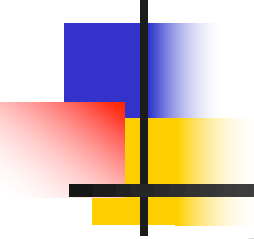


Programming Languages and Compilers (CS 421)

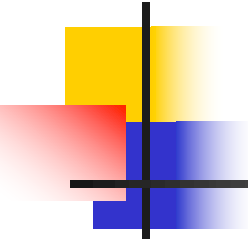


Elsa L Gunter

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

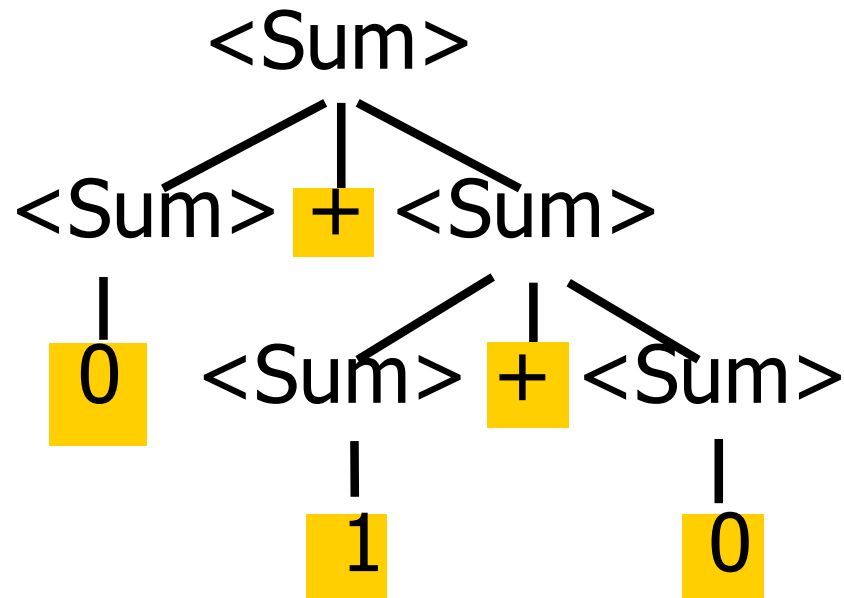
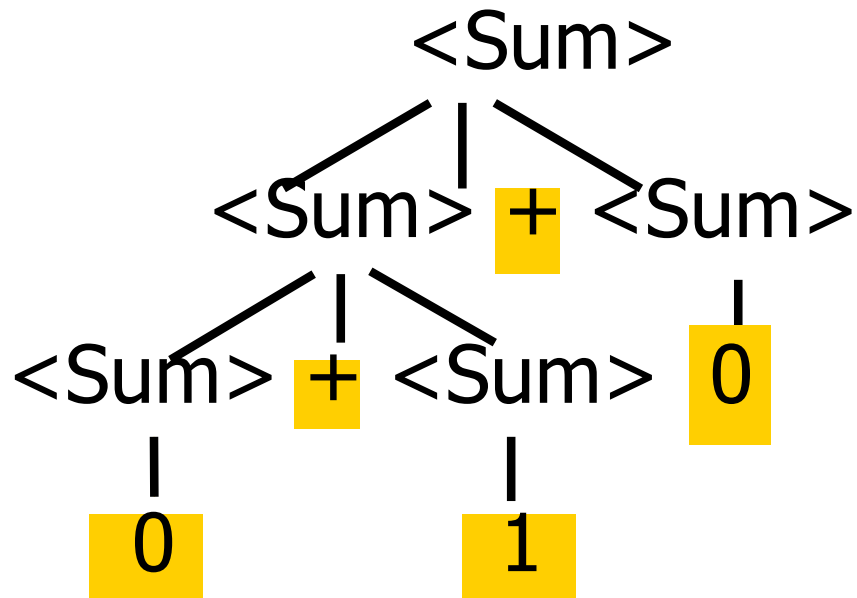


Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree
- If all BNF's for a language are ambiguous then the language is *inherently ambiguous*

Example: Ambiguous Grammar

■ $0 + 1 + 0$

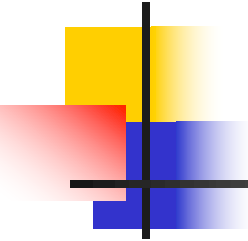




Example

- What is the value of:

$$7 - 5 - 2$$



Example

- What is the value of:

$$7 - 5 - 2$$

- Possible answers:

- In Pascal, C++, SML assoc. left

$$7 - 5 - 2 = (7 - 5) - 2 = 0$$

- In APL, associate to right

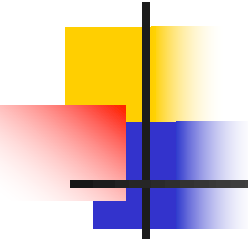
$$7 - 5 - 2 = 7 - (5 - 2) = 4$$



Example

- What is the result for:

$$3 + 4 * 5 + 6$$



Example

- What is the result for:

$$3 + 4 * 5 + 6$$

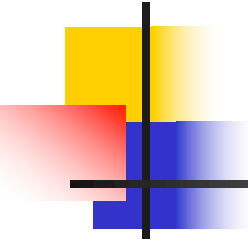
- Possible answers:

- $41 = ((3 + 4) * 5) + 6$

- $47 = 3 + (4 * (5 + 6))$

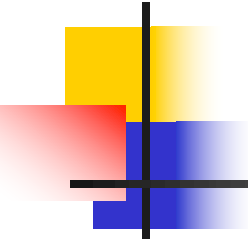
- $29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)$

- $77 = (3 + 4) * (5 + 6)$



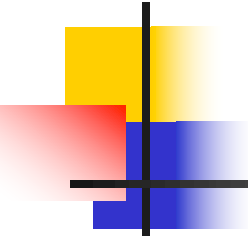
Disambiguating a Grammar

- Given ambiguous grammar G , with start symbol S , find a grammar G' with same start symbol, such that
$$\text{language of } G = \text{language of } G'$$
- Not always possible
- No algorithm in general



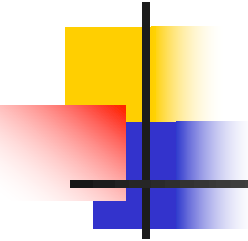
Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
 - In fact each string of terminals and non-terminals represents all strings having some property (Remember reg exp semantics)
- Identify these properties (often in terms of things that can't happen)
- Use these properties to inductively guarantee every string in language has a unique parse



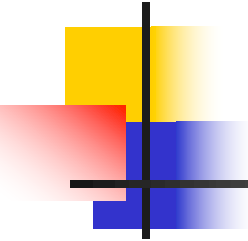
Steps to Grammar Disambiguation

- Identify the rules and a smallest use that display ambiguity
- Decide which parse to keep; why should others be thrown out?
- What syntactic restrictions on subexpressions are needed to throw out the bad (while keeping the good)?
- Add a new non-terminal and rules to describe this set of restricted subexpressions (called stratifying, or refactoring)
- **Characterize each non-terminal by a language invariant**
- Replace old rules to use new non-terminals
- Rinse and repeat



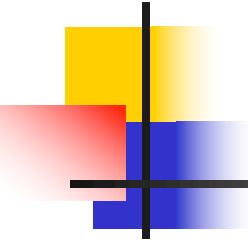
Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity
 - Will come back to this



How to Enforce Associativity

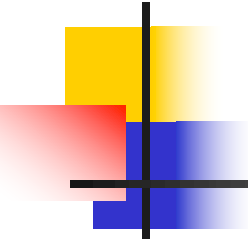
- Have at most one recursive call per production
- When two or more recursive calls would be natural leave right-most one for right associativity, left-most one for left associativity



Example

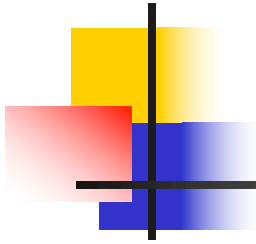
- $\langle \text{Sum} \rangle ::= 0 \mid 1 \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
 $\mid (\langle \text{Sum} \rangle)$
- Becomes
 - $\langle \text{Sum} \rangle ::= \langle \text{Num} \rangle \mid \langle \text{Num} \rangle + \langle \text{Sum} \rangle$
 - $\langle \text{Num} \rangle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

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



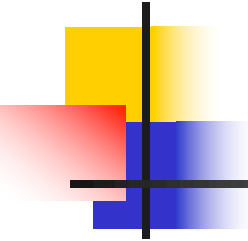
Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar



Precedence Table - Sample

	Fortan	Pascal	C/C++	Ada	SML
highest	**	*, /, div, mod	++, --	**	div, mod, /, *
	*, /	+, -	*, /, %	*, /, mod	+, -, ^
	+, -		+, -	+, -	::



Predence in Grammar

- Higher precedence translates to longer derivation chain

- Example:

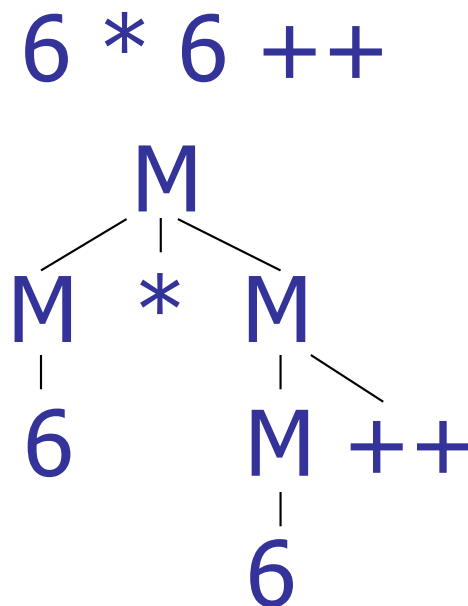
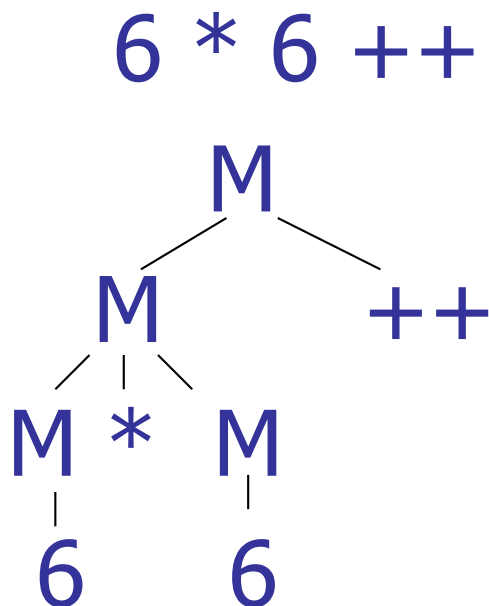
$$\begin{aligned} \langle \text{exp} \rangle ::= & 0 \mid 1 \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \\ & \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \end{aligned}$$

