KU | Fall 2018 | Drew Davidson

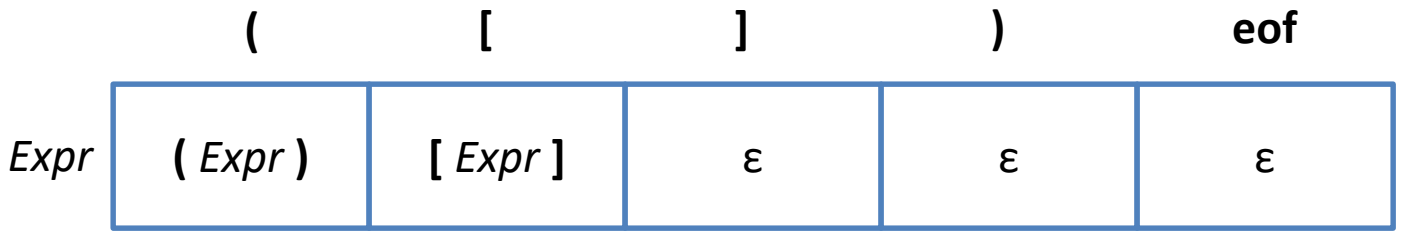
EECS 665 COMPILER CONSTRUCTION

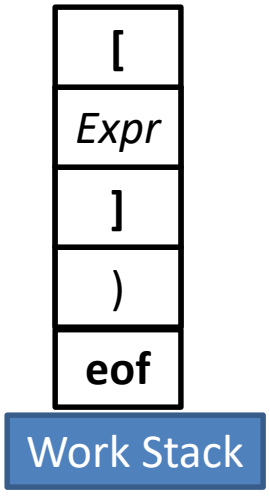
17 – Bottom-Up SDT

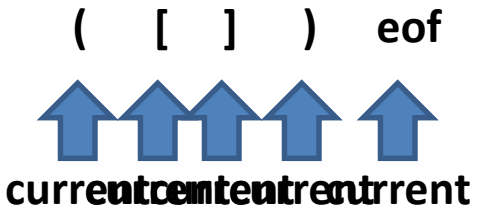
Warmup

CFG

Expr ⟶ ε
| **(** *Expr* **)**
| **[** *Expr* **]**







Lesson Plan

- Last Time:
  - SDT in practice
    - Implementation details
- This Time:
  - A bit of admin
  - SDT for top-down parsers

Quiz 2

Quiz 2: The Good

- Lots of perfect scores
  - I probably won't go much fast than we are going now
  - I would like to try to find a wa to challenge everyone
- "More material since Quiz 1 than all [of my last class]"
  - This pleases me

