

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



Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams

- Finite state automata
- Pushdown automata
- Whole family more of grammars and automata – covered in automata theory



BNF Grammars

- Start with a set of characters, **a,b,c,...**
 - We call these *terminals*
- Add a set of different characters, **X,Y,Z,...**
 - We call these *nonterminals*
- One special nonterminal **S** called *start symbol*



Sample Grammar

- Language: Parenthesized sums of 0's and 1's
- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$



BNF Grammars

- BNF rules (aka *productions*) have form

$$\mathbf{X} ::= y$$

where \mathbf{X} is any nonterminal and y is a string of terminals and nonterminals

- BNF *grammar* is a set of BNF rules such that every nonterminal appears on the left of some rule



Sample Grammar

- Terminals: 0 1 + ()
- Nonterminals: $\langle \text{Sum} \rangle$
- Start symbol = $\langle \text{Sum} \rangle$

- $\langle \text{Sum} \rangle ::= 0$
- $\langle \text{Sum} \rangle ::= 1$
- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$
- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$
- Can be abbreviated as
$$\langle \text{Sum} \rangle ::= 0 \mid 1$$
$$\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \mid (\langle \text{Sum} \rangle)$$



BNF Derivations

- Given rules

$$\mathbf{X} ::= y\mathbf{Z}w \text{ and } \mathbf{Z} ::= v$$

we may replace \mathbf{Z} by v to say

$$\mathbf{X} \Rightarrow y\mathbf{Z}w \Rightarrow yvw$$

- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal



BNF Derivations

- Start with the start symbol:

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



BNF Derivations

- Pick a non-terminal

<Sum> =>

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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



BNF Derivations

- Pick a non-terminal:

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



BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$

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

$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$



BNF Derivations

- Pick a non-terminal:

$$\begin{aligned}\langle \text{Sum} \rangle & \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle\end{aligned}$$

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$



BNF Derivations

- Pick a non-terminal:

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 1$

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$



BNF Derivations

- Pick a non-terminal:

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

$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$

$\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$

BNF Derivations

- Pick a rule and substitute:

- $\langle \text{Sum} \rangle ::= 0$

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

$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$$

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



BNF Derivations

- Pick a non-terminal:

$$\begin{aligned} \langle \text{Sum} \rangle & \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + 0 \end{aligned}$$

BNF Derivations

- Pick a rule and substitute

- $\langle \text{Sum} \rangle ::= 0$

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

$$\Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$\Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$$

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

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



BNF Derivations

- $(0 + 1) + 0$ is generated by grammar

$$\begin{aligned}\langle \text{Sum} \rangle & \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle \\ & \Rightarrow (\langle \text{Sum} \rangle + 1) + 0 \\ & \Rightarrow (0 + 1) + 0\end{aligned}$$



Extended BNF Grammars

- Alternatives: allow rules of form $X ::= y/z$
 - Abbreviates $X ::= y, X ::= z$
- Options: $X ::= y[v]z$
 - Abbreviates $X ::= yvz, X ::= yz$
- Repetition: $X ::= y\{v\}^*z$
 - Can be eliminated by adding new nonterminal V and rules $X ::= yz, X ::= yVz, V ::= v, V ::= w$



Parse Trees

- Graphical representation of derivation
- Each node labeled with either non-terminal or terminal
- If node is labeled with a terminal, then it is a leaf (no sub-trees)
- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it



Example

- Consider grammar:

$$\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$$
$$| \langle \text{factor} \rangle + \langle \text{factor} \rangle$$
$$\langle \text{factor} \rangle ::= \langle \text{bin} \rangle$$
$$| \langle \text{bin} \rangle * \langle \text{exp} \rangle$$
$$\langle \text{bin} \rangle ::= 0 \mid 1$$

- Problem: Build parse tree for $1 * 1 + 0$ as an $\langle \text{exp} \rangle$



Example cont.