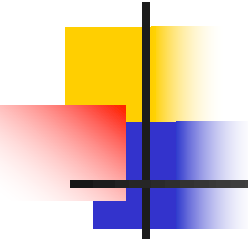
- Becomes

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{mult_exp} \rangle \\ & \mid \langle \text{exp} \rangle + \langle \text{mult_exp} \rangle \\ \langle \text{mult_exp} \rangle ::= & \langle \text{id} \rangle \mid \langle \text{mult_exp} \rangle * \langle \text{id} \rangle \\ \langle \text{id} \rangle ::= & 0 \mid 1 \end{aligned}$$

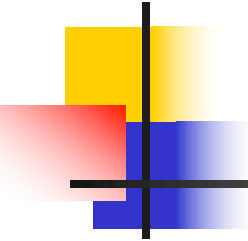
More Disambiguating Grammars

- $M ::= M * M \mid (M) \mid M ++ \mid 6$
- Ambiguous because of associativity of $*$
- because of conflict between $*$ and $++$:



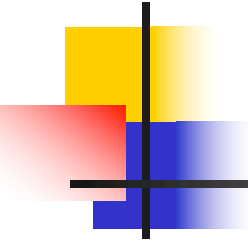

$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

- How to disambiguate?
- Choose associativity for $*$
- Choose precedence between $*$ and $++$
- Four possibilities
- Four different approaches
- Some easier than others
- Will do --- My choice

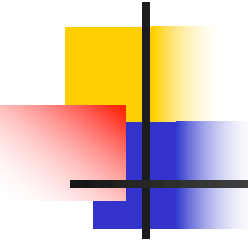

$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

- Think about $6 * 6 ++ * 6 * 6 ++$
- Let's start with observations
- If $*$ binds less tightly than $++$, then no $*$ can be the immediate subtree to a $++$.
 - We would need a language for things that don't parse as $*$
- If $*$ binds more tightly than $++$, then ...
- The right subtree to $*$ can't be a $++$
- But the left can!

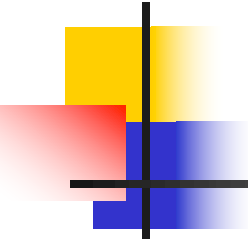
■ Need different languages of the left and right


$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

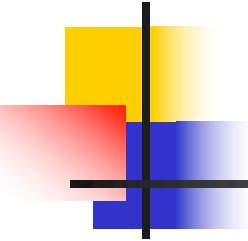
- * higher prec than ++
 - $6 * 6 ++ * 6 ++ * 6$
- $M ::= M ++ \mid \text{StarExp} \mid (M) \mid 6$
- What is StarExp
- It is everything that parses as a * and can't parse as a ++
- But what is the associativity of *?
- I'll choose right


$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

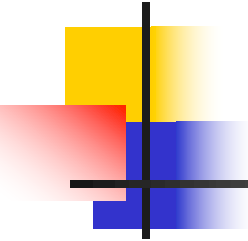
- * higher prec than ++
 - $6 * 6 ++ * 6 ++ * 6$
- * Right assoc but ++ takes all to its left
- $M ::= M++ \mid \text{StarExp} \mid (M) \mid 6$
- $\text{StarExp} ::= \text{NotStar} * \text{PossStarNotPlusPlus}$
- What is **NotStar**? It can't be *, but can be 6 or (M)
- Can it be ++? YES!
- $\text{NotStar} ::= M++ \mid (M) \mid 6$


$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

- * higher prec than ++, * right assoc
- $\text{StarExp} ::= \text{NotStar} * \text{PossStarNoPlusPlus}$
- What is $\text{PossStarNoPlusPlus}$?
- $\text{PossStarNoPlusPlus} ::=$
 $(M) \mid 6$
- $\mid \text{NotStarNoPlusPlus} * \text{PossStarNoPlusPlus}$
- But what is $\text{NotStarNotPlusPlus}$?
- Well, the other two original rules: $(M) \mid 6$


$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

- * higher prec than ++, * right assoc
- $M ::= \text{StarExp} \mid M++ \mid (M) \mid 6$
- $\text{NotStar} ::= M++ \mid (M) \mid 6$
- $\text{StarExp} ::= \text{NotStar} * \text{PossStarNoPlusPlus}$
- $\text{PossStarNoPlusPlus} ::=$
 $(M) \mid 6$
 $\mid \text{NotStarNoPlusPlus} * \text{PossStarNoPlusPlus}$
- $\text{NotStarNotPlusPlus} ::= (M) \mid 6$


$$M ::= M * M \mid (M) \mid M ++ \mid 6$$

- * higher prec than ++, * right assoc

- $M ::= \text{StarExp} \mid \text{NotStar}$

$\text{NotStar} ::= M++ \mid \text{NotStarNotPlusPlus}$

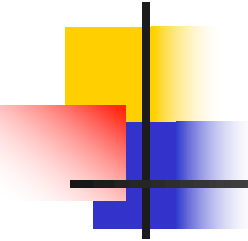
$\text{StarExp} ::= \text{NotStar} * \text{PossStarNoPlusPlus}$

$\text{PossStarNoPlusPlus} ::=$

$\text{NotStarNotPlusPlus}$

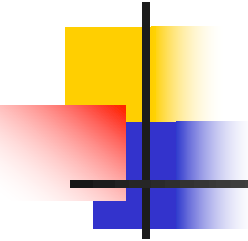
$\mid \text{NotStarNoPlusPlus} * \text{PossStarNoPlusPlus}$

$\text{NotStarNotPlusPlus} ::= (M) \mid 6$



Parser Code

- `<grammar>.ml` defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point



Ocamlyacc Input

- File format:

%{

<header>

%}

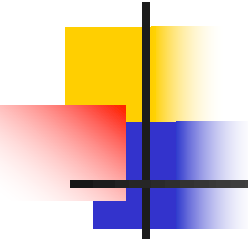
<declarations>

%%

<rules>

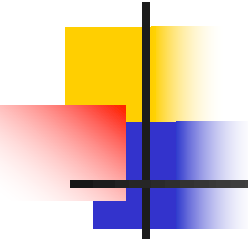
%%

<trailer>



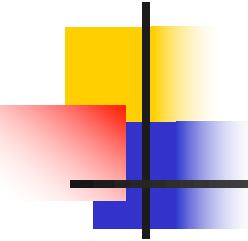
Ocamlyacc <header>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <footer> similar. Possibly used to call parser



Ocamlyacc <declarations>

- **%token** *symbol ... symbol*
- Declare given symbols as tokens
- **%token** <*type*> *symbol ... symbol*
- Declare given symbols as token constructors, taking an argument of type <*type*>
- **%start** *symbol ... symbol*
- Declare given symbols as entry points; functions of same names in <*grammar*>.ml



Ocamlyacc *<declarations>*

- **%type** *<type> symbol ... symbol*

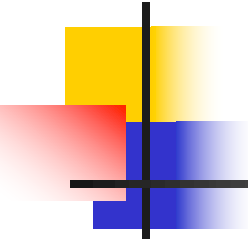
Specify type of attributes for given symbols.
Mandatory for start symbols

- **%left** *symbol ... symbol*

- **%right** *symbol ... symbol*

- **%nonassoc** *symbol ... symbol*

Associate precedence and associativity to given symbols. Same line, same precedence; earlier line, lower precedence (broadest scope)



Ocamlyacc *<rules>*

- *nonterminal* :

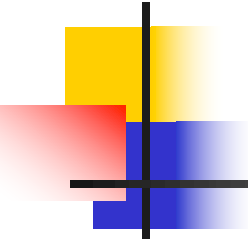
symbol ... symbol { semantic_action }

|
...
|

symbol ... symbol { semantic_action }

;
/

- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access semantic attributes (values) of symbols by position: \$1 for first symbol, \$2 to second ...



Example - Base types

```
(* File: expr.ml *)
```

```
type expr =
```

```
  Term_as_Expr of term
```

```
  | Plus_Expr of (term * expr)
```

```
  | Minus_Expr of (term * expr)
```

```
and term =
```

```
  Factor_as_Term of factor
```

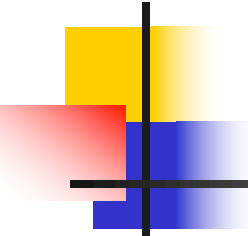
```
  | Mult_Term of (factor * term)
```

```
  | Div_Term of (factor * term)
```

```
and factor =
```

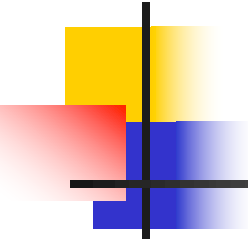
```
  Id_as_Factor of string
```

```
  | Parenthesized_Expr_as_Factor of expr
```



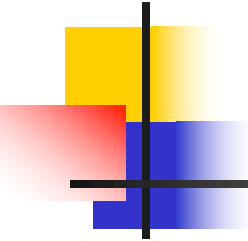
Example - Lexer (exprlex.ml)

```
{ (*open Exprparse*) }  
let numeric = ['0' - '9']  
let letter = ['a' - 'z' 'A' - 'Z']  
rule token = parse  
| "+" {Plus_token}  
| "-" {Minus_token}  
| "*" {Times_token}  
| "/" {Divide_token}  
| "(" {Left_parenthesis}  
| ")" {Right_parenthesis}  
| letter (letter|numeric|"_" )* as id {Id_token id}  
| [' ' '\t' '\n'] {token lexbuf}  
| eof {EOL}
```



Example - Parser (exprparse.mly)

```
%{ open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```



Example - Parser (exprparse.mly)

expr:

term

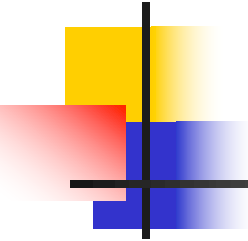
{ Term_as_Expr \$1 }

| term Plus_token expr

{ Plus_Expr (\$1, \$3) }

| term Minus_token expr

{ Minus_Expr (\$1, \$3) }



Example - Parser (exprparse.mly)

term:

factor

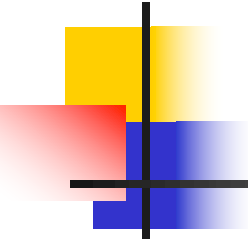
{ Factor_as_Term \$1 }

| factor Times_token term

{ Mult_Term (\$1, \$3) }

| factor Divide_token term

{ Div_Term (\$1, \$3) }



Example - Parser (exprparse.mly)

factor:

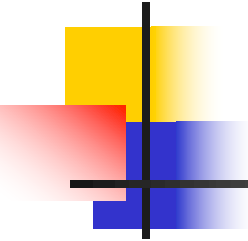
Id_token

{ Id_as_Factor \$1 }

| Left_parenthesis expr Right_parenthesis
{Parenthesized_Expr_as_Factor \$2 }

main:

| expr EOL
{ \$1 }



Example - Using Parser

```
# #use "expr.ml";;
```

```
...
```

```
# #use "exprparse.ml";;
```

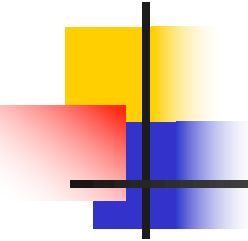
```
...
```

