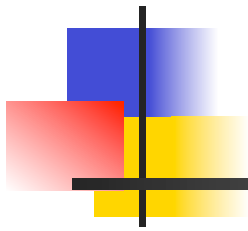


# Programming Languages and Compilers (CS 421)



Elsa L Gunter

2112 SC, UIUC

<http://courses.engr.illinois.edu/cs421>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



# Disambiguating a Grammar

---

- $\langle \text{exp} \rangle ::= 0 | 1 | b \langle \text{exp} \rangle | \langle \text{exp} \rangle a$
- $| \langle \text{exp} \rangle m \langle \text{exp} \rangle$
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.



# Disambiguating a Grammar

---

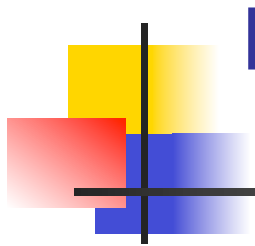
- $\langle \text{exp} \rangle ::= 0 | 1 | b \langle \text{exp} \rangle | \langle \text{exp} \rangle a$
- $| \langle \text{exp} \rangle m \langle \text{exp} \rangle$
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.
  
- $\langle \text{exp} \rangle ::= \langle \text{no } m \rangle ::= \langle \text{no } b \rangle ::=$
- $\langle \text{exp} \rangle m \langle \text{no } m \rangle | \langle \text{no } m \rangle b \langle \text{no } m \rangle | \langle \text{no } b \rangle \langle \text{no } b \rangle a | 0 | 1$



# LR Parsing

---

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

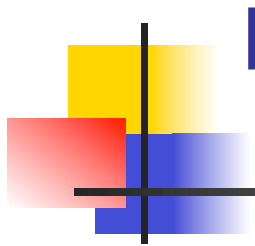


Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$$= \bullet (0 + 1) + 0 \quad \text{shift}$$

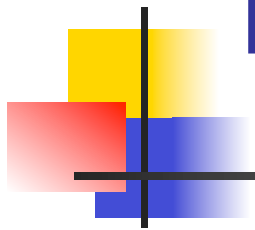


Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$$\begin{aligned} &= (\bullet 0 + 1) + 0 && \text{shift} \\ &= \bullet (0 + 1) + 0 && \text{shift} \end{aligned}$$



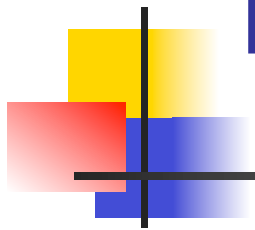
Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$$\begin{aligned} &\Rightarrow (0 \bullet + 1) + 0 \\ &= (\bullet 0 + 1) + 0 \\ &= \bullet (0 + 1) + 0 \end{aligned}$$

reduce  
shift  
shift



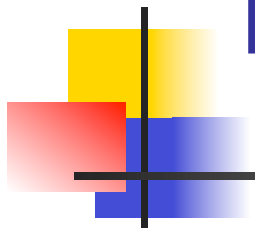
Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$= (\langle \text{Sum} \rangle \bullet + 1) + 0$	shift
$\Rightarrow (0 \bullet + 1) + 0$	reduce
$= (\bullet 0 + 1) + 0$	shift
$= \bullet (0 + 1) + 0$	shift



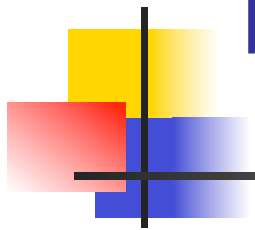


Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$= (\langle \text{Sum} \rangle + \bullet 1) + 0$	shift
$= (\langle \text{Sum} \rangle \bullet + 1) + 0$	shift
$\Rightarrow (0 \bullet + 1) + 0$	reduce
$= (\bullet 0 + 1) + 0$	shift
$= \bullet (0 + 1) + 0$	shift



Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

---

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$	reduce
$= (\langle \text{Sum} \rangle + \bullet 1) + 0$	shift
$= (\langle \text{Sum} \rangle \bullet + 1) + 0$	shift
$\Rightarrow (0 \bullet + 1) + 0$	reduce
$= (\bullet 0 + 1) + 0$	shift
$= \bullet (0 + 1) + 0$	shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$  reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$  reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$  shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$  shift  
 $\Rightarrow (0 \bullet + 1) + 0$  reduce  
 $= (\bullet 0 + 1) + 0$  shift  
 $= \bullet (0 + 1) + 0$  shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= (\langle \text{Sum} \rangle \bullet) + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$  reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$  reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$  shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$  shift  
 $\Rightarrow (0 \bullet + 1) + 0$  reduce  
 $= (\bullet 0 + 1) + 0$  shift  
 $= \bullet (0 + 1) + 0$  shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$       reduce  
 $= (\langle \text{Sum} \rangle \bullet) + 0$       shift  
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$       reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$       reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$       shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$       shift  
 $\Rightarrow (0 \bullet + 1) + 0$       reduce  
 $= (\bullet 0 + 1) + 0$       shift  
 $= \bullet (0 + 1) + 0$       shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= \langle \text{Sum} \rangle \bullet + 0$  shift  
 $\Rightarrow ( \langle \text{Sum} \rangle ) \bullet + 0$  reduce  
 $= ( \langle \text{Sum} \rangle \bullet ) + 0$  shift  
 $\Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet ) + 0$  reduce  
 $\Rightarrow ( \langle \text{Sum} \rangle + 1 \bullet ) + 0$  reduce  
 $= ( \langle \text{Sum} \rangle + \bullet 1 ) + 0$  shift  
 $= ( \langle \text{Sum} \rangle \bullet + 1 ) + 0$  shift  
 $\Rightarrow ( 0 \bullet + 1 ) + 0$  reduce  
 $= ( \bullet 0 + 1 ) + 0$  shift  
 $= \bullet ( 0 + 1 ) + 0$  shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

$= \langle \text{Sum} \rangle + \bullet 0$  shift  
 $= \langle \text{Sum} \rangle \bullet + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$  reduce  
 $= (\langle \text{Sum} \rangle \bullet) + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$  reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$  reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$  shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$  shift  
 $\Rightarrow (0 \bullet + 1) + 0$  reduce  
 $= (\bullet 0 + 1) + 0$  shift  
 $= \bullet (0 + 1) + 0$  shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$   
 $\Rightarrow \langle \text{Sum} \rangle + 0 \bullet$  reduce  
 $= \langle \text{Sum} \rangle + \bullet 0$  shift  
 $= \langle \text{Sum} \rangle \bullet + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$  reduce  
 $= (\langle \text{Sum} \rangle \bullet) + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$  reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$  reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$  shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$  shift  
 $\Rightarrow (0 \bullet + 1) + 0$  reduce  
 $= (\bullet 0 + 1) + 0$  shift  
 $= \bullet (0 + 1) + 0$  shift

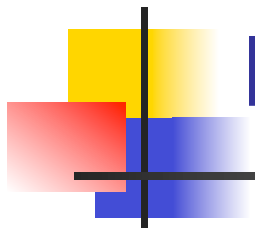


Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle$	$\Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$	●	reduce
	$\Rightarrow \langle \text{Sum} \rangle + 0$	●	reduce
	$= \langle \text{Sum} \rangle +$	● 0	shift
	$= \langle \text{Sum} \rangle$	● + 0	shift
	$\Rightarrow (\langle \text{Sum} \rangle)$	● + 0	reduce
	$= (\langle \text{Sum} \rangle$	●) + 0	shift
	$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$	●) + 0	reduce
	$\Rightarrow (\langle \text{Sum} \rangle + 1$	●) + 0	reduce
	$= (\langle \text{Sum} \rangle +$	● 1) + 0	shift
	$= (\langle \text{Sum} \rangle$	● + 1) + 0	shift
	$\Rightarrow (0$	● + 1) + 0	reduce
	$= ($	● 0 + 1) + 0	shift
	$=$	● ( 0 + 1 ) + 0	shift

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \bullet \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet$  reduce  
 $\Rightarrow \langle \text{Sum} \rangle + 0 \bullet$  reduce  
 $= \langle \text{Sum} \rangle + \bullet 0$  shift  
 $= \langle \text{Sum} \rangle \bullet + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$  reduce  
 $= (\langle \text{Sum} \rangle \bullet) + 0$  shift  
 $\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$  reduce  
 $\Rightarrow (\langle \text{Sum} \rangle + 1 \bullet) + 0$  reduce  
 $= (\langle \text{Sum} \rangle + \bullet 1) + 0$  shift  
 $= (\langle \text{Sum} \rangle \bullet + 1) + 0$  shift  
 $\Rightarrow (0 \bullet + 1) + 0$  reduce  
 $= (\bullet 0 + 1) + 0$  shift  
 $= \bullet (0 + 1) + 0$  shift