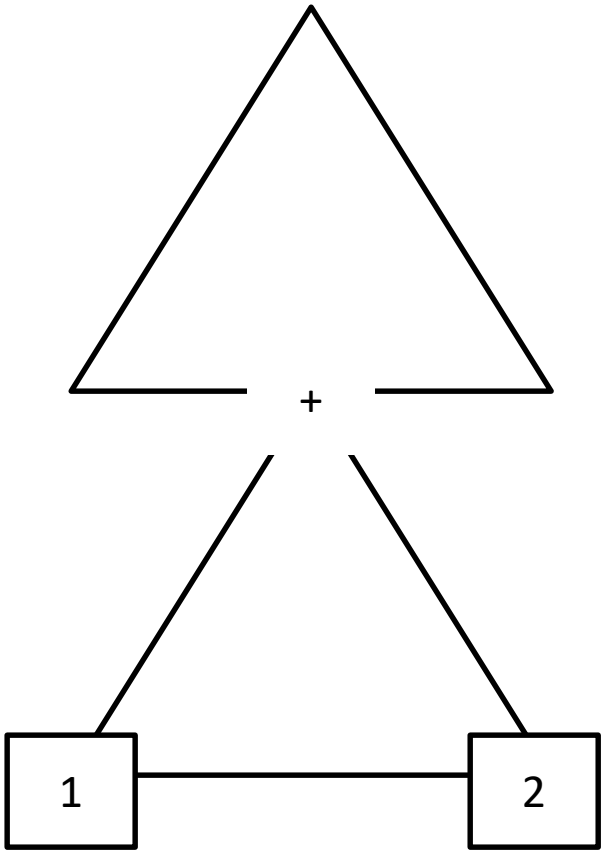
Quiz 2: The Bad & The Ugly

- Lots harder than Quiz 1
- A big drop in letter grades for some students
  - Philosophy: for the course to be *good* it can't be *trivial*
- Already wrote a long email about this
- One policy takeaway: you ***can*** replace your lowest quiz score with your final exam score

Bottom-Up SDT

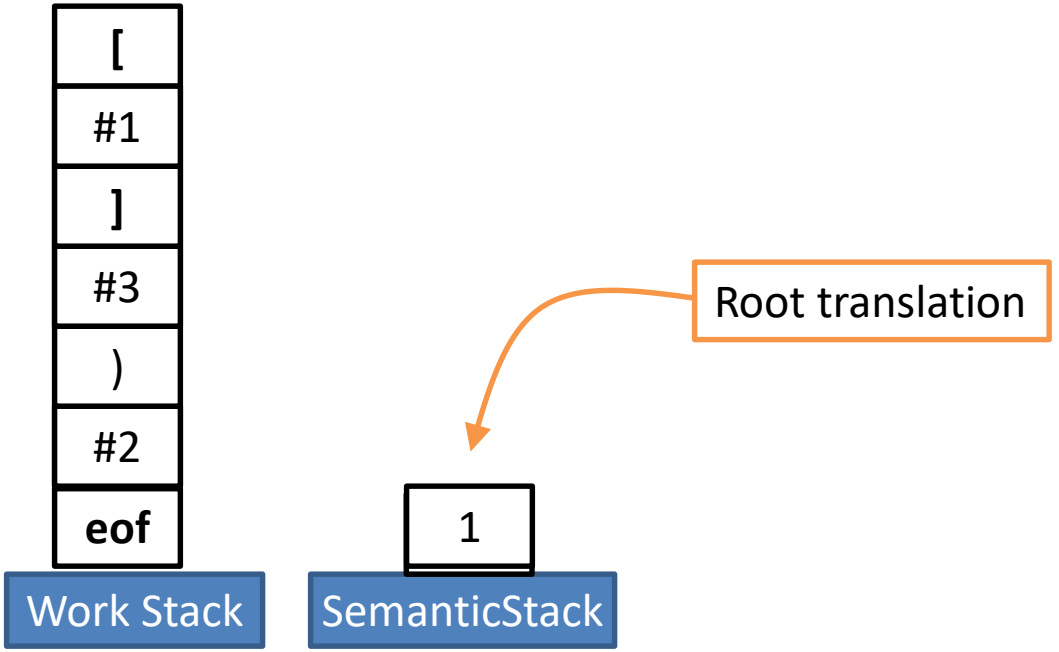
Synthesized attributes?

- So far, SDT shown as second (bottom-up) pass over parse tree
- The LL(1) parser never needed to explicitly built the parse tree (implicitly tracked via stack)



Semantic Stack

- Instead of building the parse tree, give parser second, *semantic* stack
  - Holds nonterminals' translations