```
# #use "exprlex.ml";;
```

```
...
```

```
# let test s =
```

```
    let lexbuf = Lexing.from_string (s^"\n") in  
        main token lexbuf;;
```



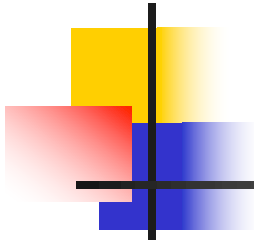
Example - Using Parser

```
# test "a + b";;
```

```
- : expr =
```

```
Plus_Expr
```

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



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

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

$= \text{pink dot} (0 + 1) + 0$ shift



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

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

$$= (\bullet 0 + 1) + 0$$

$$= \bullet (0 + 1) + 0$$

shift

shift



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

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

$$\Rightarrow (0 \text{ } \bullet \text{ } + 1) + 0$$

$$= (\bullet 0 + 1) + 0$$

$$= \bullet (0 + 1) + 0$$

reduce

shift

shift



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

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

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

$$\Rightarrow (0 \bullet + 1) + 0$$

$$= (\bullet 0 + 1) + 0$$

$$= \bullet (0 + 1) + 0$$

shift

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



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

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

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

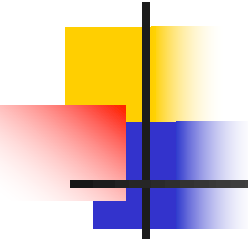
$\langle \text{Sum} \rangle \Rightarrow$

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

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

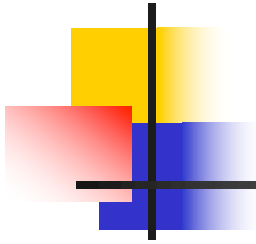


Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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

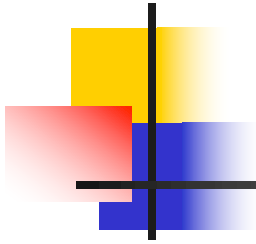
$\langle \text{Sum} \rangle \bullet \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet$	reduce
$\Rightarrow \langle \text{Sum} \rangle + 0 \bullet$	reduce
$= \langle \text{Sum} \rangle + \bullet 0$	shift
$= \langle \text{Sum} \rangle \bullet + 0$	shift