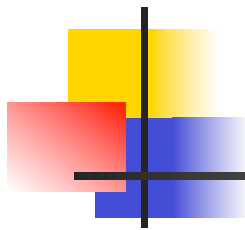
# Example

---

$$( 0 + 1 ) + 0$$



11/1/16



# Example

---

$$( \quad 0 \quad + \quad 1 \quad ) \quad + \quad 0$$





# Example

---

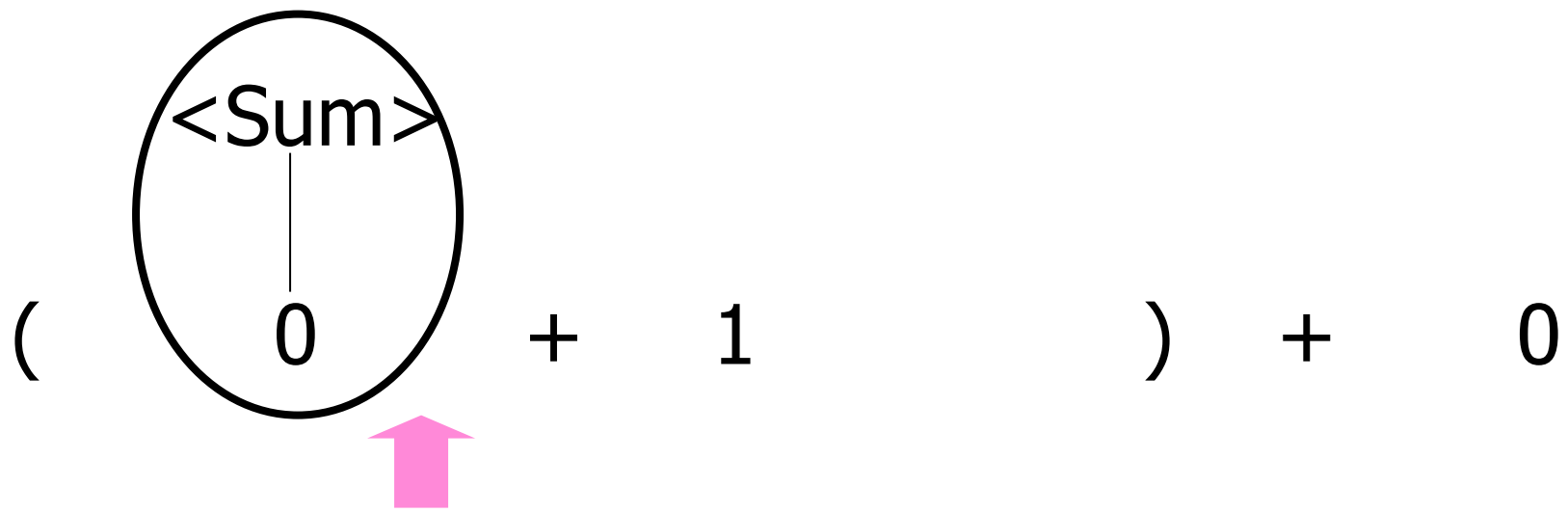
$$( 0 + 1 ) + 0$$





# Example

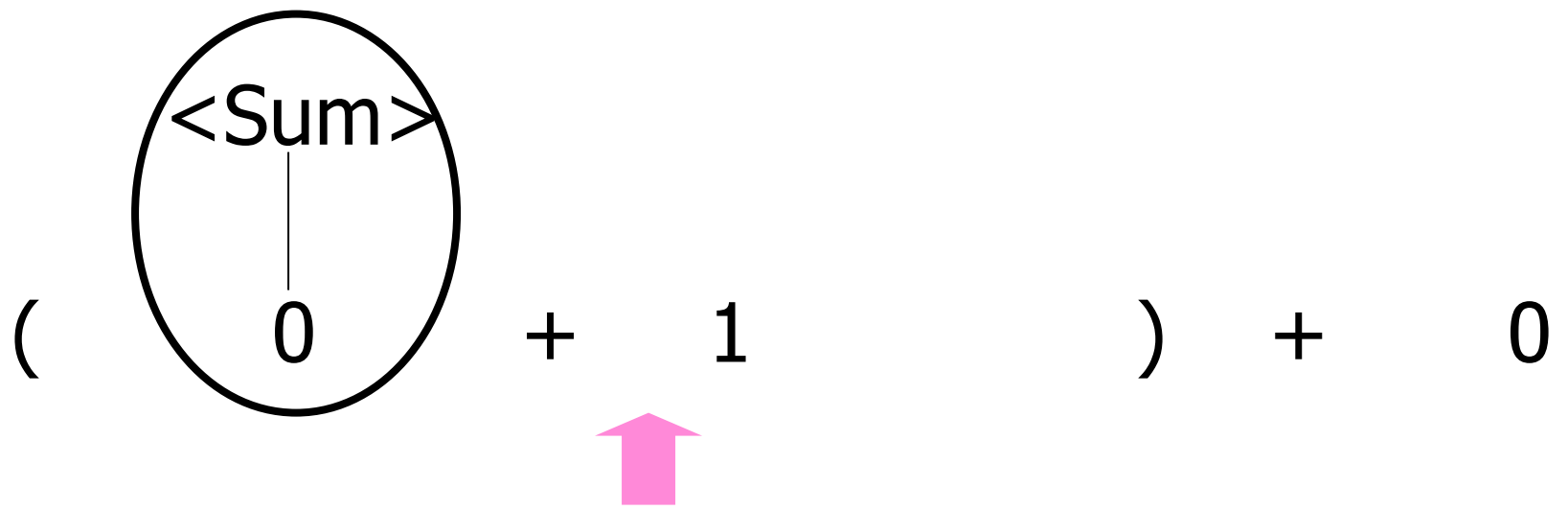
---





# Example

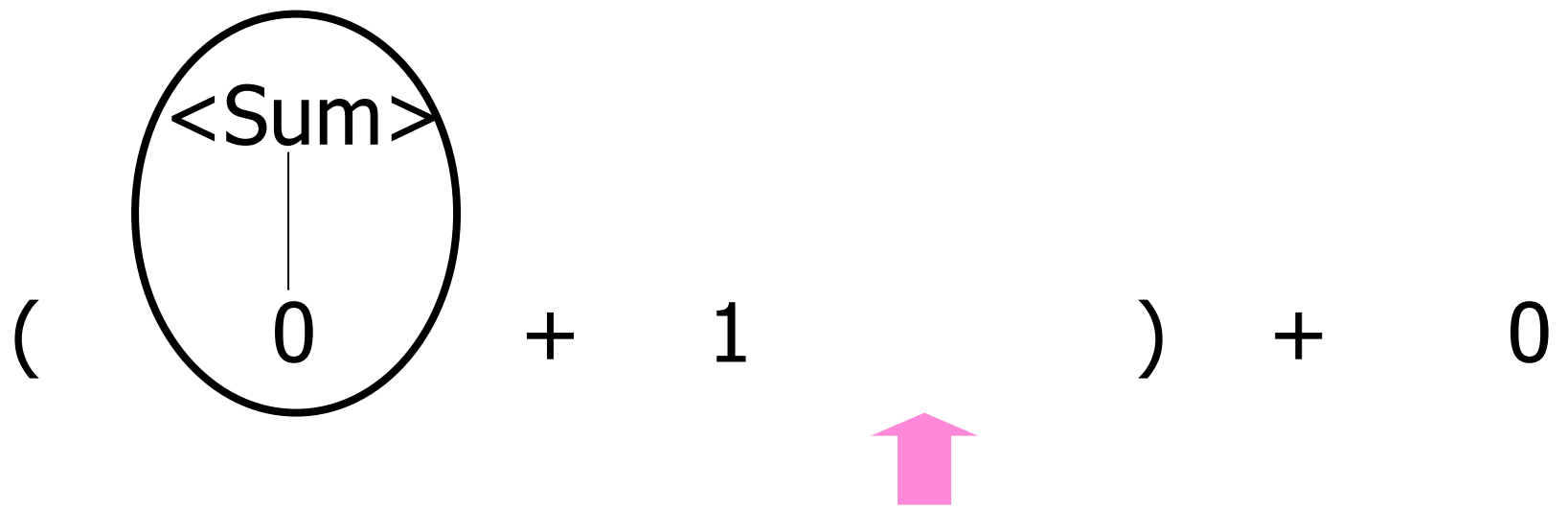
---





# Example

---

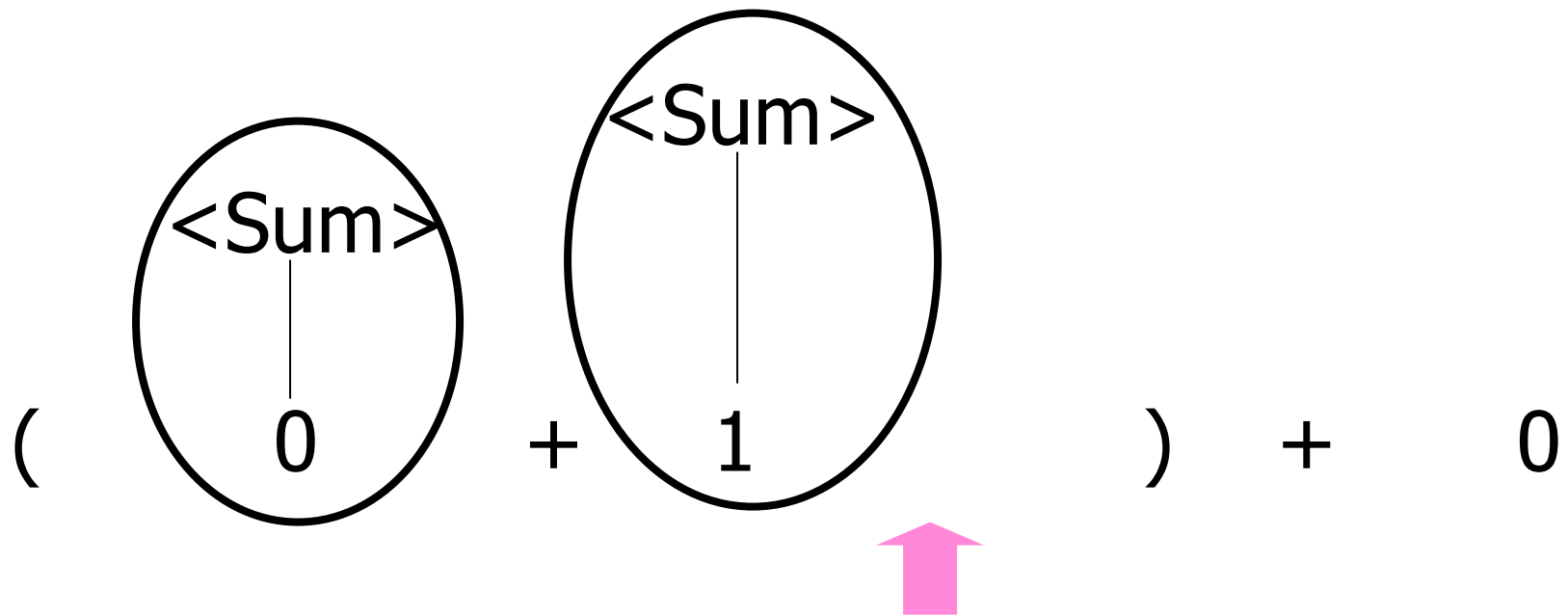






# Example

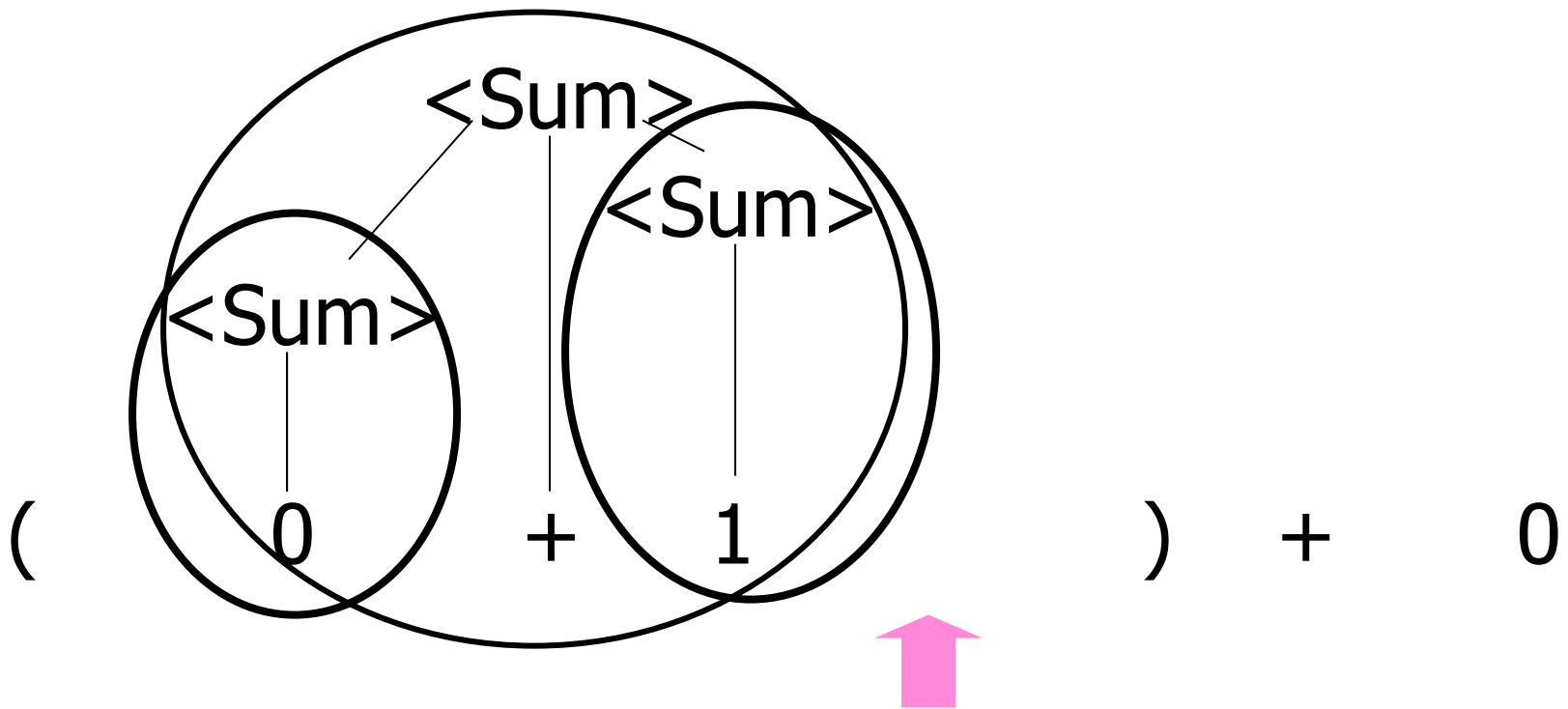
---





# Example

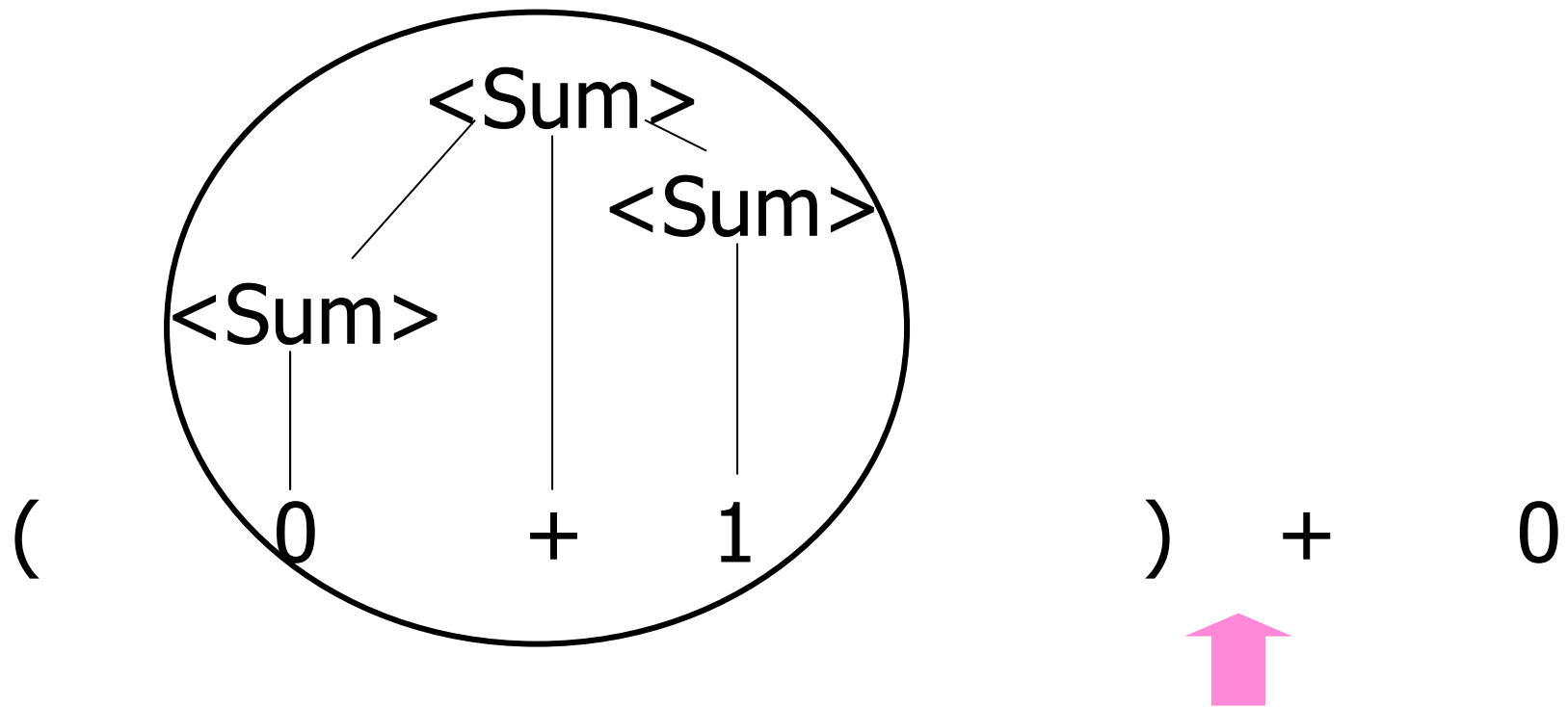
---

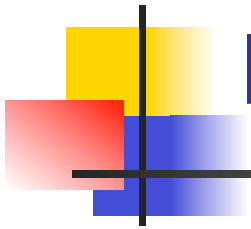




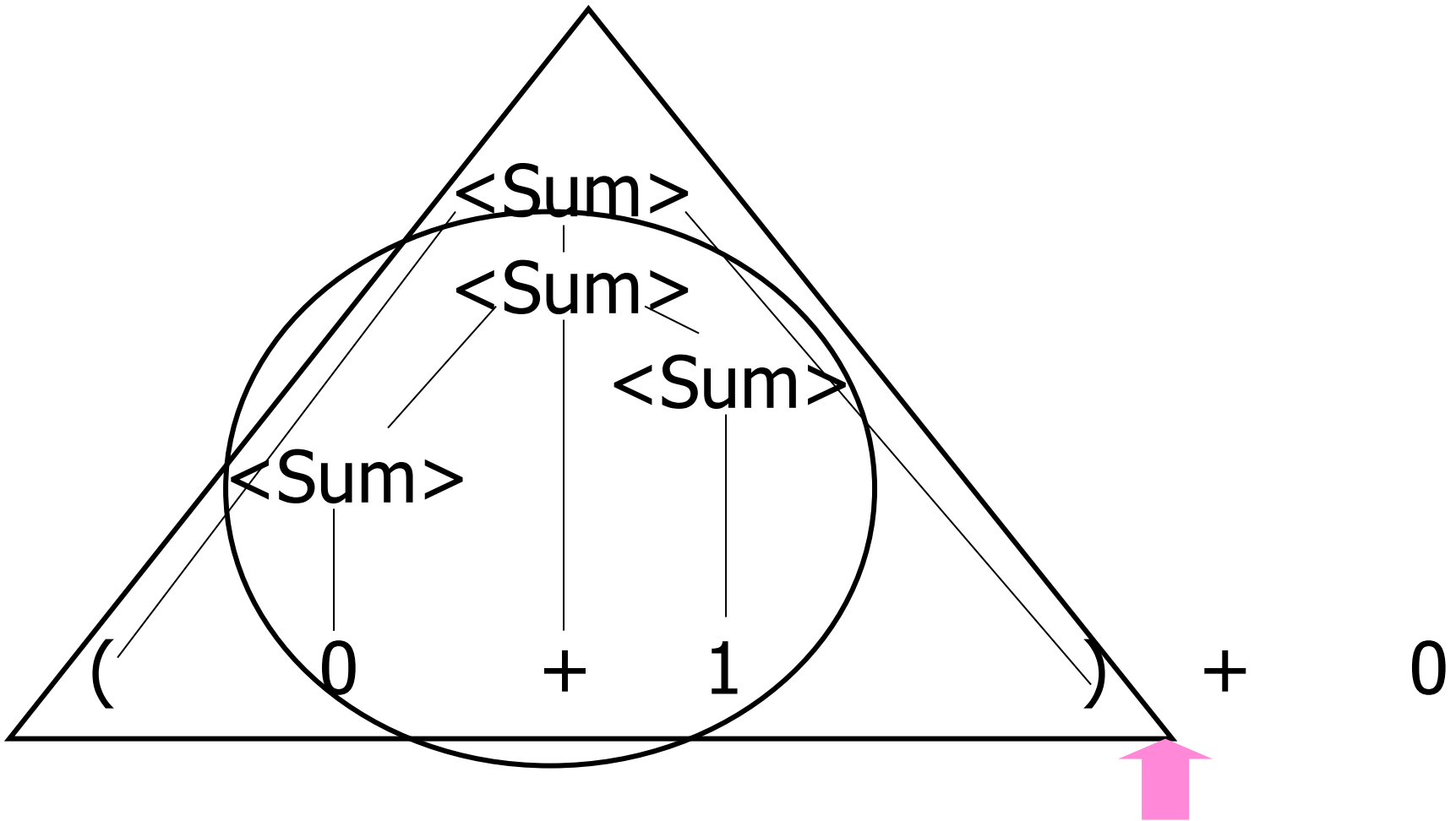
# Example

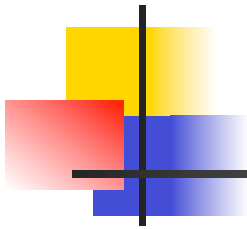
---



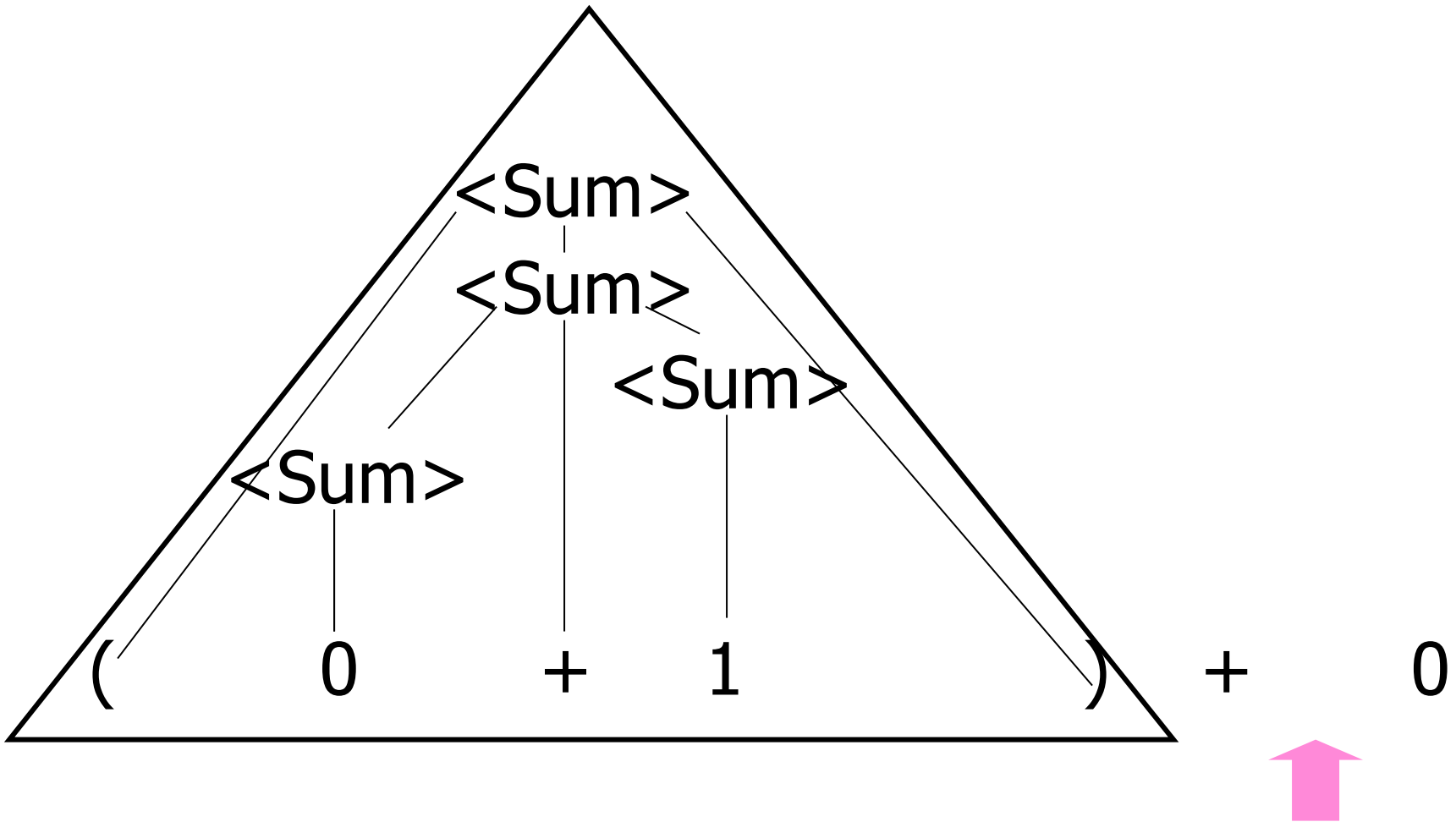


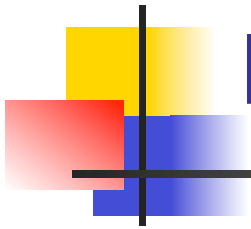
# Example



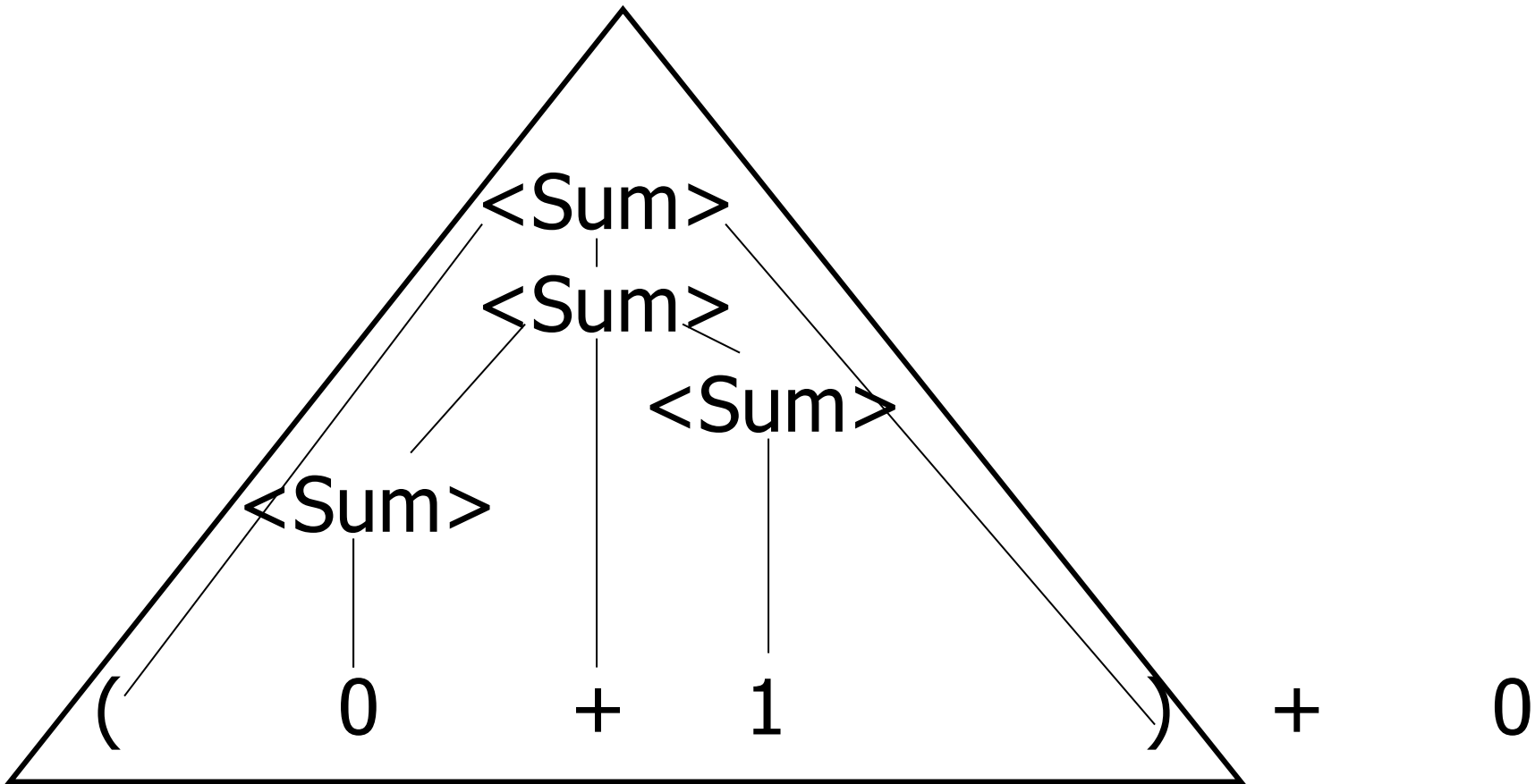


# Example





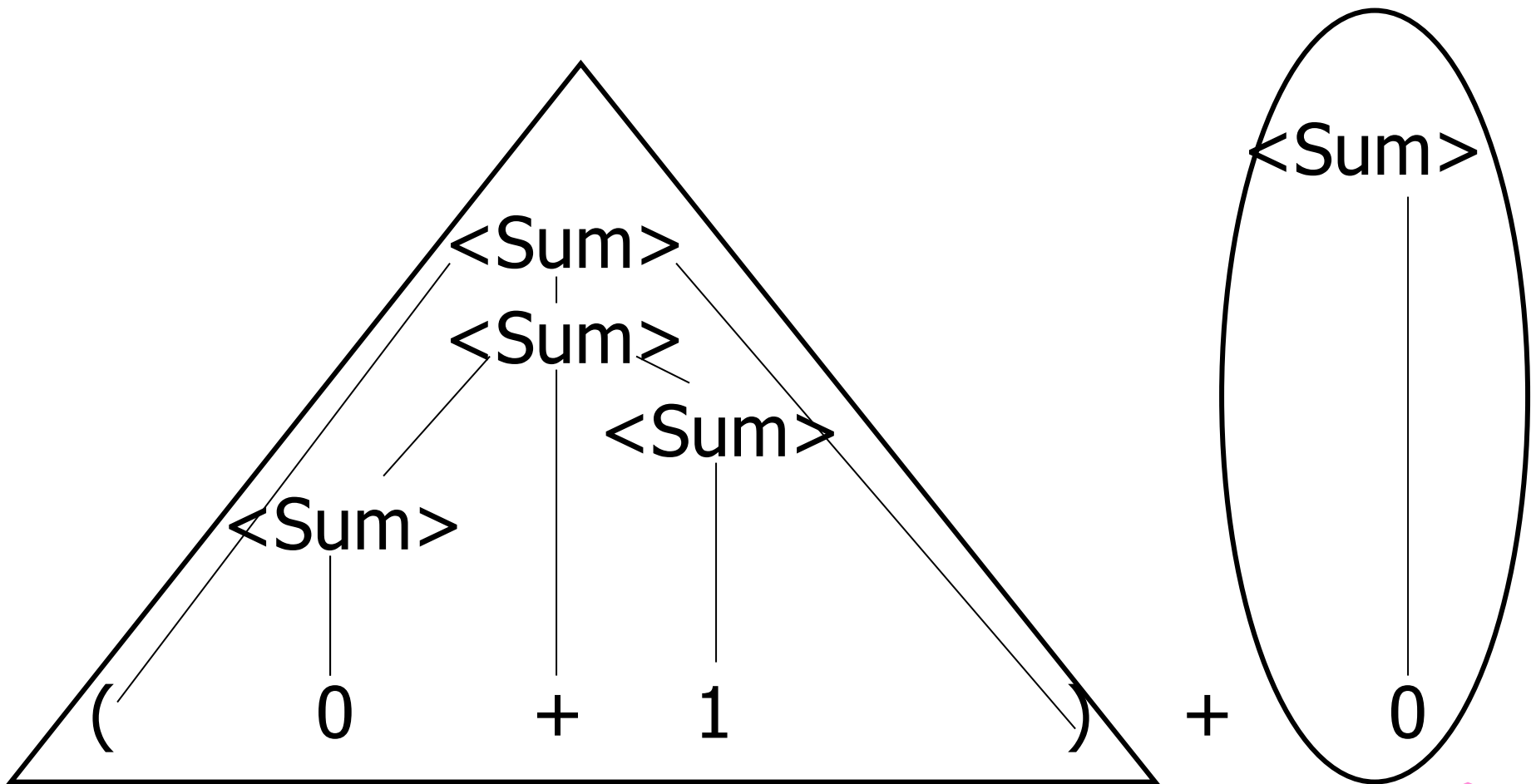
# Example

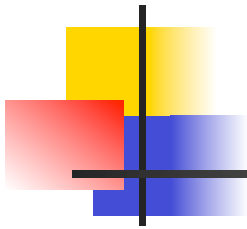




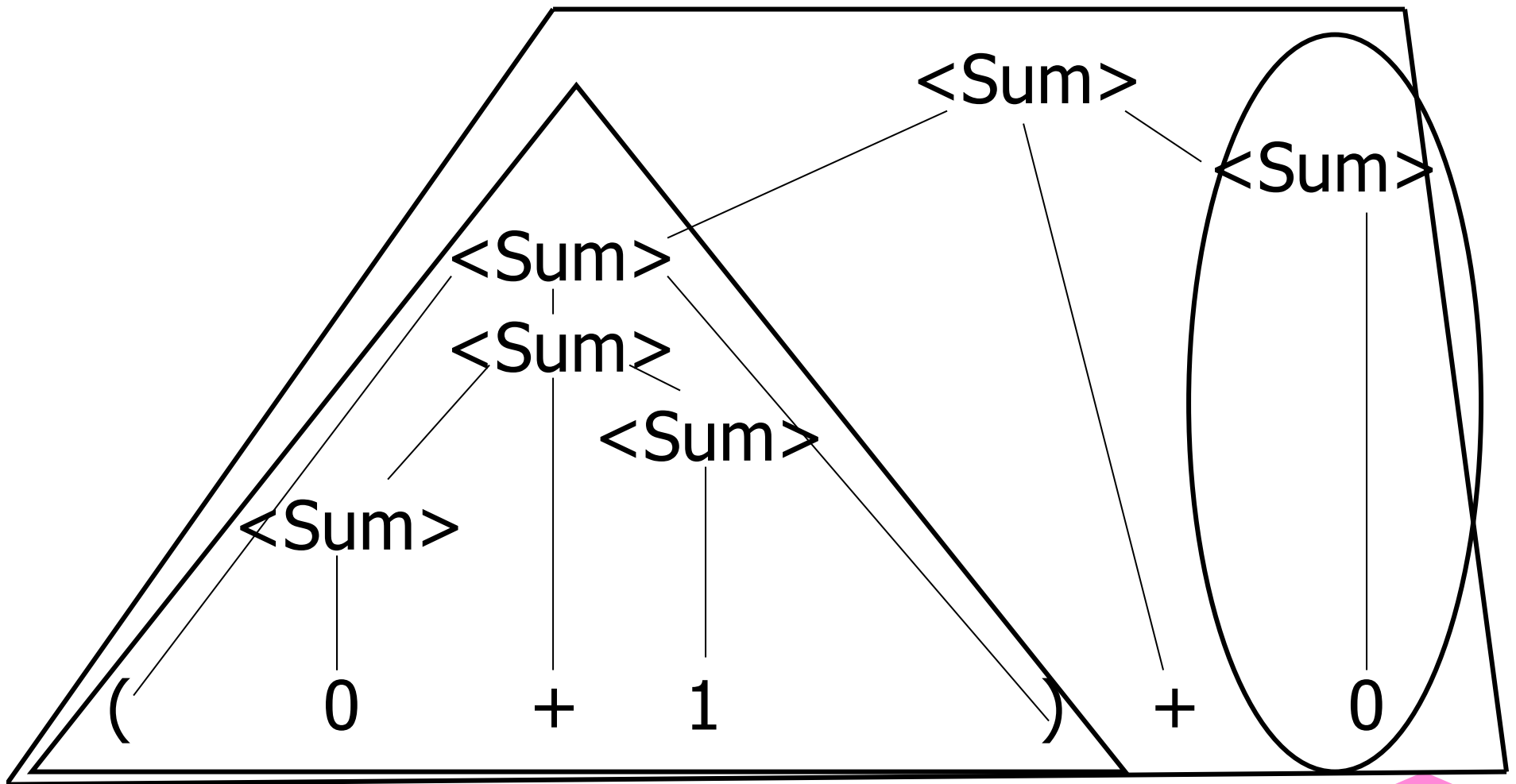
# Example

---





# Example

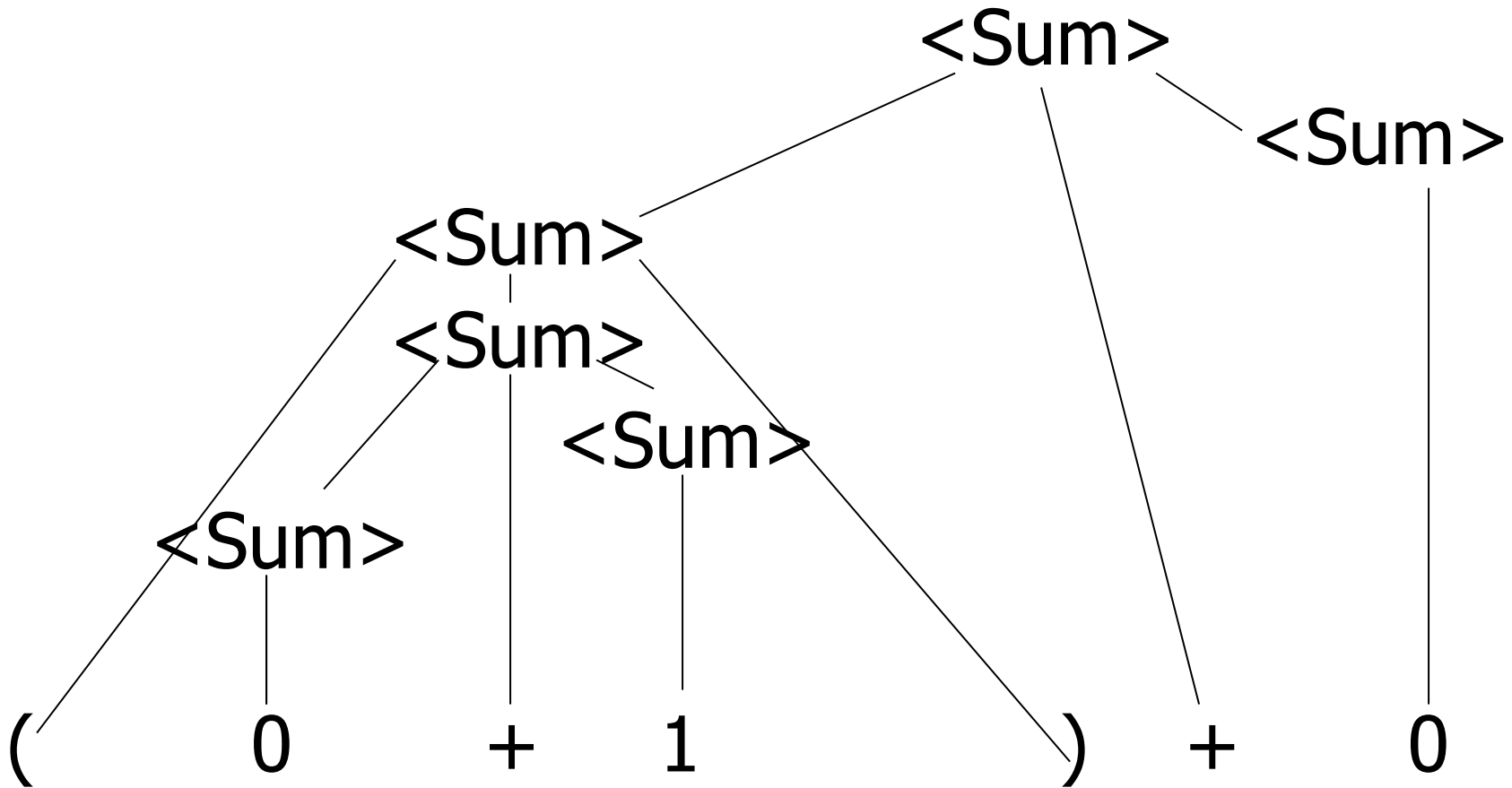






# Example

---





# LR Parsing Tables

---

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals



# Action and Goto Tables

---

- Given a state and the next input, Action table says either
  - **shift** and go to state  $n$ , or
  - **reduce** by production  $k$  (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state  $m$



# LR(i) Parsing Algorithm

---

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals



# LR(i) Parsing Algorithm

---

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push **state**(1) onto stack
- 3. Look at next  $i$  tokens from token stream ( $toks$ ) (don't remove yet)
4. If top symbol on stack is **state**( $n$ ), look up action in Action table at  $(n, toks)$



## LR(i) Parsing Algorithm

---

5. If action = **shift**  $m$ ,
- a) Remove the top token from token stream and push it onto the stack
  - b) Push **state**( $m$ ) onto stack
  - c) Go to step 3



## LR(i) Parsing Algorithm

---

6. If action = **reduce**  $k$  where production  $k$  is  
