CS536

SDT For Top-Down Parsing

Announcement: Midterm Prep

- List of topics
  - Up to and including all of last week
- Length 1hr 10min
- No extra materials allowed – just bring a pen
- Sample midterm
  - Recommended that you do this by Tuesday
  - We'll review it in class

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
- If the grammar is not LL(1) we can attempt a transformation sequence:
  1. Remove left recursion
  2. Left-factoring

Today

- Review Parse Table Construction
  - 2 examples
- Show how to do Syntax-Directed Translation using an LL(1) parser

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(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)

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

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** }

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

FOLLOW (B) = { **c**, **eof** }

FOLLOW (D) = { **a**, **c** }

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

$$S \rightarrow \mathbf{(}\ S\ \mathbf{)} \mid \mathbf{\{}\ S\ \mathbf{\}} \mid \boldsymbol{\varepsilon}$$

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 $X \longrightarrow \alpha$
 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

$$S \rightarrow \mathbf{+}\, S \mid \boldsymbol{\varepsilon}$$

FIRST (*S*) = { **+**, **ε** }

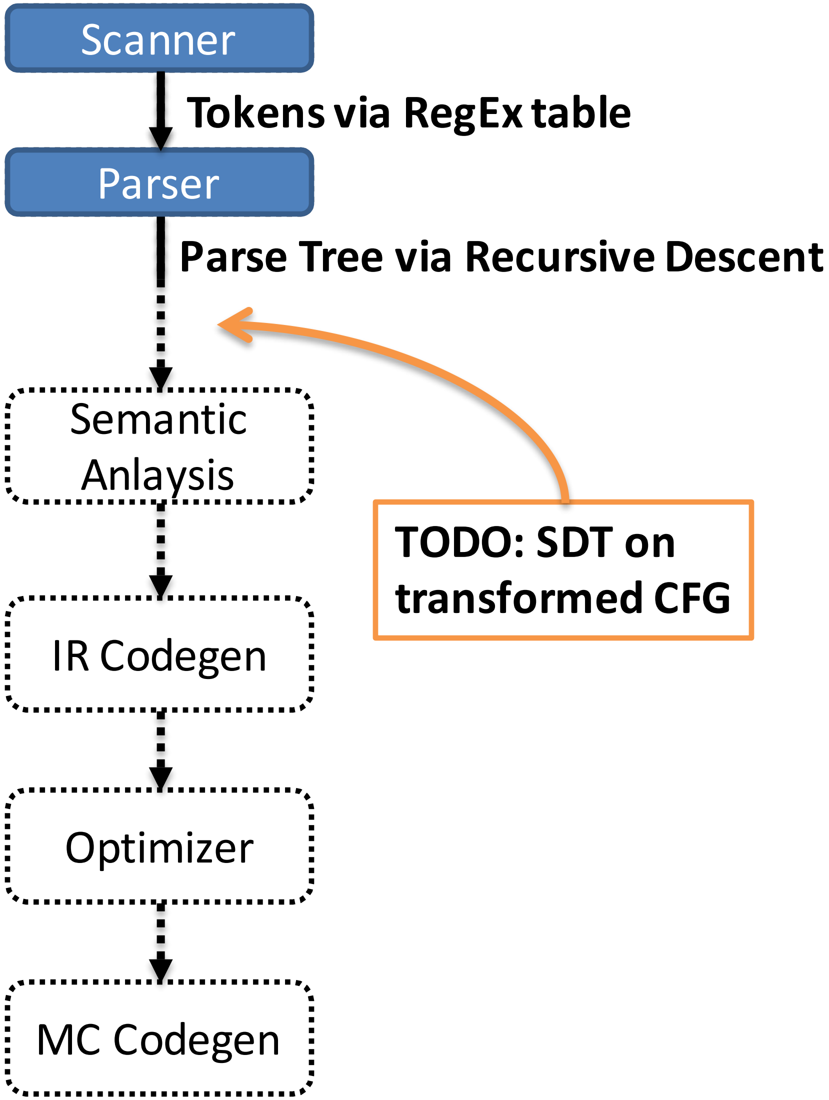
FIRST (**+** *S*) = { **+** }

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

Semantic Stack

- Instead of building the parse tree, give parser second, *semantic* stack
  - Holds nonterminals’ translations
- SDT rules converted to SDT actions on semantic stack
  - Pop translations of RHS nonterms off
  - Push computed translation of LHS nonterm on