$\Rightarrow (\langle \text{Sum}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

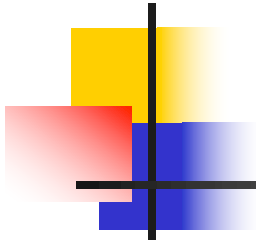




Example

(0 + 1) + 0

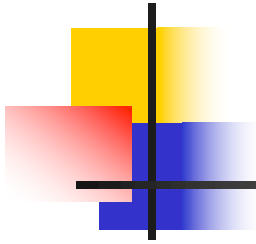




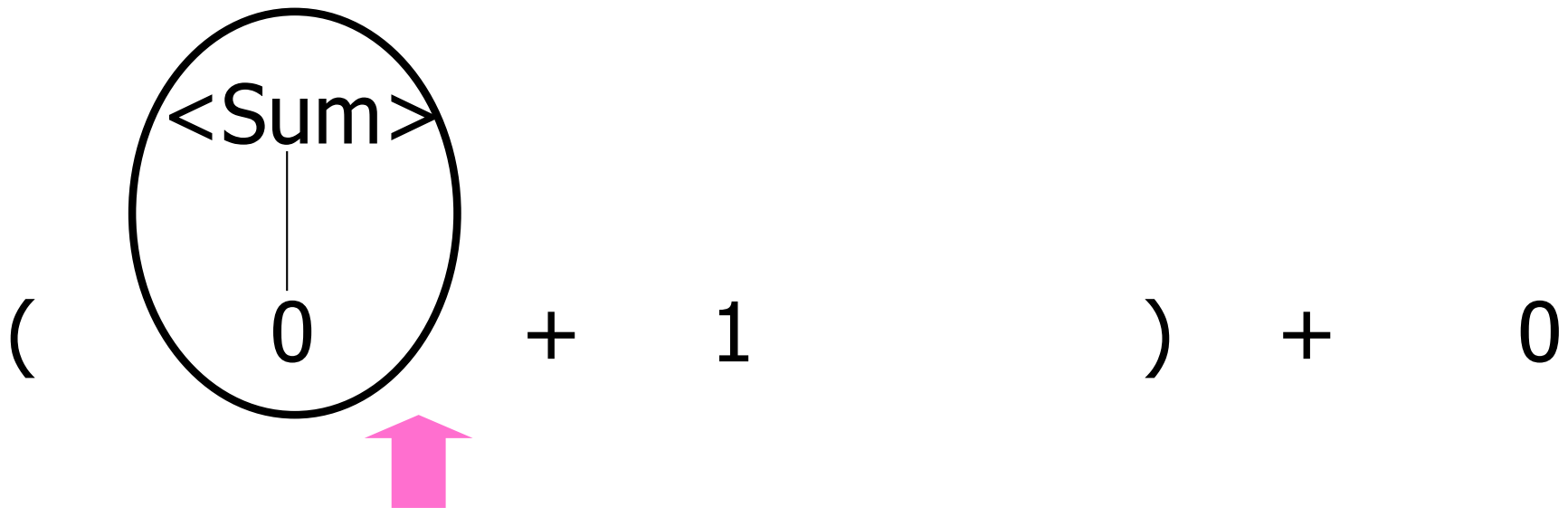
Example

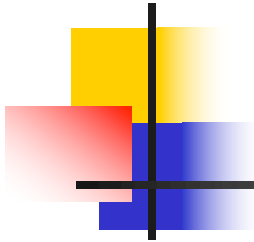
(0 + 1) + 0





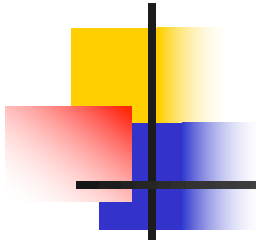
Example





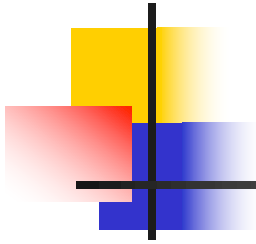
Example

$$\left(\begin{array}{c} \text{<Sum>} \\ | \\ 0 \end{array} \right) + \underset{\uparrow}{1} \quad) + 0$$

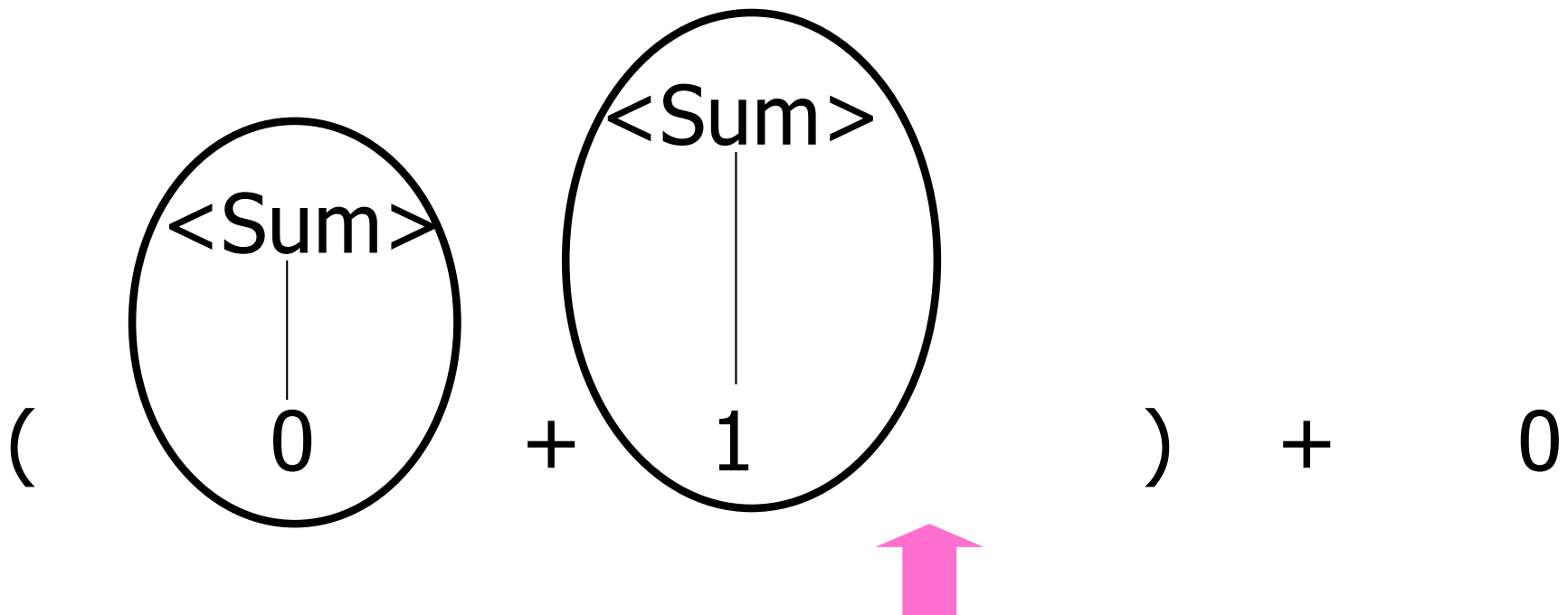


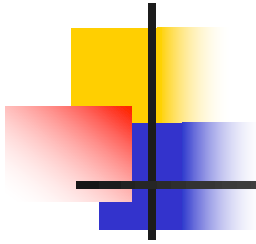
Example

$$\left(\begin{array}{c} \text{<Sum>} \\ | \\ 0 \end{array} \right) + 1 \quad \quad \quad) + 0$$

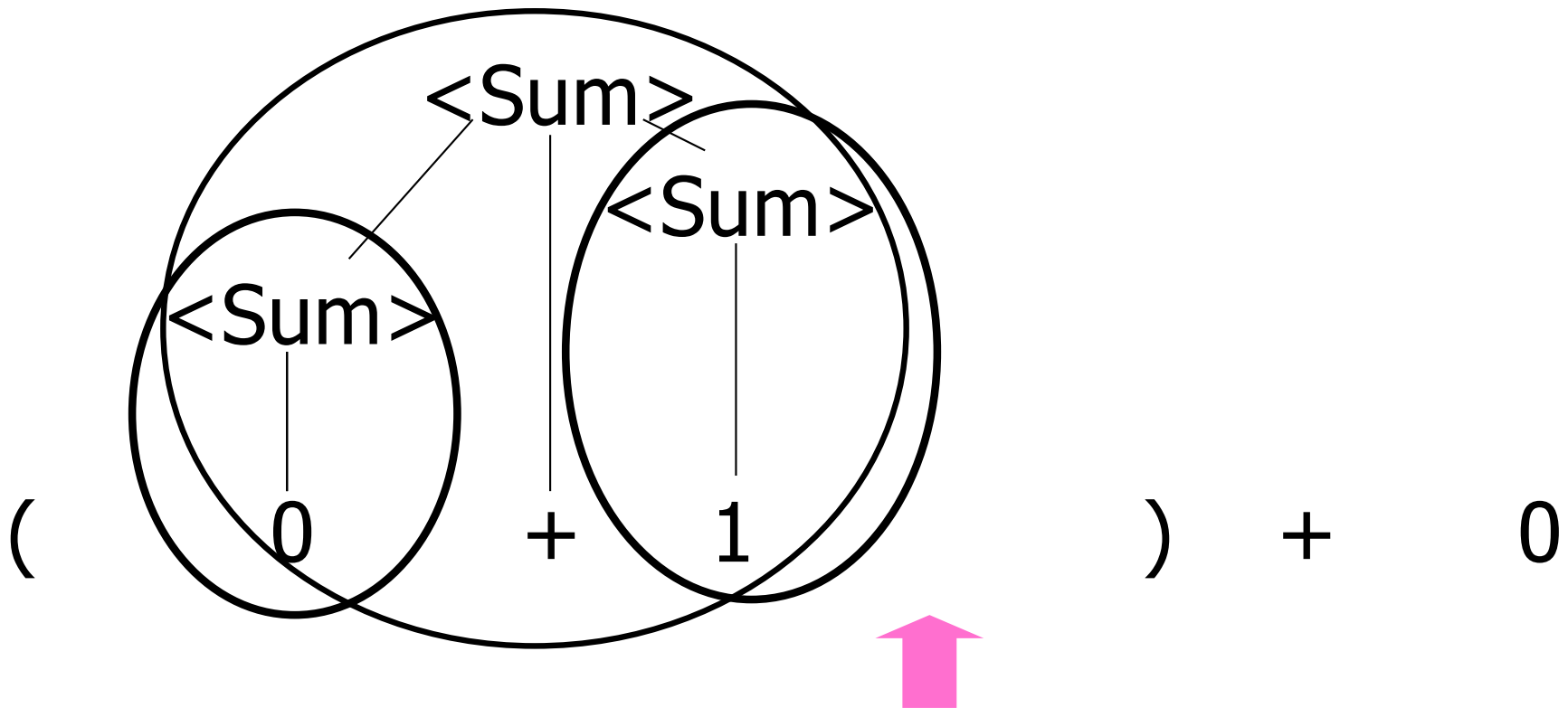



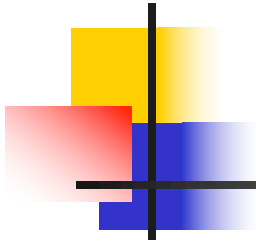
Example



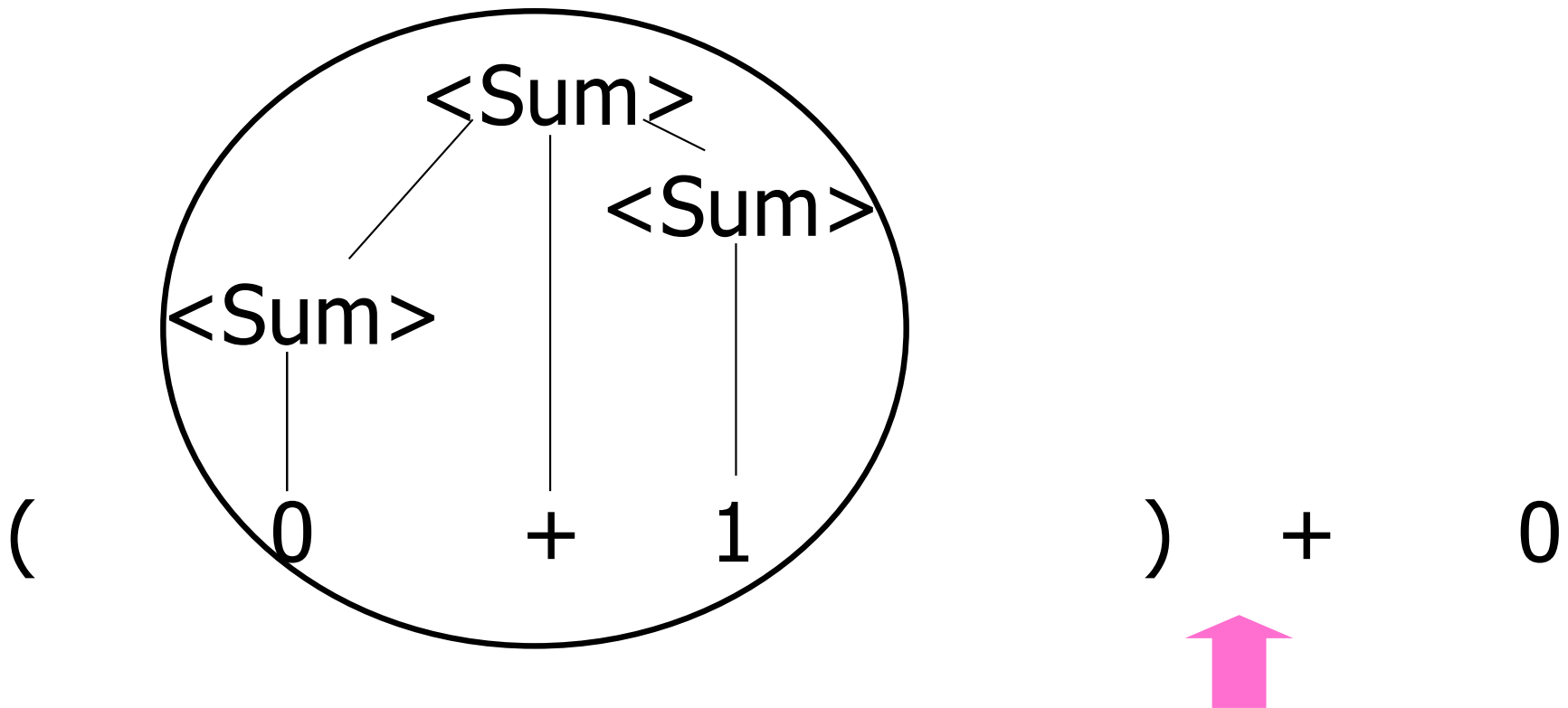


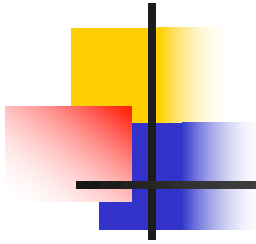
Example



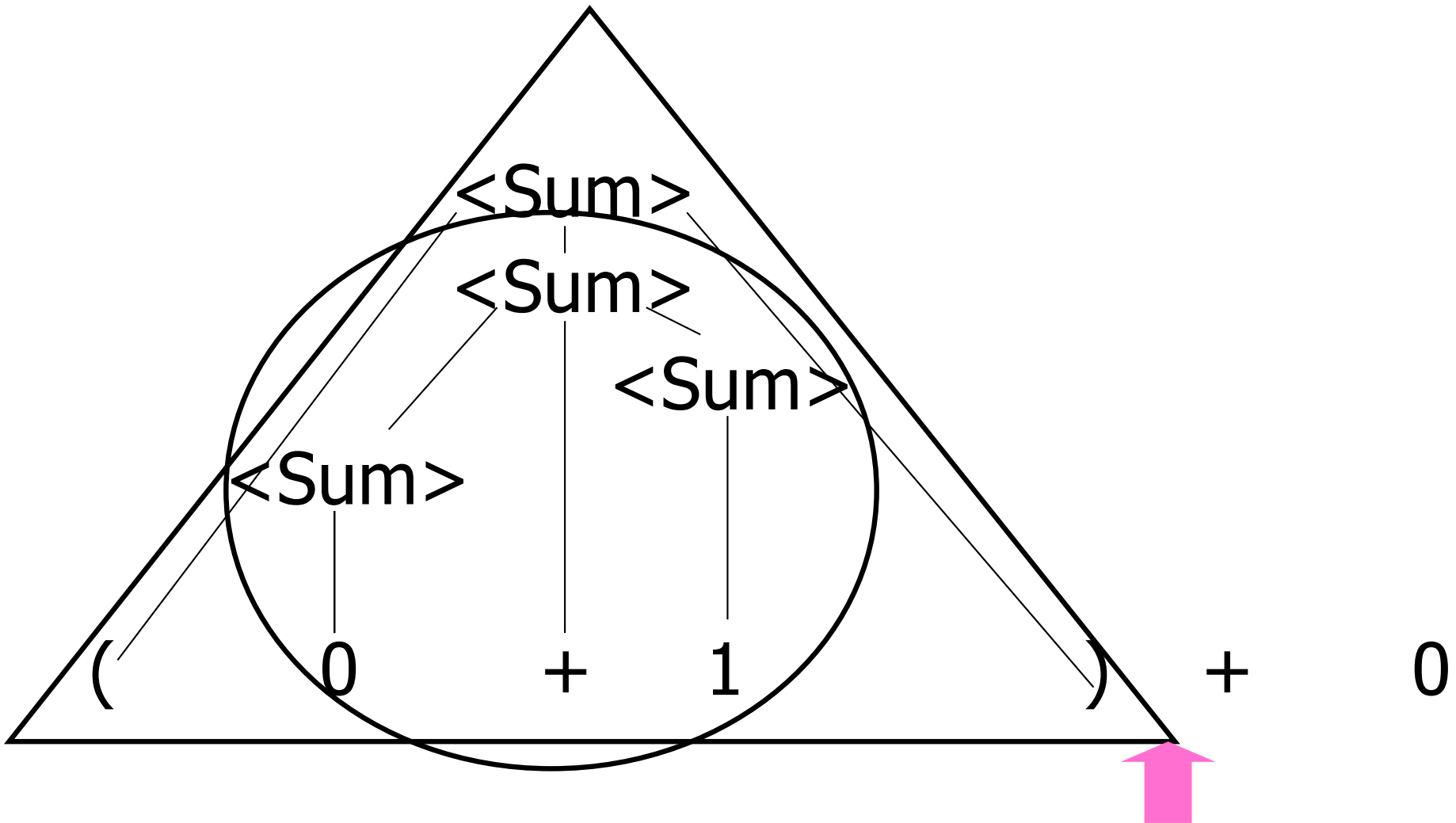


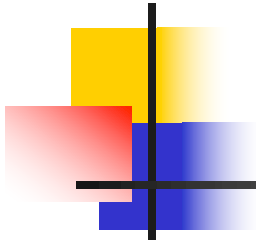
Example



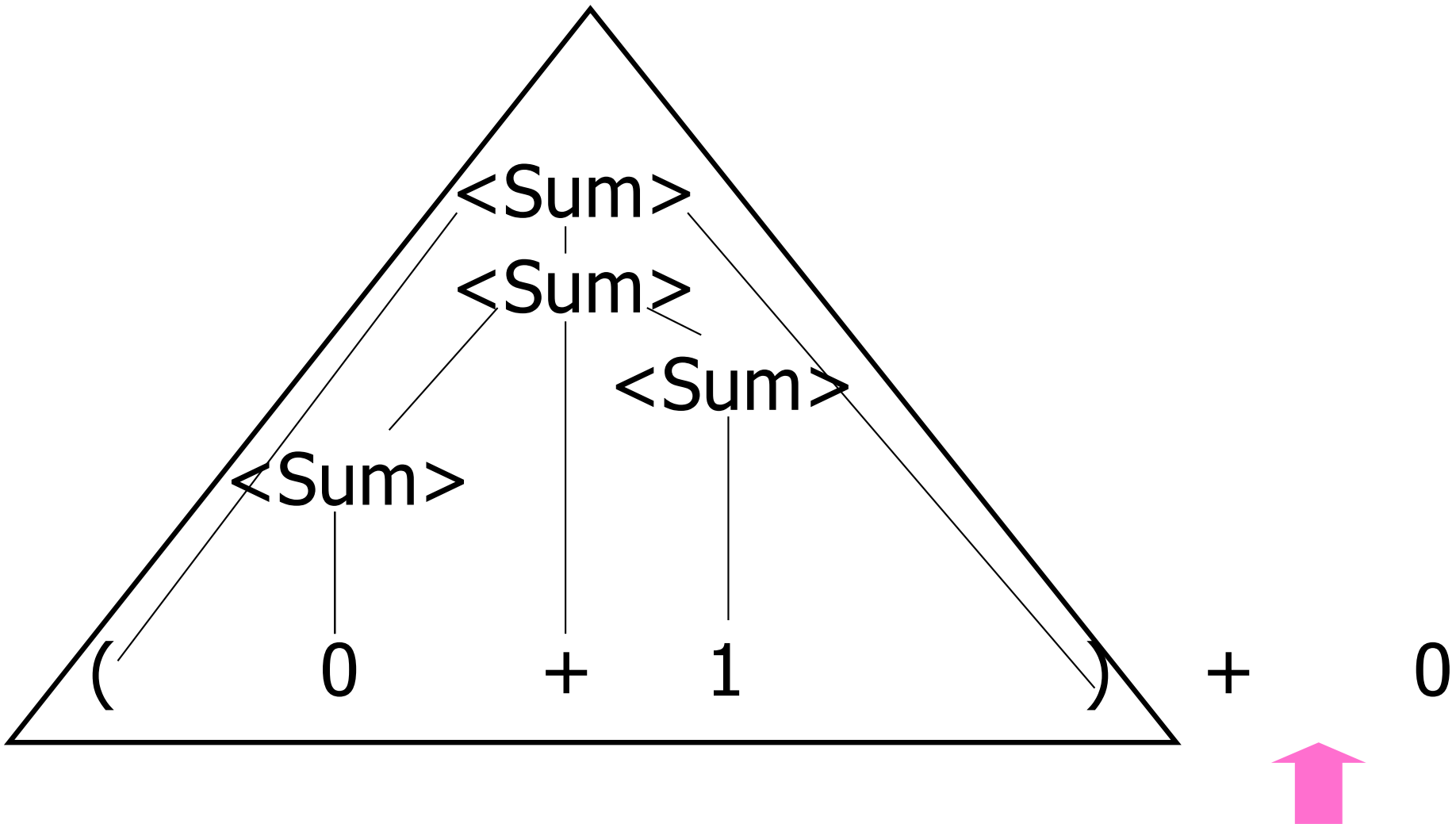


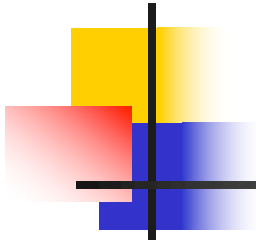
Example



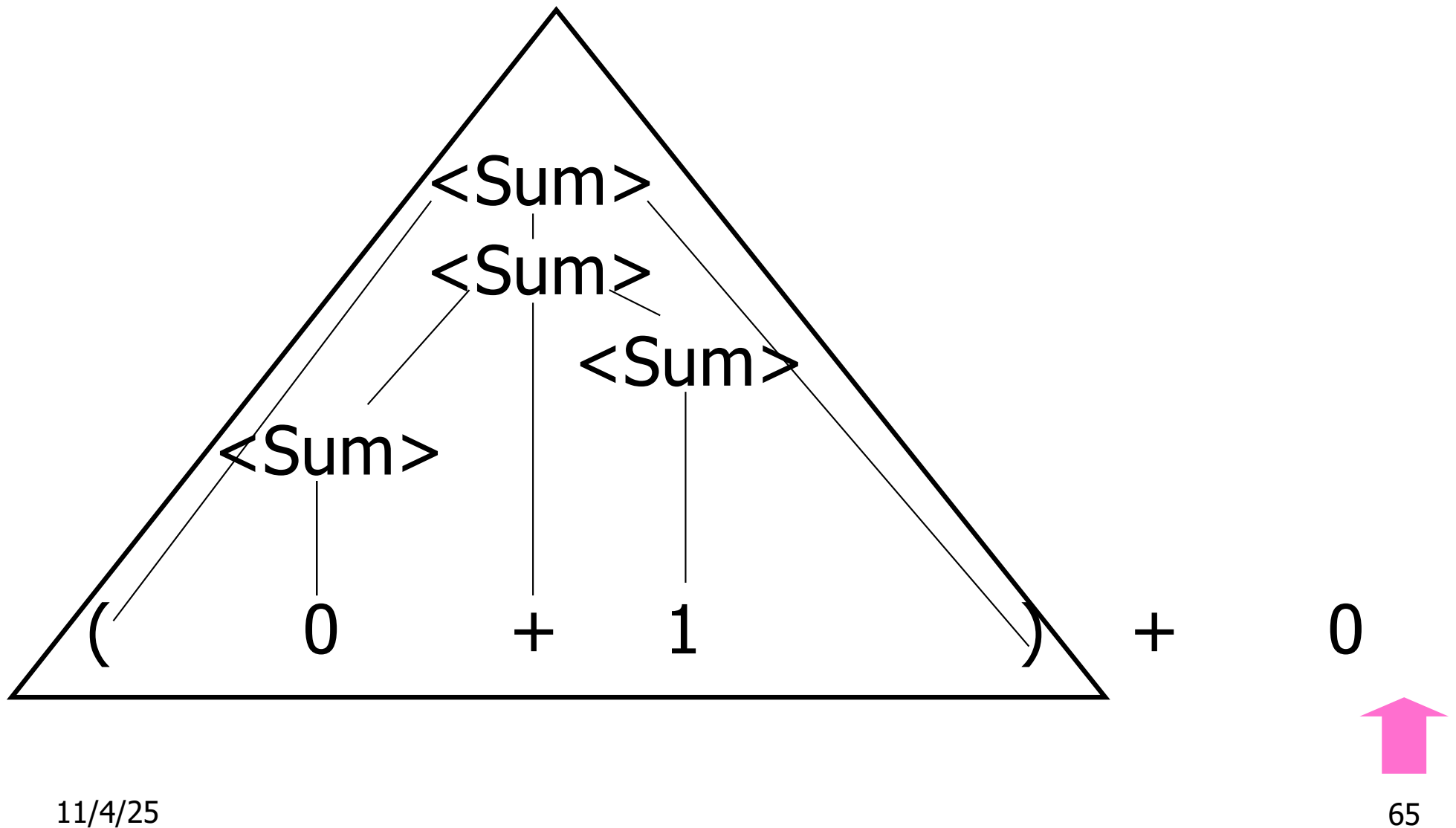


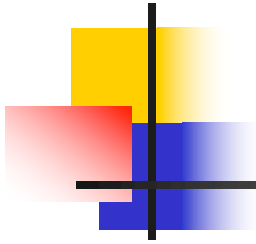
Example



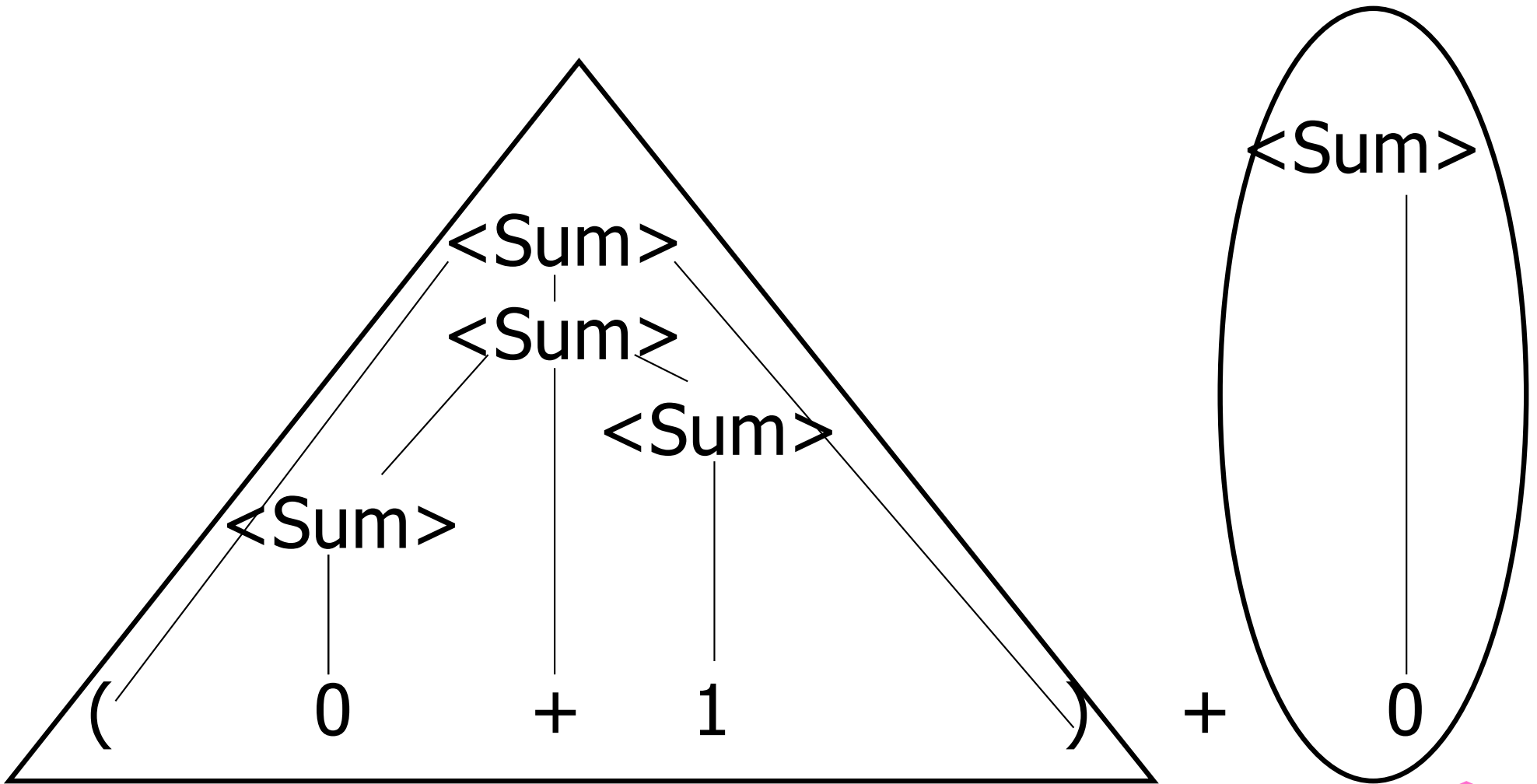


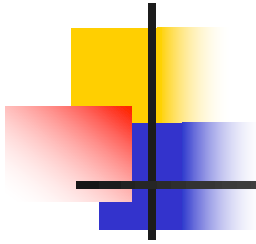
Example



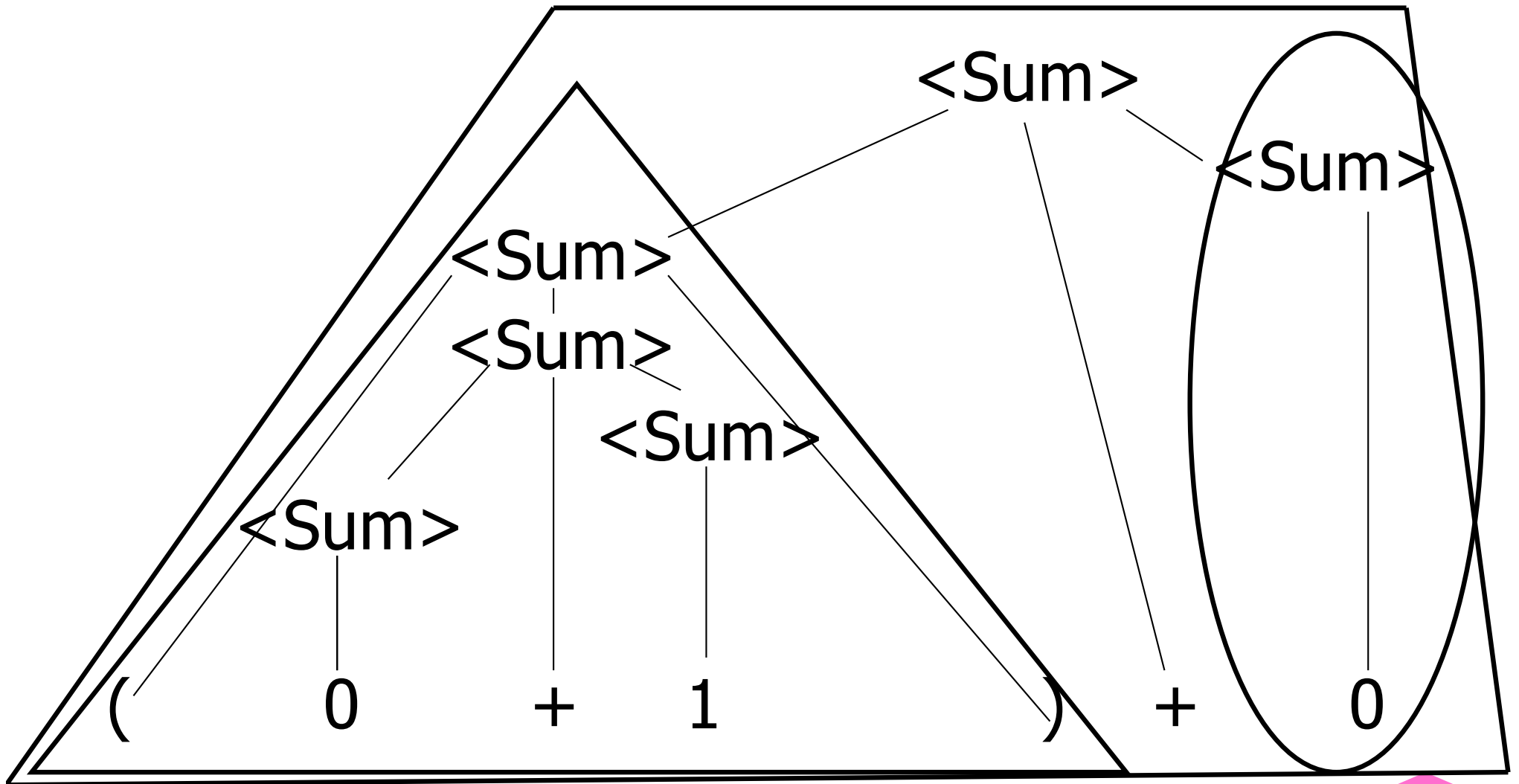


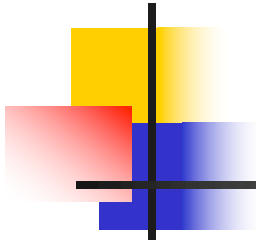
Example



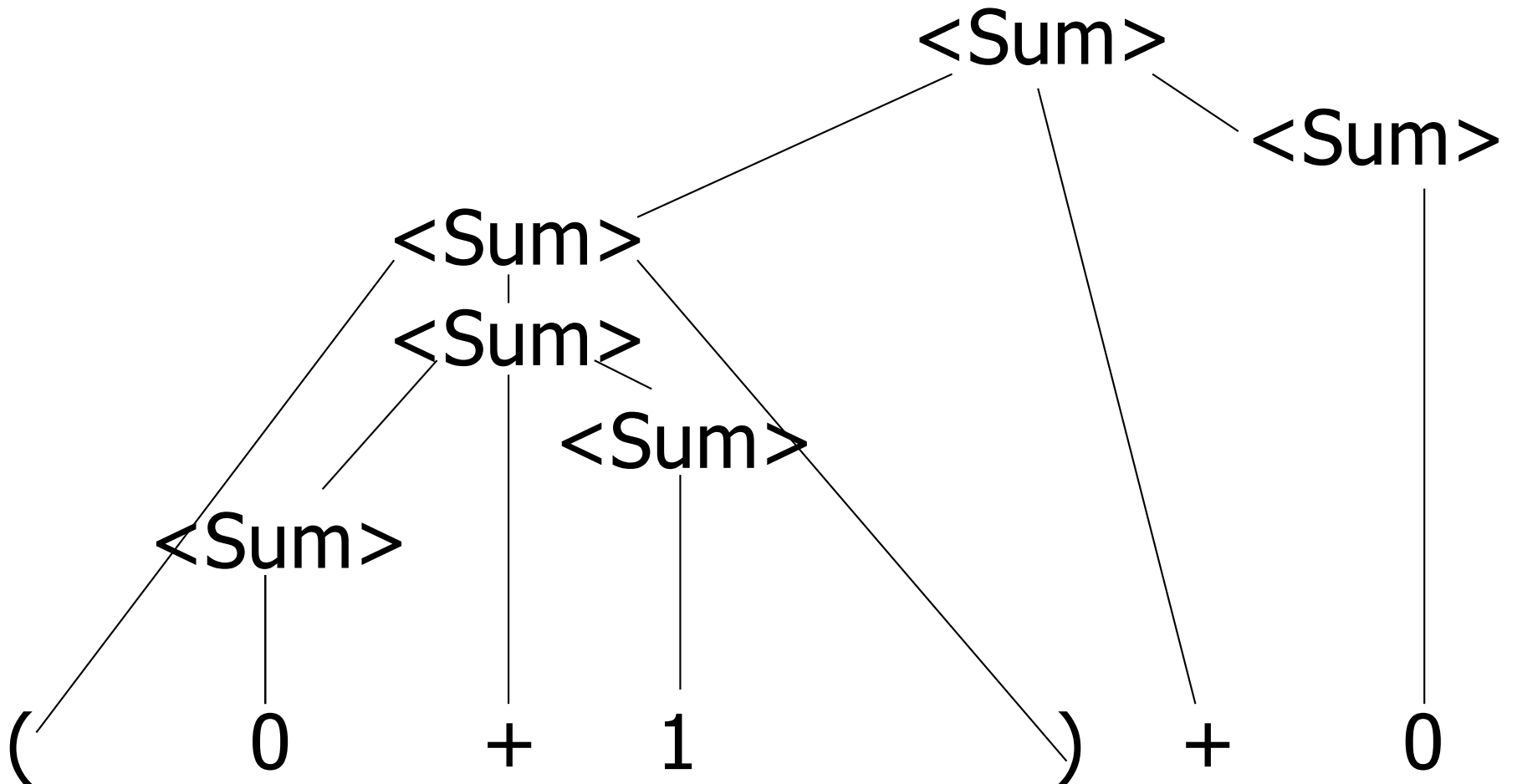


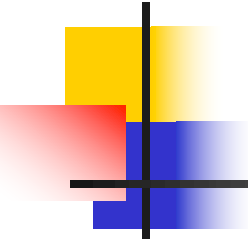
Example





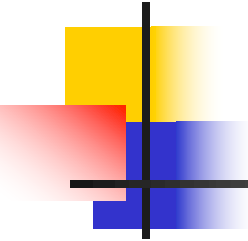
Example





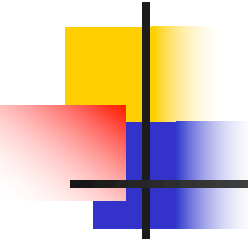
LR Parsing Tables

- 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



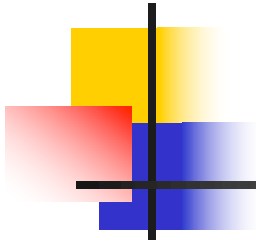
Action and Goto Tables

- 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



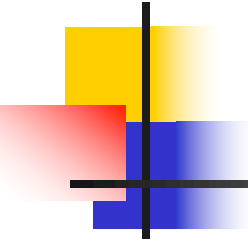
LR(i) Parsing Algorithm

- 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

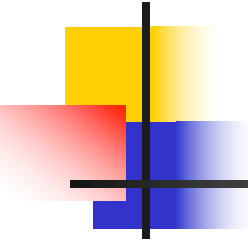
0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push **state**(1) onto stack
- 3. Look at next i tokens from token stream ($toks$) (don't remove yet)