 $E ::= u$
- a) Remove  $2 * \text{length}(u)$  symbols from stack ( $u$  and all the interleaved states)
  - b) If new top symbol on stack is **state**( $m$ ), look up new state  $p$  in  $\text{Goto}(m, E)$
  - c) Push  $E$  onto the stack, then push **state**( $p$ ) onto the stack
  - d) Go to step 3



# LR(i) Parsing Algorithm

---

7. If action = **accept**

- Stop parsing, return success

8. If action = **error**,

- Stop parsing, return failure





# Adding Synthesized Attributes

---

- Add to each **reduce** a rule for calculating the new synthesized attribute from the component attributes
- Add to each non-terminal pushed onto the stack, the attribute calculated for it
- When performing a **reduce**,
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack



# Shift-Reduce Conflicts

---

- **Problem:** can't decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar

Example:  $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$   
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\bullet 0 + 1 + 0$                       shift  
 $\rightarrow 0 \bullet + 1 + 0$                       reduce  
 $\rightarrow \langle \text{Sum} \rangle \bullet + 1 + 0$                       shift  
 $\rightarrow \langle \text{Sum} \rangle + \bullet 1 + 0$                       shift  
 $\rightarrow \langle \text{Sum} \rangle + 1 \bullet + 0$                       reduce  
 $\rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet + 0$



## Example - cont

---

- **Problem:** shift or reduce?
- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce
- Shift first - right associative
- Reduce first- left associative



# Reduce - Reduce Conflicts

---

- **Problem:** can't decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom:** RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors



## Example

---

■  $S ::= A \mid aB$      $A ::= abc$      $B ::= bc$

● abc                    shift

a ● bc                    shift

ab ● c                    shift

abc ●

■ Problem: reduce by  $B ::= bc$  then by  $S ::= aB$ , or by  $A ::= abc$  then  $S ::= A$ ?