- $1 * 1 + 0$: $\langle \text{exp} \rangle$

$\langle \text{exp} \rangle$ is the start symbol for this parse tree



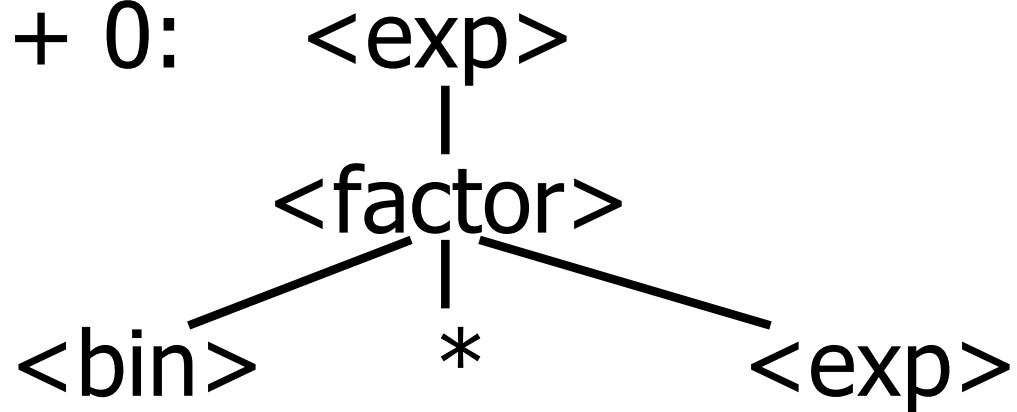
Example cont.

■ $1 * 1 + 0$: $\langle \text{exp} \rangle$
|
 $\langle \text{factor} \rangle$

Use rule: $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$

Example cont.

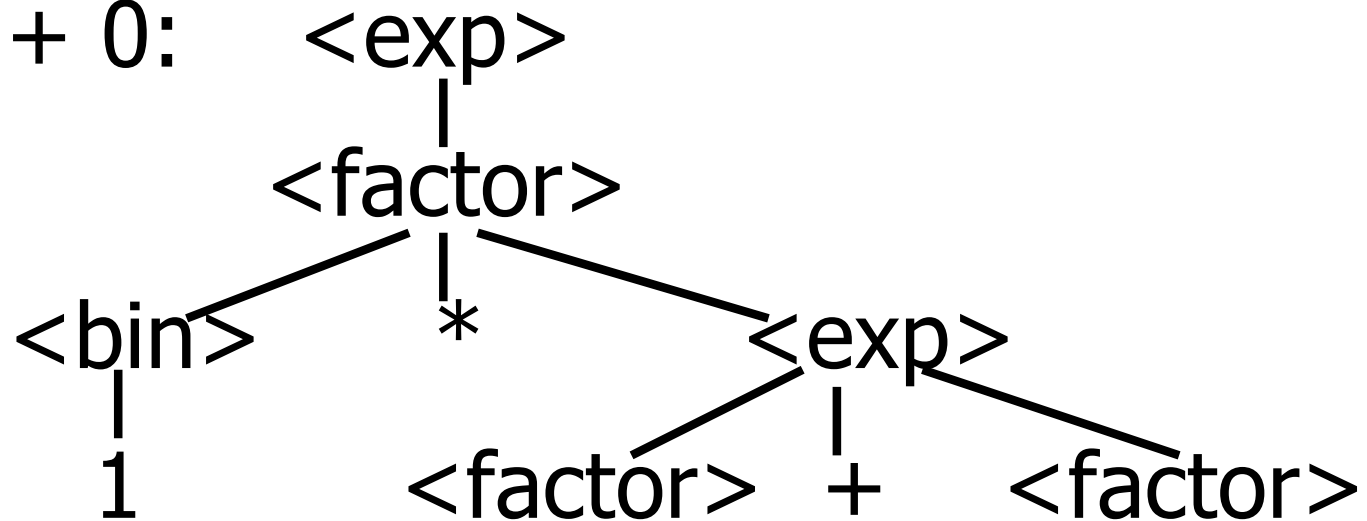
- $1 * 1 + 0$:



Use rule: $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle * \langle \text{exp} \rangle$

Example cont.