4. If top symbol on stack is **state**(n), look up action in Action table at $(n, toks)$



LR(i) Parsing Algorithm

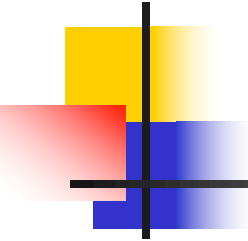
5. If action = **shift** m ,

- 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



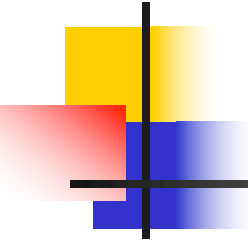
LR(i) Parsing Algorithm

7. If action = **accept**

- Stop parsing, return success

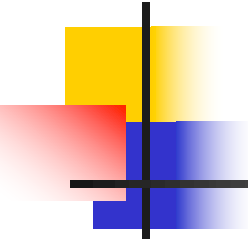
8. If action = **error**,

- Stop parsing, return failure



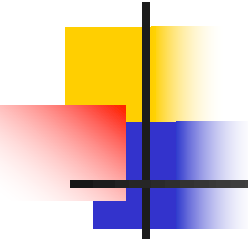
Adding Synthesized Attributes

- 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



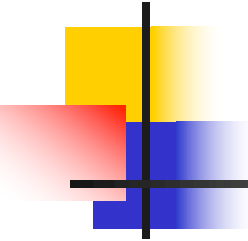
Shift-Reduce Conflicts

- **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



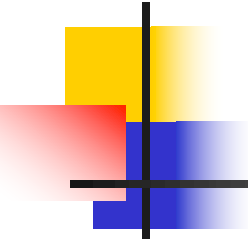
Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$
 $\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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



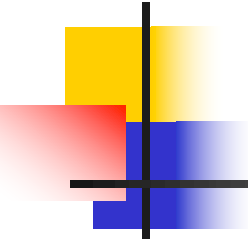
Example - cont

- **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

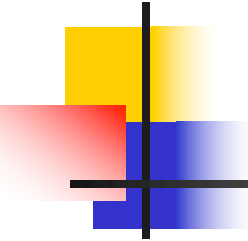


Example

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

● abc	shift
a ● bc	shift
ab ● c	shift
abc ●	

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



Programming Languages & Compilers

Three Main Topics of the Course

I

New
Programming
Paradigm

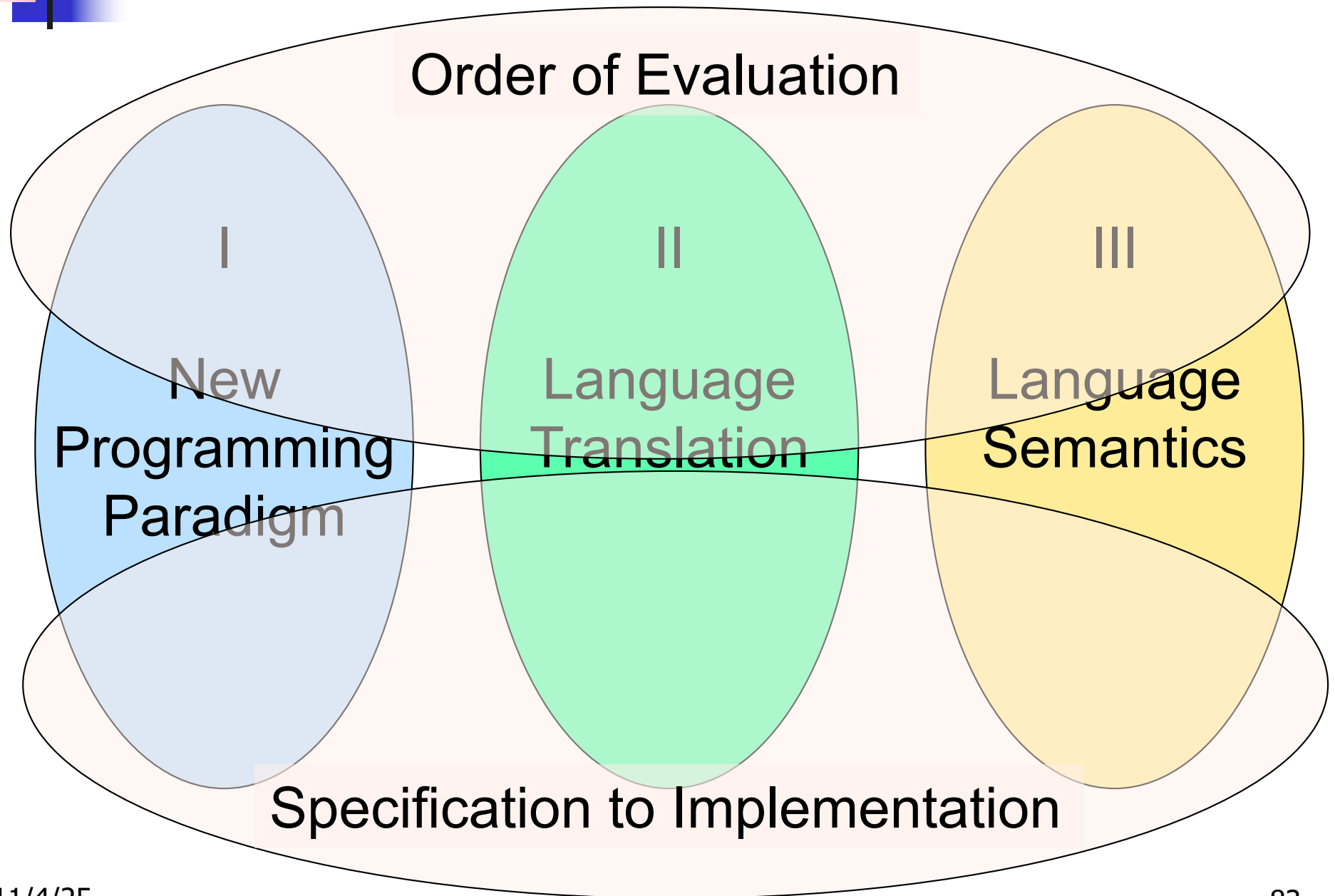
II

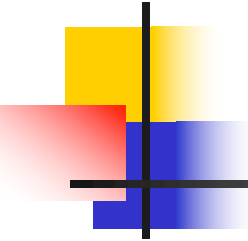
Language
Translation

III

Language
Semantics

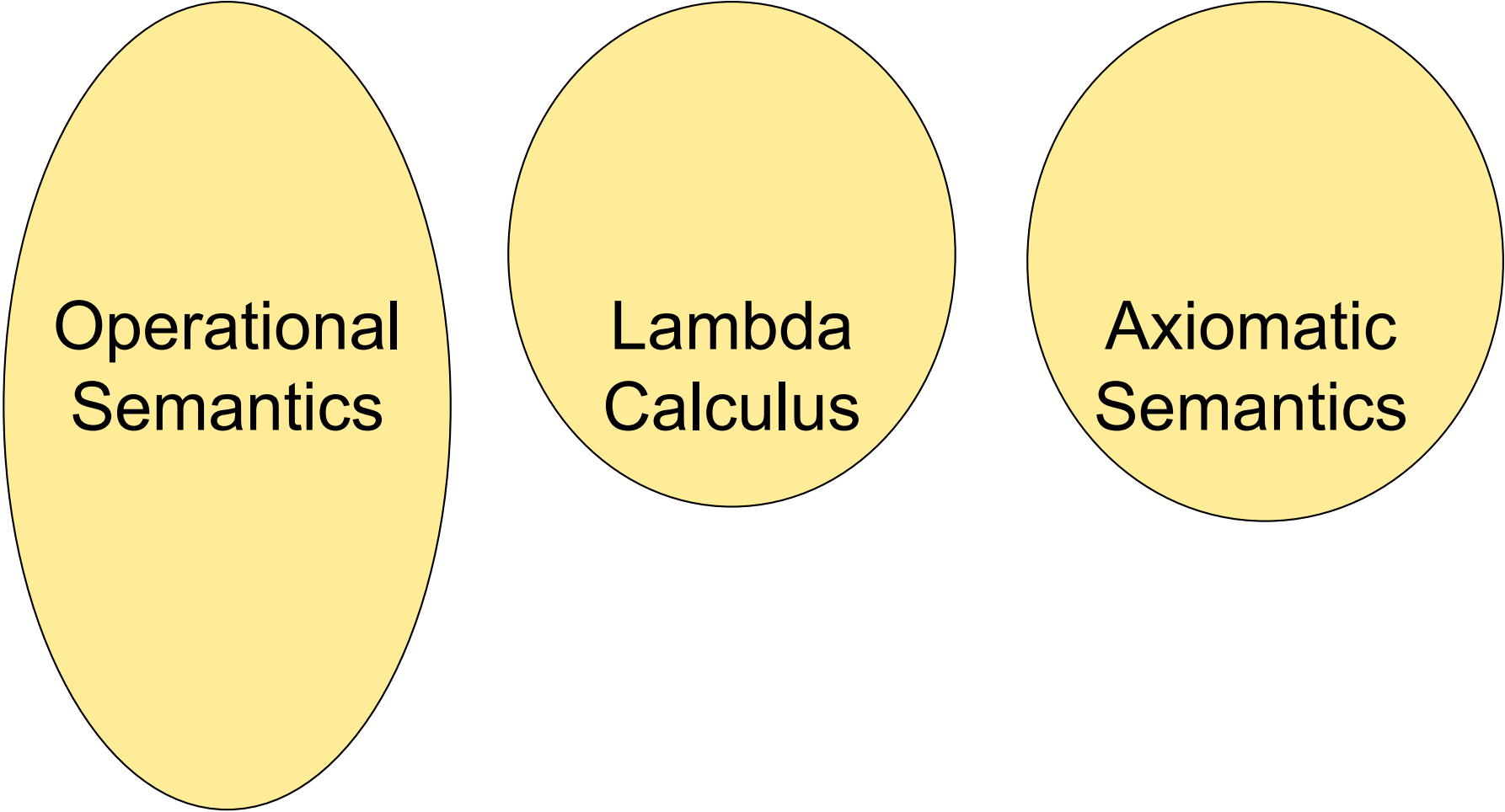
Programming Languages & Compilers





Programming Languages & Compilers

III : Language Semantics

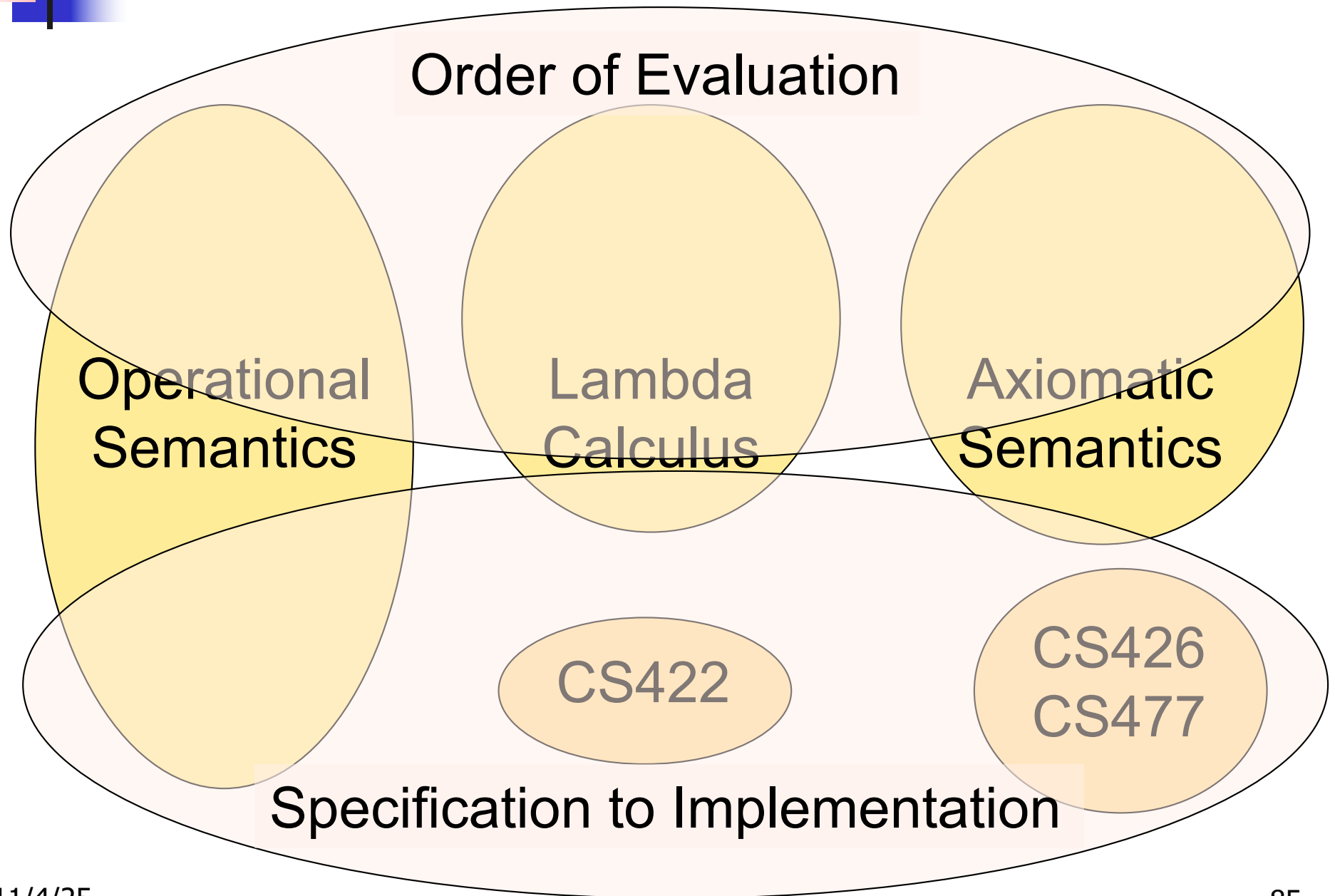


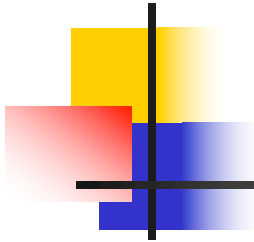
Operational
Semantics

Lambda
Calculus

Axiomatic
Semantics

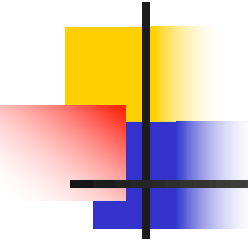
Programming Languages & Compilers





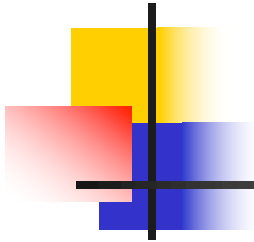
Semantics

- 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



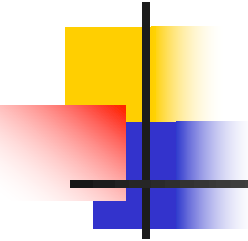
Dynamic semantics

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