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

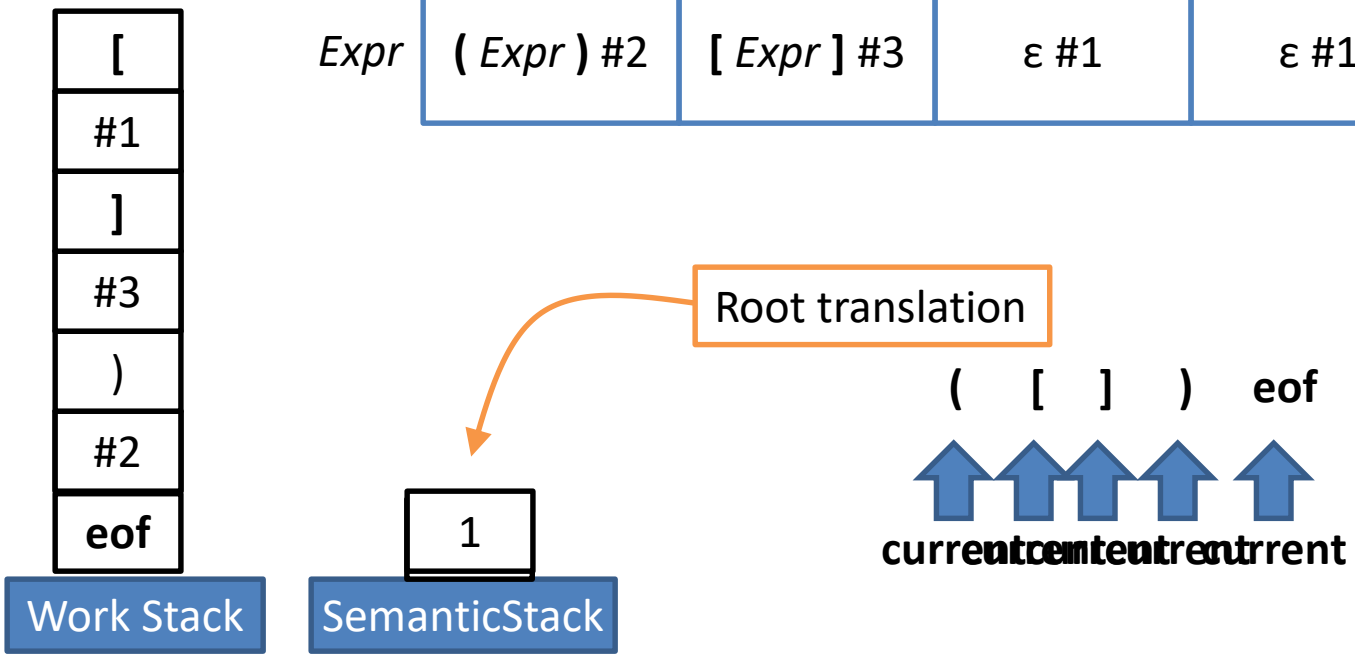
CFG

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 <u>after</u> their corresponding nonterminal, <u>before</u> their corresponding terminal
- Translations popped right to left in action

<u>CFG</u>

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

<u>SDT Actions</u>

#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)

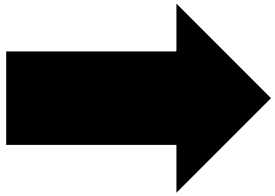
#4 push(**intlit**.value)

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)