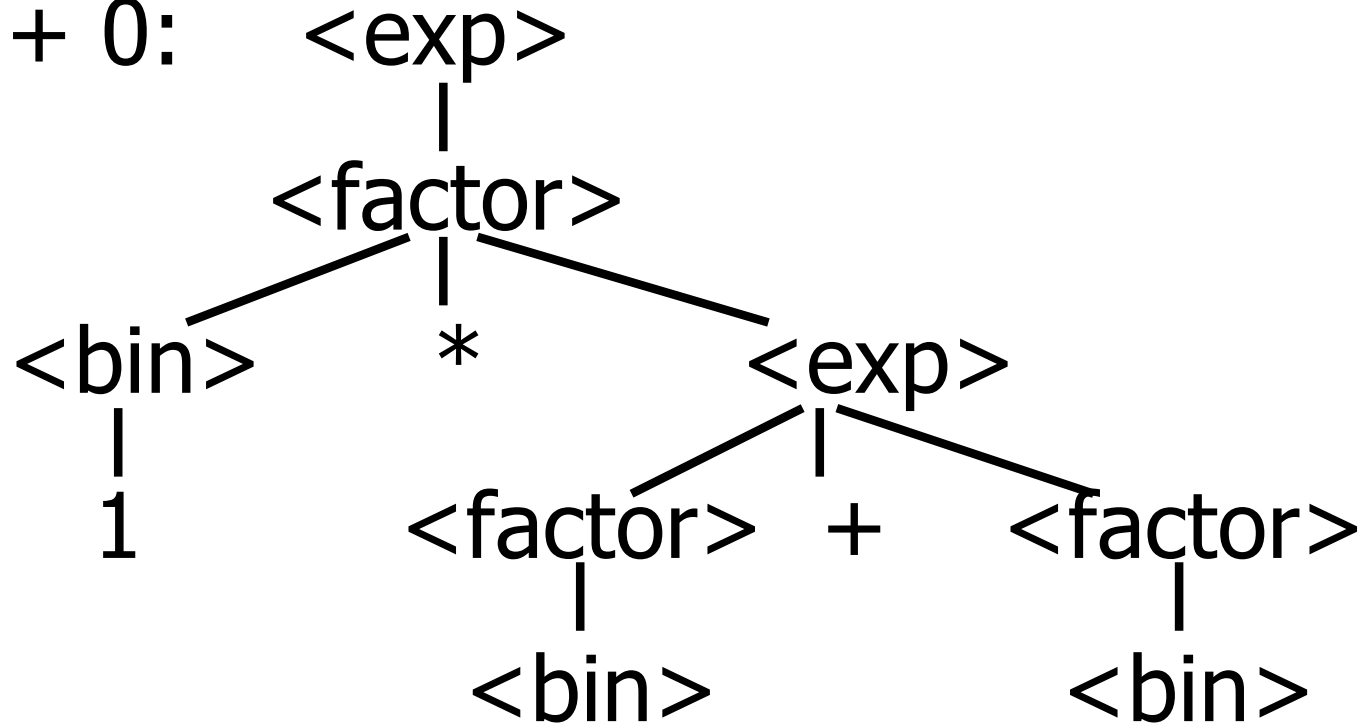
- 1 * 1 + 0:



Use rules: $\langle \text{bin} \rangle ::= 1$ and
 $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle + \langle \text{factor} \rangle$

Example cont.