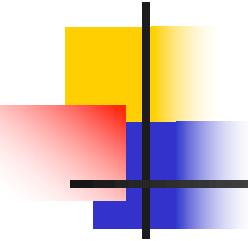
Dynamic Semantics

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



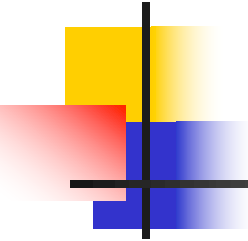
Operational Semantics

- 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



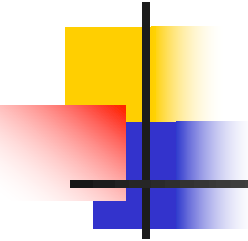
Axiomatic Semantics

- 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



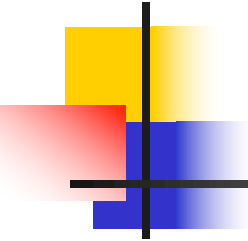
Axiomatic Semantics

- 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



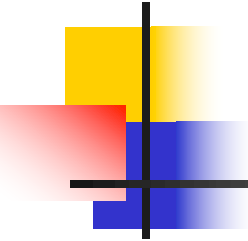
Natural Semantics

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

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

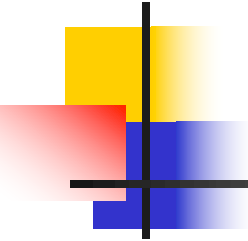
or

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



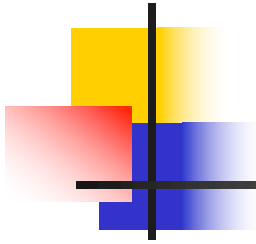
Simple Imperative Programming Language

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



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:

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

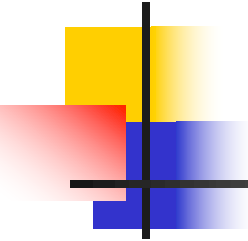
$$\frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \ \& \ B', m) \Downarrow b}$$

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

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

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

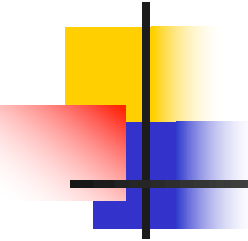
$$\frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$



Relations

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

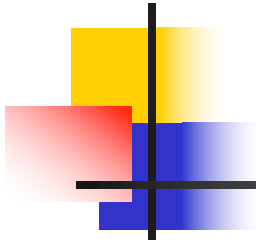
- By $U \sim V = b$, we mean does (the meaning of) the relation $\sim$ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V



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 $U \text{ op } V$

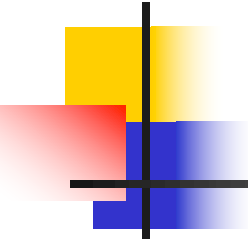


Commands

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

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

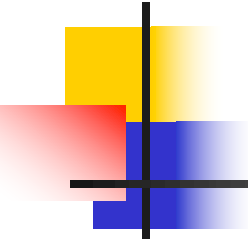
Sequencing:
$$\frac{(C, m) \Downarrow m' \quad (C', m') \Downarrow m''}{(C; C', m) \Downarrow m''}$$



If Then Else Command

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