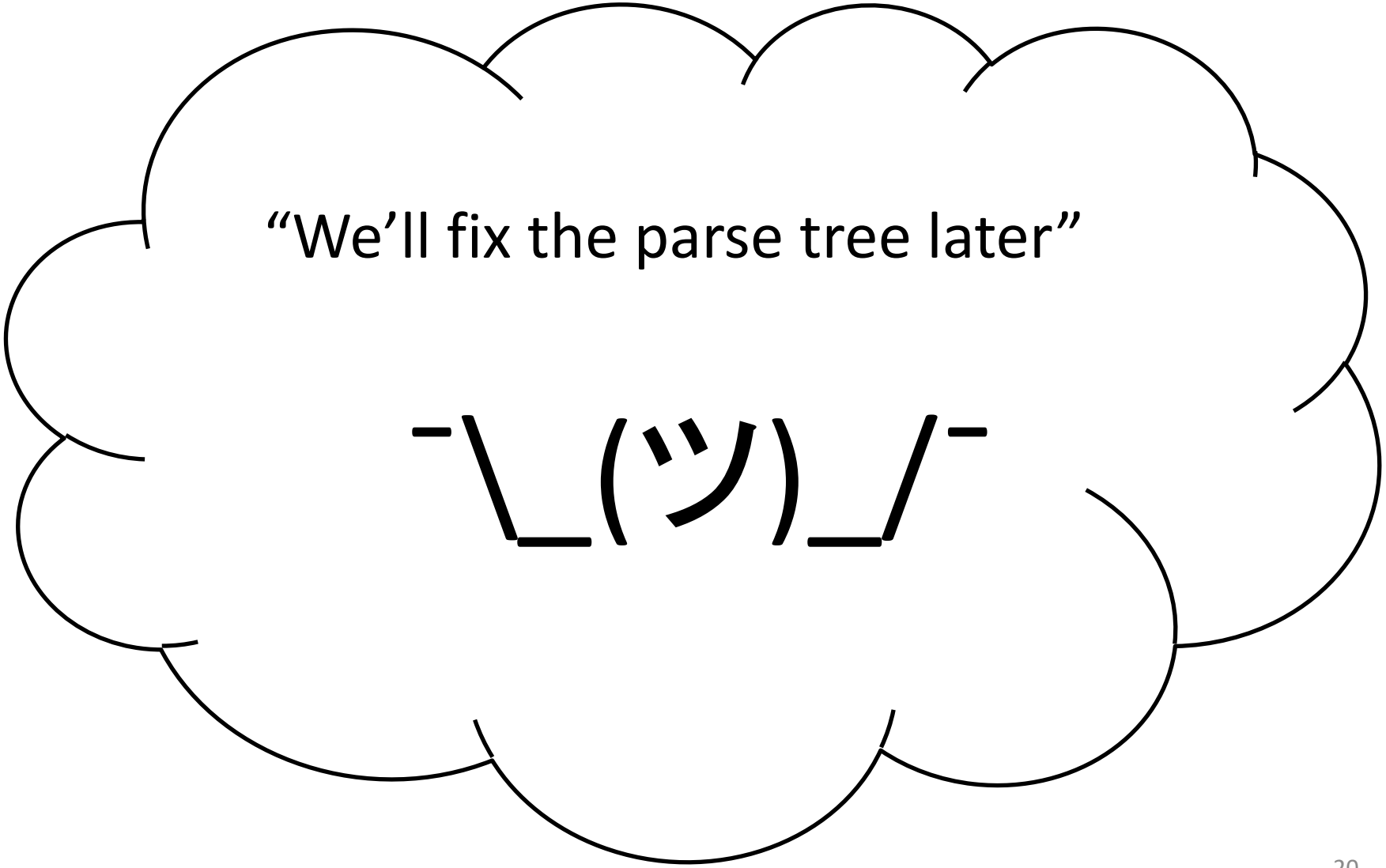
#2 push(**intlit**.value)