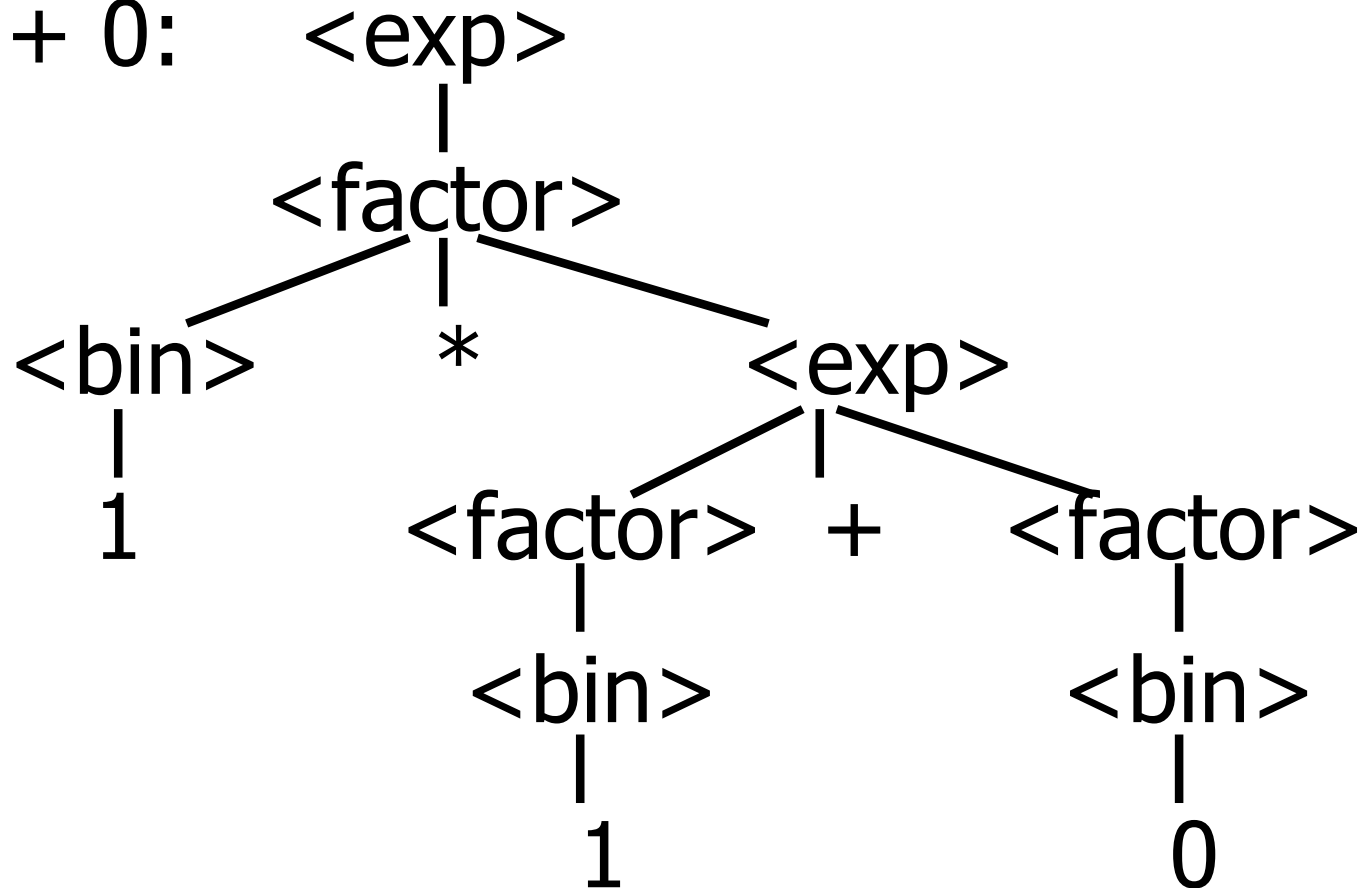
- $1 * 1 + 0$:



Use rule: $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle$

Example cont.