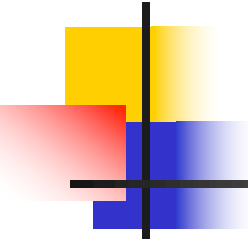
$$\frac{(B, m) \Downarrow \text{false} \quad (C', m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$



While Command

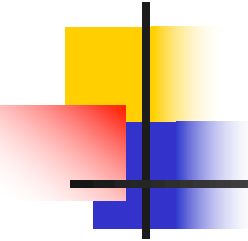
$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

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



Example: If Then Else Rule

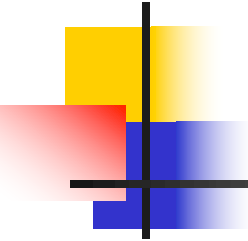
(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}) \Downarrow ?$



Example: If Then Else Rule

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

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



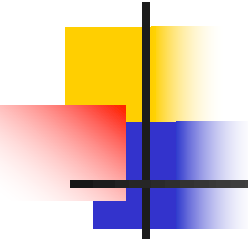
Example: Arith Relation

$? > ? = ?$

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

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

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



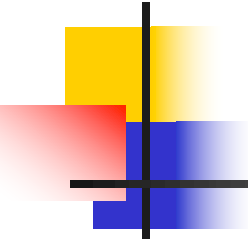
Example: Identifier(s)

$7 > 5 = \text{true}$

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

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

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



Example: Arith Relation

$7 > 5 = \text{true}$

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

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

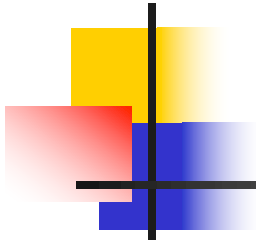
$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \\ \{x \rightarrow 7\}) \Downarrow ?$

Example: If Then Else Rule

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

Example: Assignment

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



Example: Arith Op

$$\begin{array}{rcl} & & ? + ? = ? \\ & & \underline{(2, \{x \rightarrow 7\}) \Downarrow? \quad (3, \{x \rightarrow 7\}) \Downarrow?} \\ & & (2+3, \{x \rightarrow 7\}) \Downarrow? \\ & & \underline{(y := 2 + 3, \{x \rightarrow 7\})} \\ & & \Downarrow? \\ & & . \\ \begin{array}{rcl} & & 7 > 5 = \text{true} \\ \underline{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5} & & \\ (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} & & \\ \hline \end{array} & & \end{array}$$

