr ⟶ *Expr + Term* #1
| *Term*
Term ⟶ #2 **intlit**

Transformed CFG

Expr ⟶ Term *Expr'*
Expr' ⟶ **+** *Term* #1 *Expr'*
| ε
Term ⟶ #2 **intlit**

"Evaluation" SDT Actions

#1 tTrans = pop ;
eTrans = pop ;
push(eTrans + tTrans)

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

"AST" SDT Actions

#1 tTrans = pop ;
eTrans = pop ;
push(new PlusNode(tTrans, eTrans))

#2 push(new IntLitNode(**intlit**.value))

AST Example

Transformed CFG

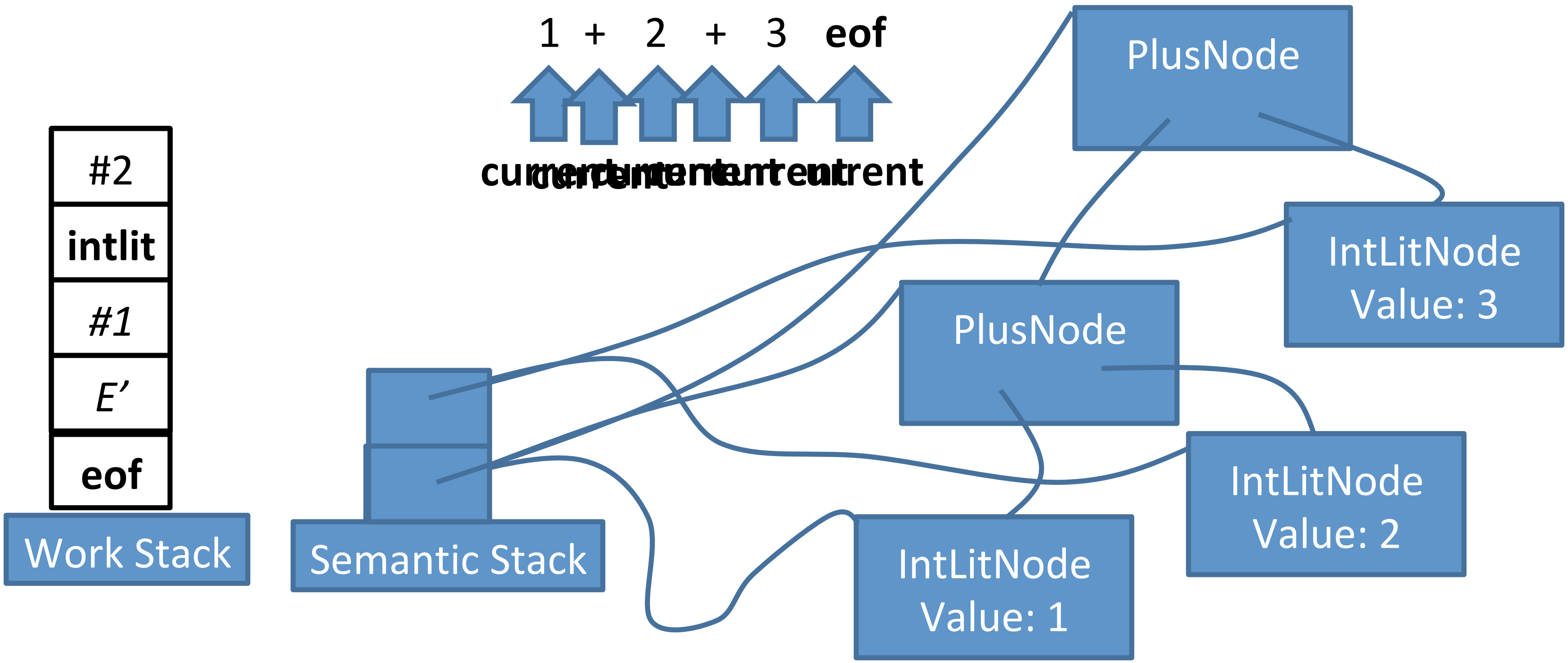
E ⟶ *T E'*
E' ⟶ **+** *T* #1 *E'*
| ε
T ⟶ #2 **intlit**

"AST" SDT Actions

```
#1 tTrans = pop ;
   eTrans = pop ;
   push(new PlusNode(tTrans, eTrans))

#2 push(new IntLitNode(intlit.value))
```

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



1 + 2 + 3 **eof**

current current current current current current

#2
intlit
#1
E'
eof
Work Stack

Semantic Stack

PlusNode

PlusNode

IntLitNode
Value: 3

IntLitNode
Value: 2

IntLitNode
Value: 1

We now have an AST

At this point, we have completed the frontend for (a) compiler

- Only recognize LL(1)

LL(1) is not a great class of languages

```
if (e1)
   stmt1
if (e2)
    stmt2
else
   stmt3
```

Grammar Snippet

IfStmt -> **if lparens** Exp **rparens** *Stmts*
          | **if lparens** Exp **rparens** *Stmts* **else** *Stmts*