"AST" SDT Actions

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

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

Needn’t worry about wonky parse tree



AST Example

Transformed CFG

$E \longrightarrow T\ E'$

$E' \longrightarrow +\ T\ \#1\ E'$

$\quad\quad\ |\ \ \varepsilon$

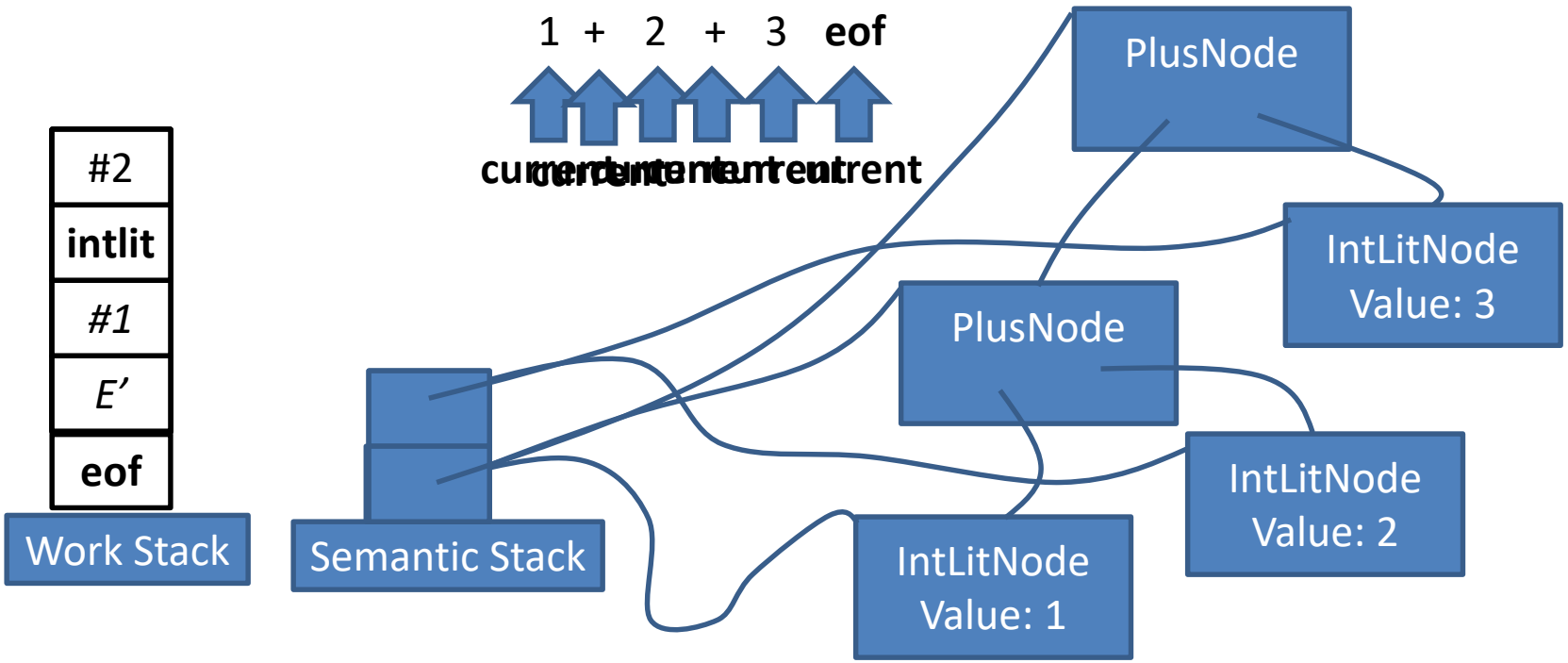
$T \longrightarrow \#2$ **intlit**