# Recursive Descent Parsing

---

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars
- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation)



# Recursive Descent Parsing

---

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate
- Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram





# Recursive Descent Parsing

---

- Each subprogram must be able to decide how to begin parsing by looking at the left-most character in the string to be parsed
  - May do so directly, or indirectly by calling another parsing subprogram
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
  - Sometimes can modify grammar to suit



# Sample Grammar

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle$   
 $\mid \langle \text{term} \rangle - \langle \text{expr} \rangle$

$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle$   
 $\mid \langle \text{factor} \rangle / \langle \text{term} \rangle$

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid ( \langle \text{expr} \rangle )$



# Tokens as OCaml Types

---

- + - \* / ( ) <id>

- Becomes an OCaml datatype

type token =

  Id\_token of string

  | Left\_parenthesis | Right\_parenthesis

  | Times\_token | Divide\_token

  | Plus\_token | Minus\_token



# Parse Trees as Datatypes

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle$   
 $\mid \langle \text{term} \rangle - \langle \text{expr} \rangle$

type expr =

Term\_as\_Expr of term

| Plus\_Expr of (term \* expr)

| Minus\_Expr of (term \* expr)



# Parse Trees as Datatypes

---

$$\begin{aligned} \langle \text{term} \rangle ::= & \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \\ & \langle \text{term} \rangle \\ & \mid \langle \text{factor} \rangle / \langle \text{term} \rangle \end{aligned}$$

and term =

- Factor\_as\_Term of factor
- | Mult\_Term of (factor \* term)
- | Div\_Term of (factor \* term)



# Parse Trees as Datatypes

---

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid ( \langle \text{expr} \rangle )$

and factor =

Id\_as\_Factor of string

| Parenthesized\_Expr\_as\_Factor of expr