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

CFG

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

SDT Actions

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

Action Numbers: Example 1

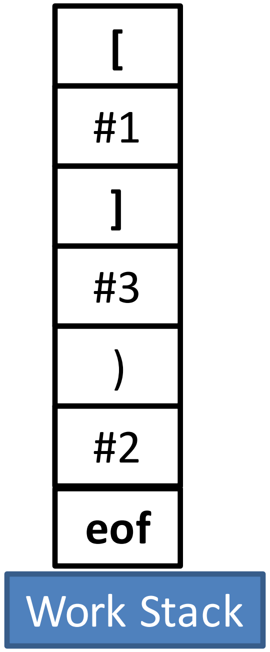
CFG

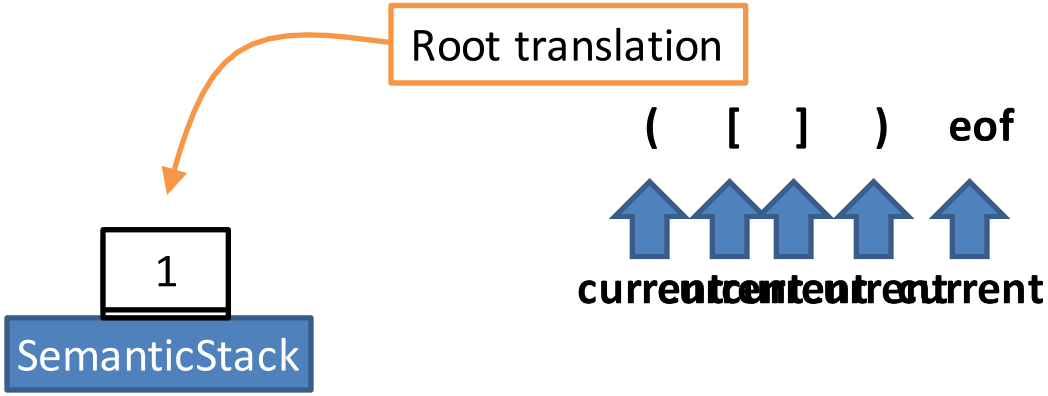
Expr ⟶ ε #1
| **(** *Expr* **)** #2
| **[** *Expr* **]** #3

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 nonterminal, 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 tTrans = pop; eTrans = pop ; push(tTrans * eTrans)
#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

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

SDT Actions

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

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

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

Example: SDT on Transformed Grammar

CFG

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

SDT Actions

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

What about ASTs?

- Push and pop nodes 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

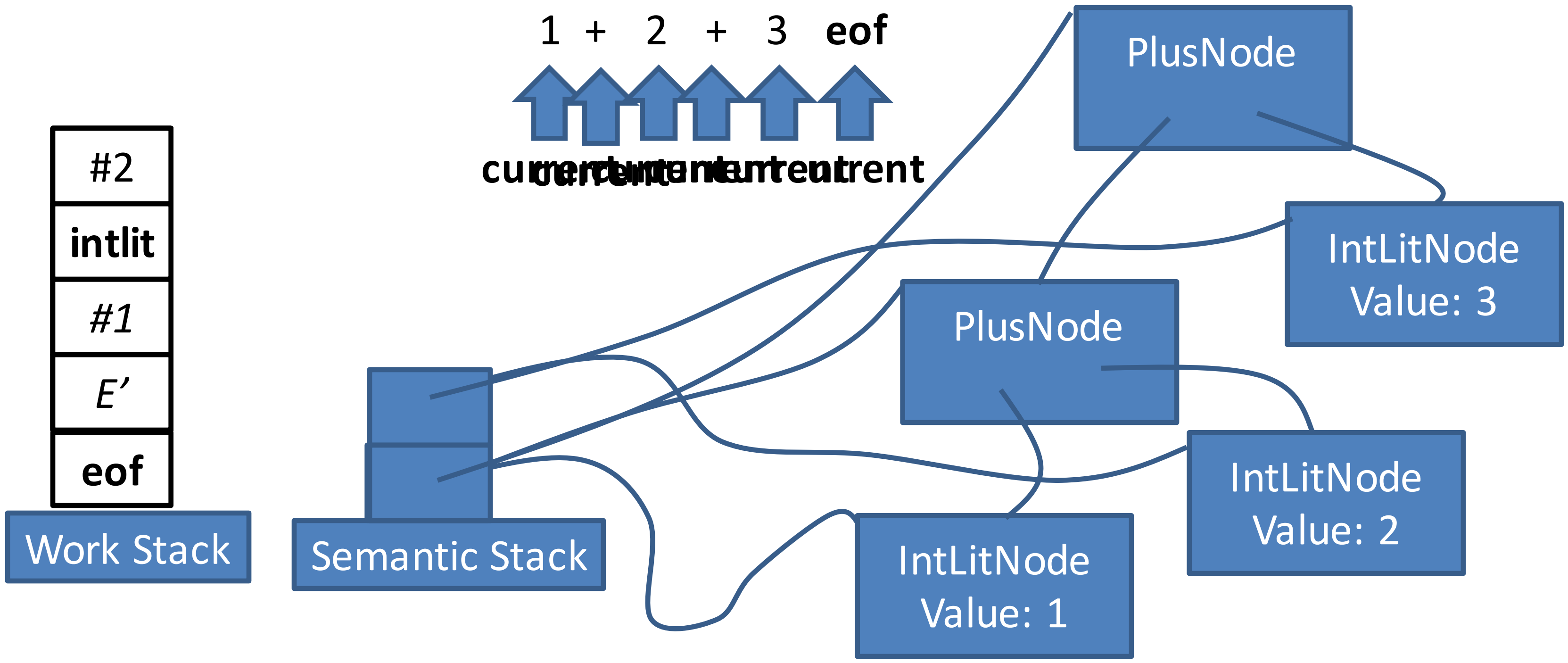
Transformed CFG

$E \longrightarrow T\ E'$
$E' \longrightarrow$ **+** T #1 E'
$\quad |\ \ \varepsilon$
$T \longrightarrow$ #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*