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

Example: Numerals

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

Example: Arith Op

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

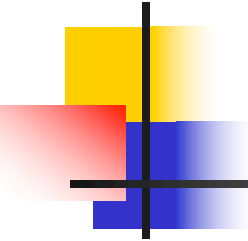
Example: Assignment

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

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

Example: If Then Else Rule

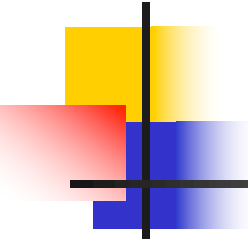
$$\begin{array}{c}
 \begin{array}{c}
 7 > 5 = \text{true} \\
 \hline
 (x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \\
 \hline
 (x > 5, \{x \rightarrow 7\}) \Downarrow \text{true} \\
 \hline
 \text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{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



Let in Command

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

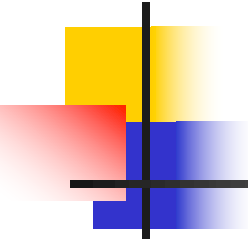
Where $m''(y) = m'(y)$ for $y \neq I$ and
 $m''(I) = m(I)$ if $m(I)$ is defined,
and $m''(I)$ is undefined otherwise



Example

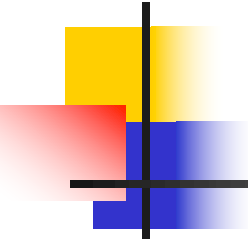
$$\frac{\frac{(x, \{x \rightarrow 5\}) \Downarrow 5 \quad (3, \{x \rightarrow 5\}) \Downarrow 3}{(x+3, \{x \rightarrow 5\}) \Downarrow 8}}{(5, \{x \rightarrow 17\}) \Downarrow 5 \quad (x := x+3, \{x \rightarrow 5\}) \Downarrow \{x \rightarrow 8\}} \Downarrow ?$$

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



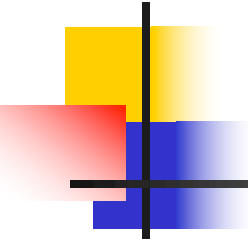
Example

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



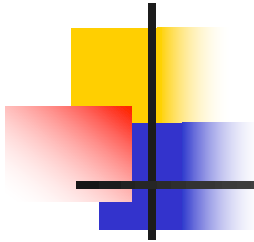
Comment

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



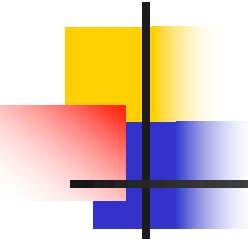
Interpretation Versus Compilation

- 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



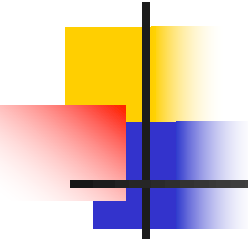
Interpreter

- 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



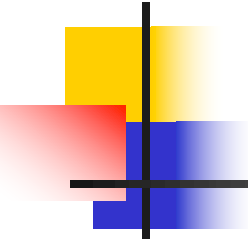
Interpreter

- 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 compute_com
 $(\text{While}(b,c), \text{compute_com}(c,m))$
- May fail to terminate - exceed stack limits
- Returns no useful information then