# Parsing Lists of Tokens

---

- Will create three mutually recursive functions:
  - $\text{expr} : \text{token list} \rightarrow (\text{expr} * \text{token list})$
  - $\text{term} : \text{token list} \rightarrow (\text{term} * \text{token list})$
  - $\text{factor} : \text{token list} \rightarrow (\text{factor} * \text{token list})$
- Each parses what it can and gives back parse and remaining tokens



# Parsing an Expression

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [ ( + | - ) \langle \text{expr} \rangle ]$

let rec expr tokens =

(match term tokens

with ( term\_parse , tokens\_after\_term) ->

(match tokens\_after\_term

with( Plus\_token :: tokens\_after\_plus) ->





# Parsing an Expression

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [ ( + | - ) \langle \text{expr} \rangle ]$

let rec expr tokens =

(match **term tokens**

with ( term\_parse , tokens\_after\_term) ->

(match tokens\_after\_term

with ( Plus\_token :: tokens\_after\_plus) ->



# Parsing a Plus Expression

---

`<expr> ::= <term> [ ( + | - ) <expr> ]`

let rec expr tokens =

(match term tokens

with ( **term\_parse** , tokens\_after\_term) ->

(match tokens\_after\_term

with ( Plus\_token :: tokens\_after\_plus) ->

# Parsing a Plus Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \underline{[( + | - ) \langle \text{expr} \rangle ]}$

let rec expr tokens =

(match term tokens

with ( **term\_parse** , tokens\_after\_term) ->

(match **tokens\_after\_term**

with ( Plus\_token :: tokens\_after\_plus) ->



# Parsing a Plus Expression

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [ ( + | - ) \langle \text{expr} \rangle ]$

let rec expr tokens =

(match term tokens

with ( term\_parse , tokens\_after\_term) ->

(match tokens\_after\_term

with ( **Plus\_token** :: tokens\_after\_plus) ->



# Parsing a Plus Expression

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle$

(match **expr tokens\_after\_plus**  
with ( expr\_parse , tokens\_after\_expr) ->  
( Plus\_Expr ( term\_parse , expr\_parse ),  
tokens\_after\_expr))