"AST" SDT Actions

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

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

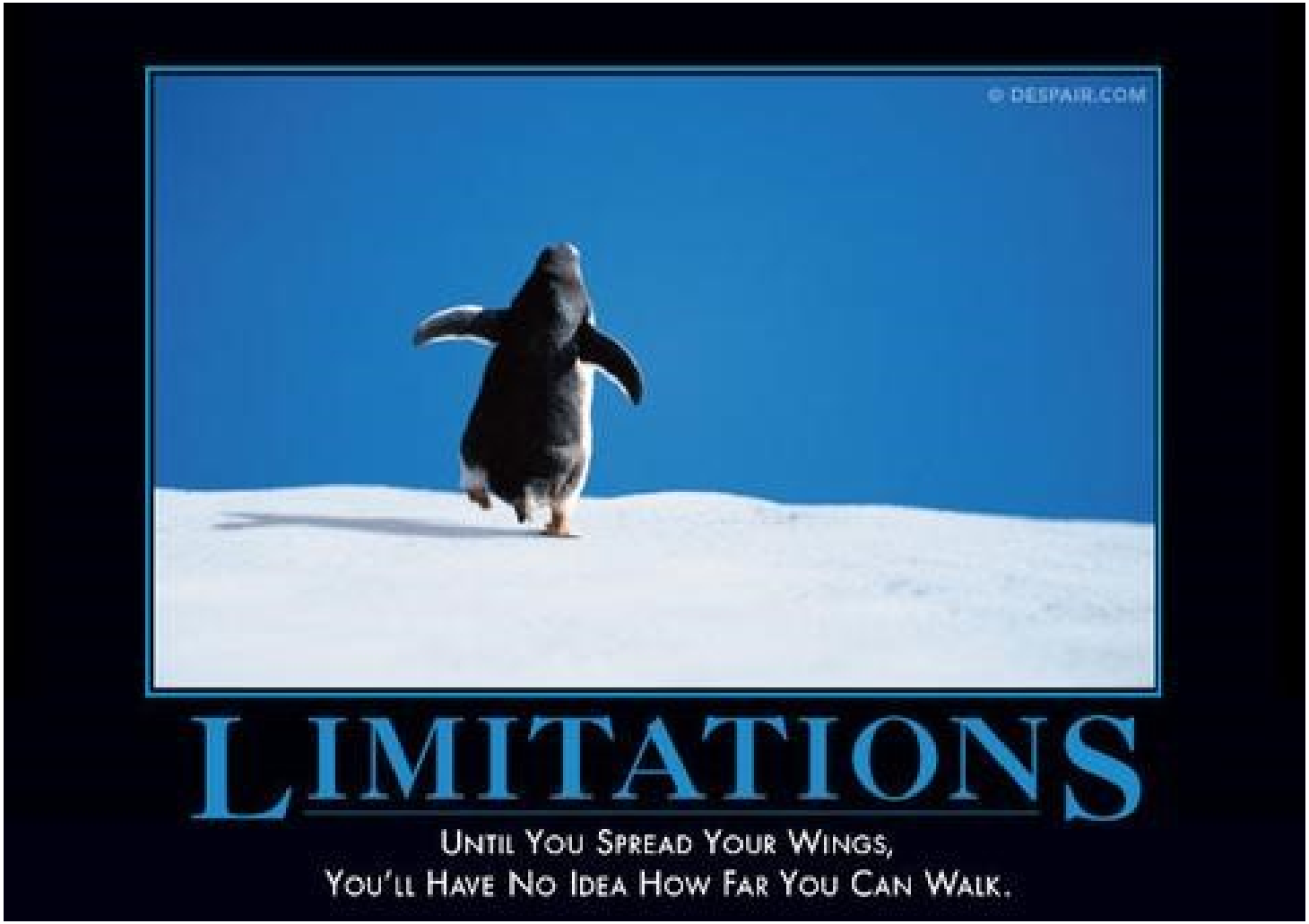
	intlit	**+**	**EOF**
E	*T E'*		
E'		*+ T #1 E'*	ε
T	**#2 intlit**		



We can build ASTs!

- We have the tools for syntax definitions
  - Formalisms
    - LR/LL
  - Software
    - Flex/Bison
    - ANTLR (?)