# Parsing a Plus Expression

---

`<expr> ::= <term> + <expr>`

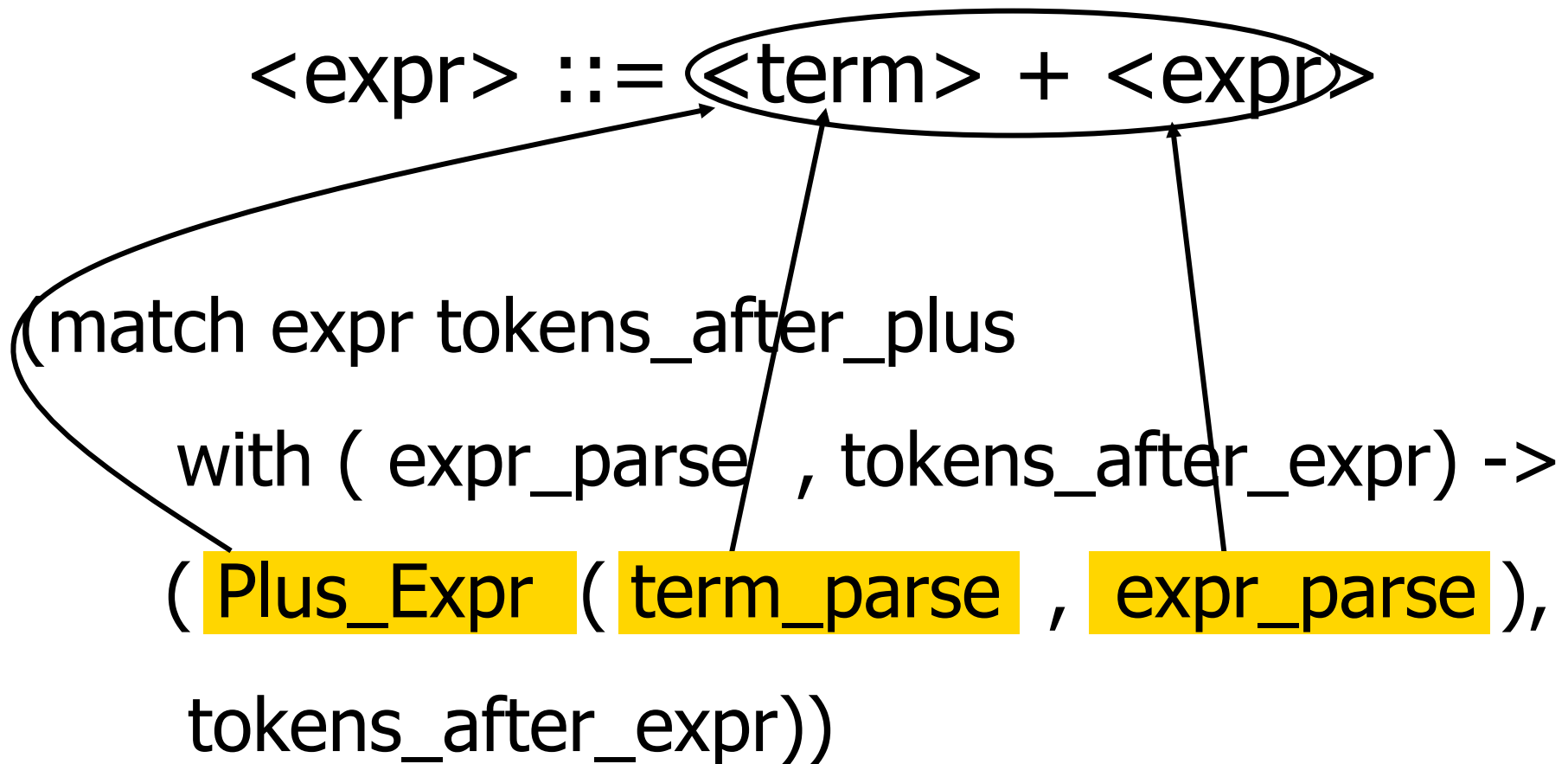
(match expr tokens\_after\_plus

with ( **expr\_parse** , tokens\_after\_expr) ->

( Plus\_Expr ( term\_parse , expr\_parse ),

tokens\_after\_expr))

# Building Plus Expression Parse Tree





# Parsing a Minus Expression

---

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle - \langle \text{expr} \rangle$

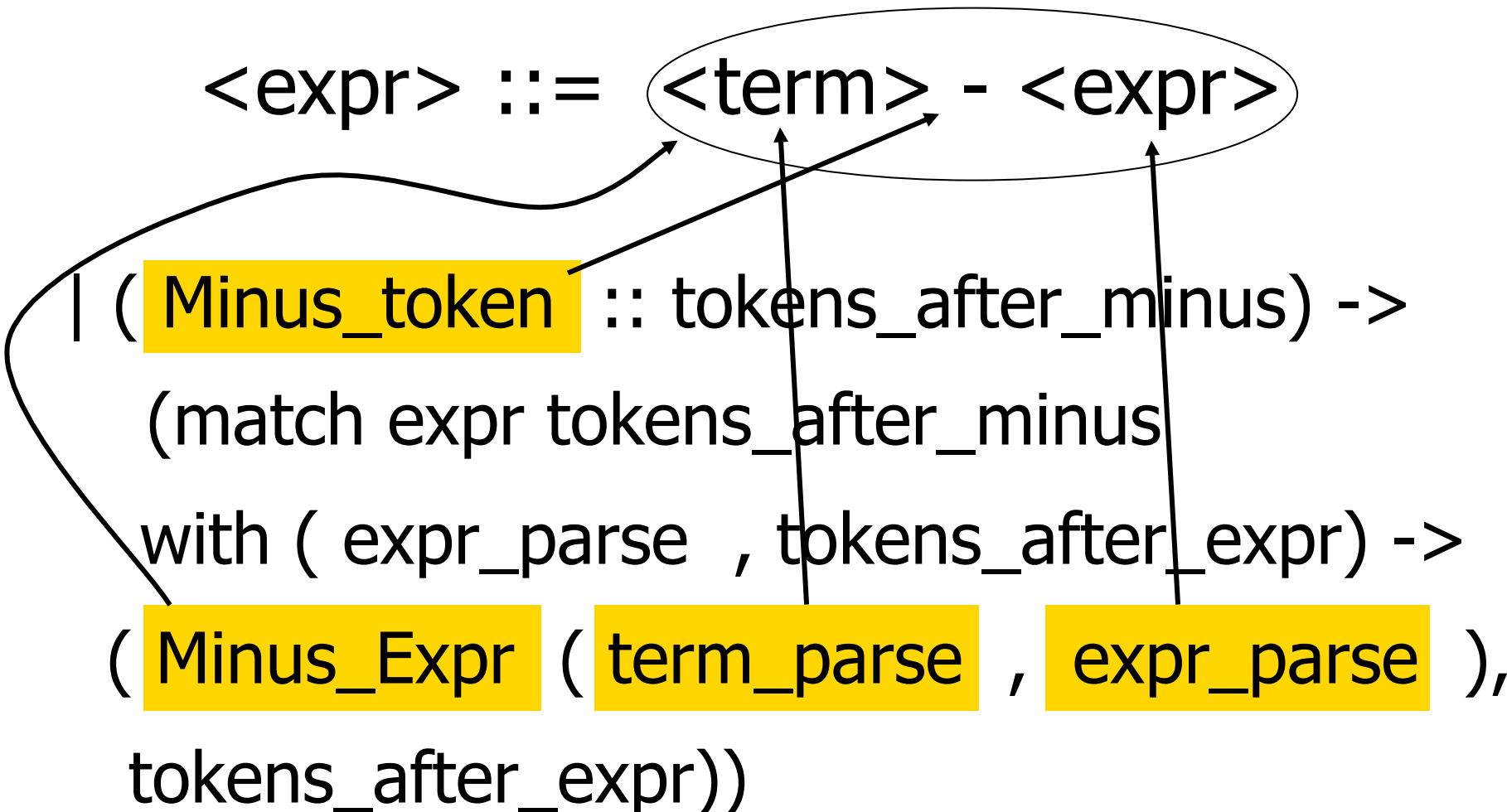
```
| ( Minus_token :: tokens_after_minus) ->  
  (match expr tokens_after_minus  
   with ( expr_parse , tokens_after_expr) ->  
        ( Minus_Expr ( term_parse , expr_parse ),  
          tokens_after_expr))
```



# Parsing a Minus Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle - \langle \text{expr} \rangle$

| ( **Minus\_token** :: tokens\_after\_minus) ->  
(match expr tokens\_after\_minus  
with ( expr\_parse , tokens\_after\_expr) ->  
( **Minus\_Expr** ( **term\_parse** , **expr\_parse** ),  
tokens\_after\_expr))





# Parsing an Expression as a Term

---

`<expr> ::= <term>`

| `_ -> (Term_as_Expr term_parse , tokens_after_term))`



- Code for **term** is same except for replacing addition with multiplication and subtraction with division



## Parsing Factor as Id

---

`<factor> ::= <id>`

and factor tokens =

(match tokens

with (Id\_token id\_name :: tokens\_after\_id) =

( **Id\_as\_Factor** id\_name, tokens\_after\_id)



## Parsing Factor as Parenthesized Expression

---

$\langle \text{factor} \rangle ::= ( \langle \text{expr} \rangle )$

| factor ( **Left\_parenthesis** :: tokens) =  
(match expr tokens  
with ( **expr\_parse** , tokens\_after\_expr) ->

# Parsing Factor as Parenthesized Expression

$\langle \text{factor} \rangle ::= ( \langle \text{expr} \rangle )$

(match tokens\_after\_expr

with **Right\_parenthesis** :: tokens\_after\_rparen ->

( **Parenthesized\_Expr\_as\_Factor** **expr\_parse** ,  
tokens\_after\_rparen)



# Error Cases

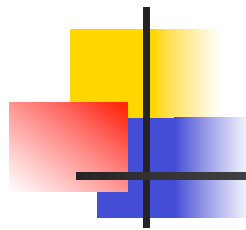
---

- What if no matching right parenthesis?

```
| _ -> raise (Failure "No matching  
rparen" ) )
```

- What if no leading id or left parenthesis?

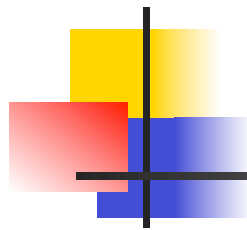
```
| _ -> raise (Failure "No id or lparen" ) );;
```



$( a + b ) * c - d$

---

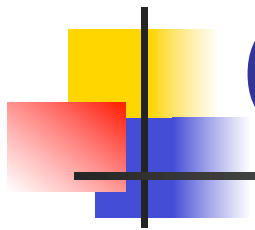
```
expr [Left_parenthesis; Id_token "a";  
      Plus_token; Id_token "b";  
      Right_parenthesis; Times_token;  
      Id_token "c"; Minus_token;  
      Id_token "d"];;
```


$$(a + b) * c - d$$

---

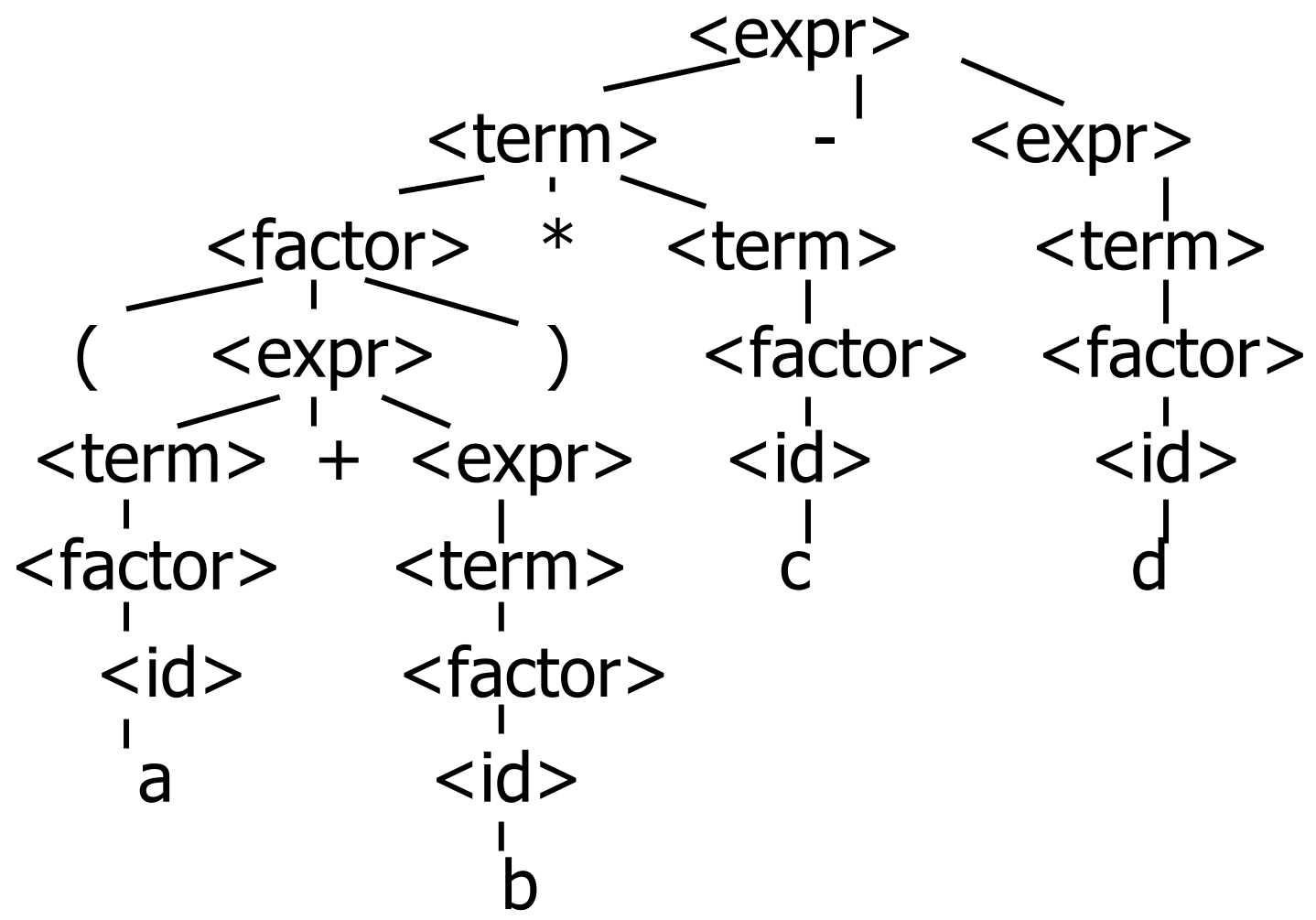
```
- : expr * token list =  
(Minus_Expr  
  (Mult_Term  
    (Parenthesized_Expr_as_Factor  
      (Plus_Expr  
        (Factor_as_Term (Id_as_Factor "a"),  
          Term_as_Expr (Factor_as_Term  
            (Id_as_Factor "b")))),  
        Factor_as_Term (Id_as_Factor "c")),  
      Term_as_Expr (Factor_as_Term (Id_as_Factor  
        "d")))),  
  [])
```

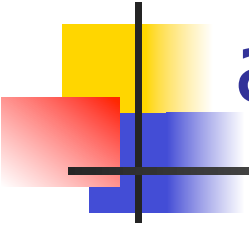




$(a + b) * c - d$

---





a + b \* c - d

---

```
# expr [Id_token "a"; Plus_token; Id_token "b";  
      Times_token; Id_token "c"; Minus_token;  
      Id_token "d"];;
```

```
- : expr * token list =
```

```
(Plus_Expr
```

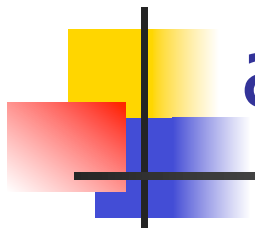
```
  (Factor_as_Term (Id_as_Factor "a"),
```

```
  Minus_Expr
```

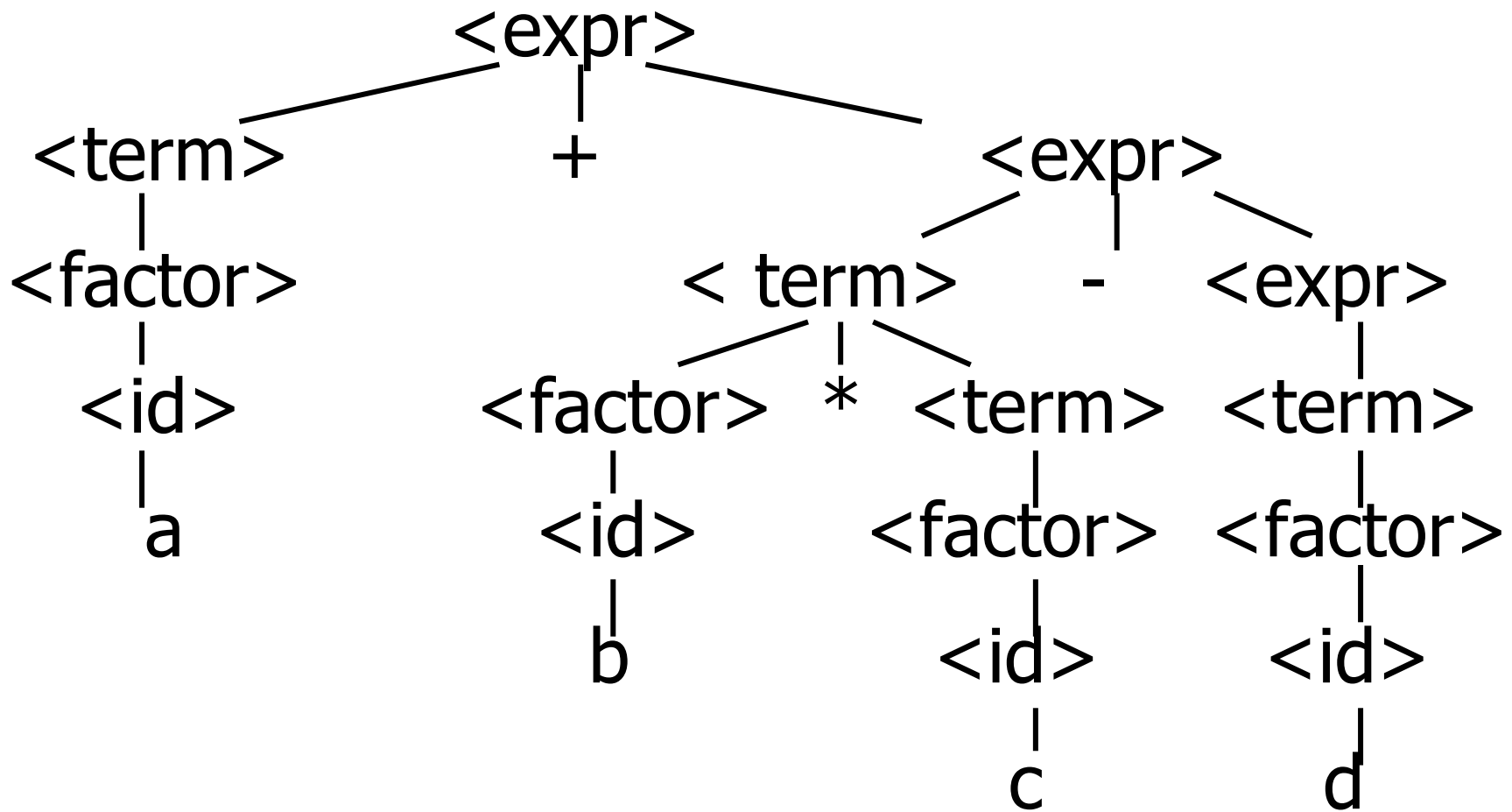
```
    (Mult_Term (Id_as_Factor "b", Factor_as_Term  
              (Id_as_Factor "c")),
```

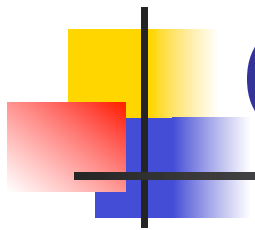
```
    Term_as_Expr (Factor_as_Term (Id_as_Factor  
                                  "d"))),
```

```
[])
```



a + b \* c - d





( a + b \* c - d

---

```
# expr [Left_parenthesis; Id_token "a";  
Plus_token; Id_token "b"; Times_token;  
Id_token "c"; Minus_token; Id_token "d"];;
```

Exception: Failure "No matching rparen".

Can't parse because it was expecting a right parenthesis but it got to the end without finding one



$(a + b) * c - d *$

---

```
expr [Id_token "a"; Plus_token; Id_token "b";  
      Right_parenthesis; Times_token; Id_token "c";  
      Minus_token; Id_token "d"];;
```

- : expr \* token list =

(Plus\_Expr

(Factor\_as\_Term (Id\_as\_Factor "a"),

Term\_as\_Expr (Factor\_as\_Term (Id\_as\_Factor  
"b"))),

[Right\_parenthesis; Times\_token; Id\_token "c";  
Minus\_token; Id\_token "d"])



# Parsing Whole String

---

- Q: How to guarantee whole string parses?
- A: Check returned tokens empty

let parse tokens =

match **expr** tokens

with (expr\_parse, []) -> expr\_parse

| \_ -> raise (Failure "No parse");;

- Fixes <expr> as start symbol



# Streams in Place of Lists

---

- More realistically, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Will use  $(\text{token} * (\text{unit} \rightarrow \text{token}))$  or  $(\text{token} * (\text{unit} \rightarrow \text{token option}))$   
in place of token list



# Problems for Recursive-Descent Parsing

---

- Left Recursion:

$A ::= Aw$

translates to a subroutine that loops forever

- Indirect Left Recursion:

$A ::= Bw$

$B ::= Av$

causes the same problem





## Problems for Recursive-Descent Parsing

---

- Parser must always be able to choose the next action based only on the very next token
- Pairwise Disjointedness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token



# Pairwise Disjointedness Test

---

- For each rule

$$A ::= y$$

Calculate

$$\text{FIRST}(y) =$$

$$\{a \mid y \Rightarrow^* aw\} \cup \{\varepsilon \mid \text{if } y \Rightarrow^* \varepsilon\}$$

- For each pair of rules  $A ::= y$  and  $A ::= z$ , require  $\text{FIRST}(y) \cap \text{FIRST}(z) = \{\}$



## Example

---

Grammar:

$$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$$
$$\langle A \rangle ::= \langle A \rangle b \mid b$$
$$\langle B \rangle ::= a \langle B \rangle \mid a$$
$$\text{FIRST}(\langle A \rangle b) = \{b\}$$
$$\text{FIRST}(b) = \{b\}$$