Limitations of our SDT Formalisms



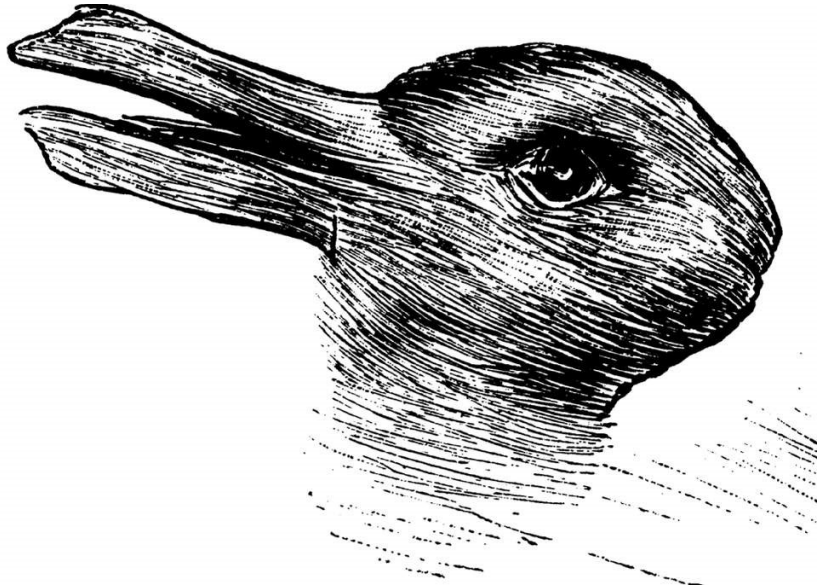
(One of) C's Dirty Secret(s)

The grammar is ambiguous!

C's Ambiguity

Grammar Snippet

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



Duck or Rabbit?

The Dangling Else Problem

Grammar Snippet

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

```
if e1 then if e2 then s else s2
```

=?

```
if e1 then (if e2 then s) else s2
```

or

```
if e1 then (if e2 then s else s2)
```

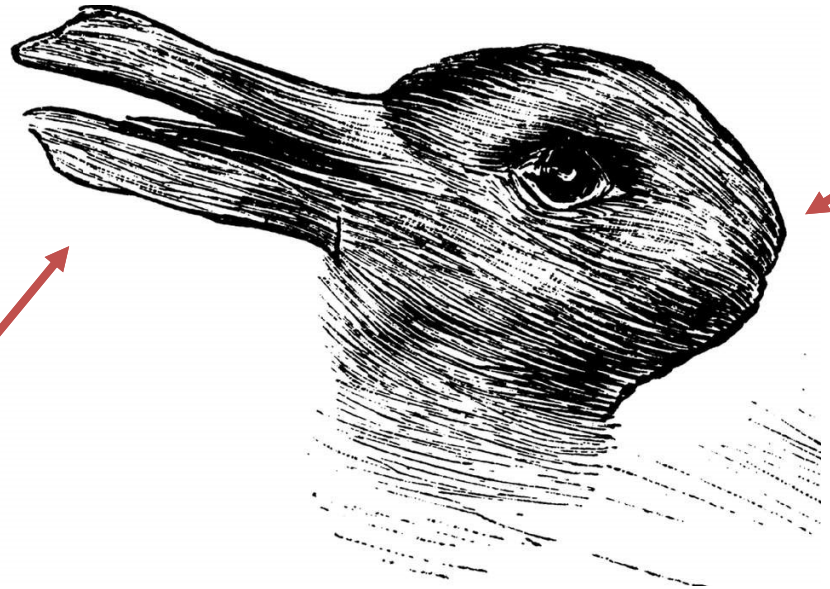
Dangling Else – What do now?

- Our formalisms seem too weak to cover basic programming problems
- We'll just fudge it
  - Just augment the rules to specify the right action

Just Define a Convention

Grammar Snippet

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



~~Duck or Rabbit?~~

A rabbit

The Dangling Else Problem

Grammar Snippet

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

```
if e1 then if e2 then s else s2
```

=?

This one

```
if e1 then (if e2 then s) else s2
```

or

```
if e1 then (if e2 then s else s2)
```

How Implementations Handle Dangling Else?

- By “accident”
- LL(1)
  - Table collision
    - Just keep the right entry
- LR(1)
  - Shift reduce conflict
    - Just favor shift

The Dangling Else Problem

Grammar Snippet

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

```
if e1 then if e2 then s else s2
```

The problem of *Language Intent*:

```
if e1 then (if e2 then s) else s2
if e1 then (if e2 then s else s2)
```

The problem of *Language Parsing*:

Can LL(1) handle this?

Can SLR/LR handle this?