Rules for  $\langle A \rangle$  not pairwise disjoint



# Eliminating Left Recursion

---

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity

- Given

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$  and

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$

- Add new non-terminal  $\langle e \rangle$  and replace above rules with

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$

$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle \mid \varepsilon$



# Factoring Grammar

---

- Test too strong: Can't handle

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [ ( + | - ) \langle \text{expr} \rangle ]$

- Answer: Add new non-terminal and replace above rules by

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$

$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle$

$\langle e \rangle ::= - \langle \text{term} \rangle \langle e \rangle$

$\langle e \rangle ::= \varepsilon$

- You are delaying the decision point



# Example

---

Both  $\langle A \rangle$  and  $\langle B \rangle$   
have problems:

Transform grammar  
to:

$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$	$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$
$\langle A \rangle ::= \langle A \rangle b \mid b$	$\langle A \rangle ::= b \langle A1 \rangle$
$\langle B \rangle ::= a \langle B \rangle \mid a$	$\langle A1 \rangle ::= b \langle A1 \rangle \mid \varepsilon$
	$\langle B \rangle ::= a \langle B1 \rangle$
	$\langle B1 \rangle ::= a \langle B1 \rangle \mid \varepsilon$



# Semantics

---

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference



# Dynamic semantics

---

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics





# Dynamic Semantics

---

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes



# Operational Semantics

---

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations



# Axiomatic Semantics

---

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages



# Axiomatic Semantics

---

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :  
    {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*



# Denotational Semantics

---

- Construct a function  $\mathcal{M}$  assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs



# Natural Semantics

---

- Aka Structural Operational Semantics, aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

or

$$(E, m) \Downarrow v$$



# Simple Imperative Programming Language

---

- $I \in \text{Identifiers}$
- $N \in \text{Numerals}$
- $B ::= \text{true} \mid \text{false} \mid B \ \& \ B \mid B \ \text{or} \ B \mid \text{not} \ B$   
 $\mid E < E \mid E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- $C ::= \text{skip} \mid C; C \mid I ::= E$   
 $\mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$

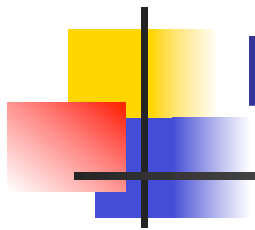


# Natural Semantics of Atomic Expressions

---

- Identifiers:  $(I, m) \Downarrow m(I)$
- Numerals are values:  $(N, m) \Downarrow N$
- Booleans:  $(\text{true}, m) \Downarrow \text{true}$   
 $(\text{false}, m) \Downarrow \text{false}$





# Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \ \& \ B', m) \Downarrow \text{false}}$$

$$\frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \ \& \ B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \ \text{or} \ B', m) \Downarrow \text{true}}$$

$$\frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \ \text{or} \ B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}}$$

$$\frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$



# Relations

---

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

- By  $U \sim V = b$ , we mean does (the meaning of) the relation  $\sim$  hold on the meaning of  $U$  and  $V$
- May be specified by a mathematical expression/equation or rules matching  $U$  and  $V$



# Arithmetic Expressions

---

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \text{ op } V = N}{(E \text{ op } E', m) \Downarrow N}$$

where  $N$  is the specified value for  $U \text{ op } V$



# Commands

---

Skip:  $(\text{skip}, m) \Downarrow m$

Assignment: 
$$\frac{(E, m) \Downarrow V}{(I ::= E, m) \Downarrow m[I \leftarrow V]}$$

Sequencing: 
$$\frac{(C, m) \Downarrow m' \quad (C', m') \Downarrow m''}{(C; C', m) \Downarrow m''}$$



## If Then Else Command

---

$$\frac{(B, m) \Downarrow \text{true} \quad (C, m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

$$\frac{(B, m) \Downarrow \text{false} \quad (C', m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$



# While Command

---

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B, m) \Downarrow \text{true} \quad (C, m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$



## Example: If Then Else Rule

---

---

(if  $x > 5$  then  $y := 2 + 3$  else  $y := 3 + 4$  fi,  
 $\{x \rightarrow 7\}) \Downarrow ?$



# Example: If Then Else Rule

---

---

$(x > 5, \{x \rightarrow 7\}) \Downarrow ?$

---

$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}$   
 $\{x \rightarrow 7\}) \Downarrow ?$





# Example: Arith Relation

---

? > ? = ?

$(x, \{x \rightarrow 7\}) \Downarrow ? \quad (5, \{x \rightarrow 7\}) \Downarrow ?$

---

$(x > 5, \{x \rightarrow 7\}) \Downarrow ?$

---

(if  $x > 5$  then  $y := 2 + 3$  else  $y := 3 + 4$  fi,  
 $\{x \rightarrow 7\}) \Downarrow ?$



## Example: Identifier(s)

---

$7 > 5 = \text{true}$

$(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5$

$(x > 5, \{x \rightarrow 7\}) \Downarrow ?$

---

$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?$



## Example: Arith Relation

---

$7 > 5 = \text{true}$

$(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5$

---

$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$

---

$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}$   
 $\{x \rightarrow 7\}) \Downarrow ?$



# Example: If Then Else Rule

---

$7 > 5 = \text{true}$

$(x, \{x \rightarrow 7\}) \Downarrow 7$     $(5, \{x \rightarrow 7\}) \Downarrow 5$

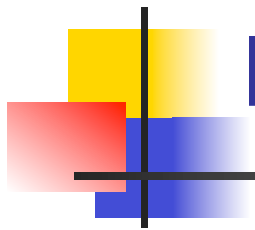
$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$

$(y := 2 + 3, \{x \rightarrow 7\})$

$\Downarrow ?$

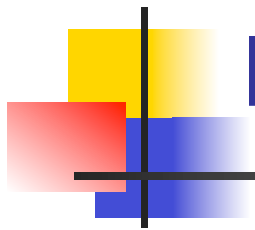
$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}$

$\{x \rightarrow 7\}) \Downarrow ?$



# Example: Assignment

$$\begin{array}{c}
7 > 5 = \text{true} \\
\hline
(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \\
(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \\
\hline
(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\}) \Downarrow ?
\end{array}$$



# Example: Arith Op

$$? + ? = ?$$

$$\frac{(2, \{x \rightarrow 7\}) \Downarrow ? \quad (3, \{x \rightarrow 7\}) \Downarrow ?}{}$$

$$7 > 5 = \text{true}$$

$$\frac{(2+3, \{x \rightarrow 7\}) \Downarrow ?}{}$$

$$\frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5}{}$$

$$(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow ?$$

$$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$$

$$\Downarrow ?$$

$$\frac{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \quad (y := 2 + 3, \{x \rightarrow 7\}) \Downarrow ?}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}$$

$$\{x \rightarrow 7\}) \Downarrow ?$$



# Example: Numerals

---

$$2 + 3 = 5$$

$$\frac{(2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3}{\quad}$$

$$7 > 5 = \text{true}$$

$$\frac{(2+3, \{x \rightarrow 7\}) \Downarrow ?}{\quad}$$

$$\frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5}{\quad}$$

$$(y := 2 + 3, \{x \rightarrow 7\})$$

$$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$$

$$\Downarrow ?$$

$$\frac{(x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}{\quad}$$



# Example: Arith Op

---

$$2 + 3 = 5$$

$$\frac{(2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3}{\quad}$$

$$7 > 5 = \text{true}$$

$$\frac{(2+3, \{x \rightarrow 7\}) \Downarrow 5}{\quad}$$

$$\frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5}{\quad}$$

$$(y := 2 + 3, \{x \rightarrow 7\})$$

$$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$$

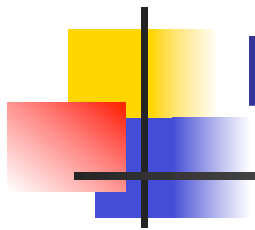
$$\Downarrow ?$$

---


$$\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}$$

$$\{x \rightarrow 7\}) \Downarrow ?$$





# Example: Assignment

$$\begin{array}{c}
 2 + 3 = 5 \\
 \hline
 (2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3 \\
 \hline
 (2+3, \{x \rightarrow 7\}) \Downarrow 5 \\
 \hline
 (y := 2 + 3, \{x \rightarrow 7\}) \\
 \Downarrow \{x \rightarrow 7, y \rightarrow 5\} \\
 \hline
 \begin{array}{c}
 7 > 5 = \text{true} \\
 \hline
 (x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \\
 \hline
 (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \\
 \hline
 (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
 \{x \rightarrow 7\}) \Downarrow ?
 \end{array}
 \end{array}$$



## Example: If Then Else Rule

---

$$\begin{array}{c}
 2 + 3 = 5 \\
 \frac{(2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3}{(2+3, \{x \rightarrow 7\}) \Downarrow 5} \\
 7 > 5 = \text{true} \\
 \frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}} \quad \frac{(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow 5}{\Downarrow \{x \rightarrow 7, y \rightarrow 5\}} \\
 \hline
 (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
 \{x \rightarrow 7\}) \Downarrow \{x \rightarrow 7, y \rightarrow 5\}
 \end{array}$$



## Let in Command

---

$$\frac{(E, m) \Downarrow v \quad (C, m[I \leftarrow v]) \Downarrow m'}{(\text{let } I = E \text{ in } C, m) \Downarrow m''}$$

Where  $m''(y) = m'(y)$  for  $y \neq I$  and  
 $m''(I) = m(I)$  if  $m(I)$  is defined,  
and  $m''(I)$  is undefined otherwise



# Example

---

$$\frac{(x, \{x \rightarrow 5\}) \Downarrow 5 \quad (3, \{x \rightarrow 5\}) \Downarrow 3}{}$$

$$\frac{(x+3, \{x \rightarrow 5\}) \Downarrow 8}{}$$

$$\frac{(5, \{x \rightarrow 17\}) \Downarrow 5 \quad (x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}{}$$

$$(\text{let } x = 5 \text{ in } (x := x+3), \{x \rightarrow 17\}) \Downarrow ?$$



# Example

---

$$\frac{(x, \{x \rightarrow 5\}) \Downarrow 5 \quad (3, \{x \rightarrow 5\}) \Downarrow 3}{}$$

$$\frac{(x+3, \{x \rightarrow 5\}) \Downarrow 8}{}$$

$$\frac{(5, \{x \rightarrow 17\}) \Downarrow 5 \quad (x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}}{}$$

$$(\text{let } x = 5 \text{ in } (x := x+3), \{x \rightarrow 17\}) \Downarrow \{x \rightarrow 17\}$$



# Comment

---

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explicitly*
- Clash of constructs apparent in awkward semantics



# Interpretation Versus Compilation

---

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed



# Interpreter

---

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations





# Interpreter

---

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next “state”
  - To get final value, put in a loop



## Natural Semantics Example

---

- $\text{compute\_exp} (\text{Var}(v), m) = \text{look\_up } v \ m$
- $\text{compute\_exp} (\text{Int}(n), \_) = \text{Num } (n)$
- ...
- $\text{compute\_com}(\text{IfExp}(b,c1,c2),m) =$   
if  $\text{compute\_exp} (b,m) = \text{Bool}(\text{true})$   
then  $\text{compute\_com} (c1,m)$   
else  $\text{compute\_com} (c2,m)$



## Natural Semantics Example

---

- $\text{compute\_com}(\text{While}(b,c), m) =$   
if  $\text{compute\_exp}(b,m) = \text{Bool}(\text{false})$   
then  $m$   
else  $\text{compute\_com}$   
     $(\text{While}(b,c), \text{compute\_com}(c,m))$
- May fail to terminate - exceed stack limits
- Returns no useful information then