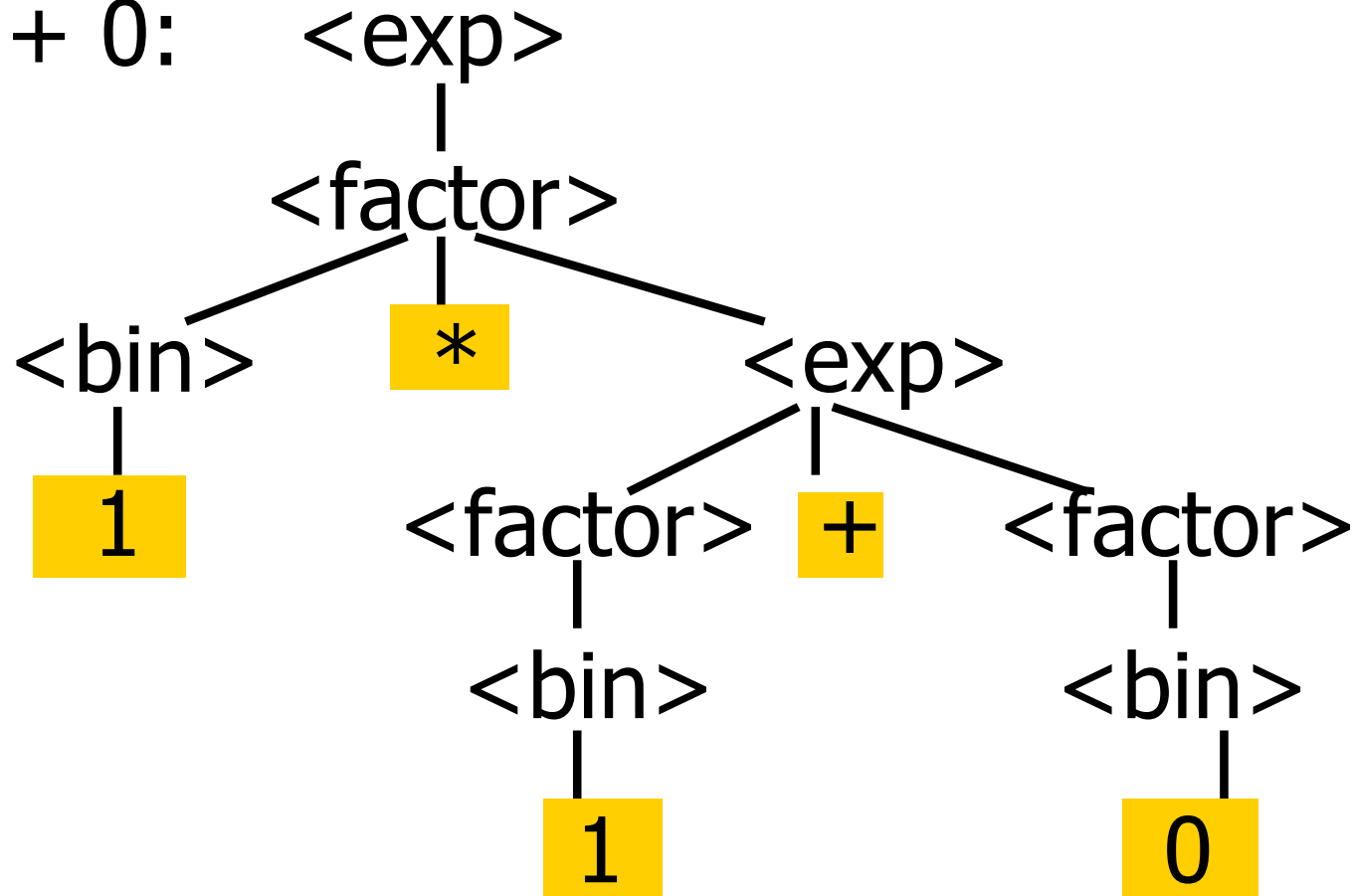
- 1 * 1 + 0:



Use rules: $\langle \text{bin} \rangle ::= 1 \mid 0$

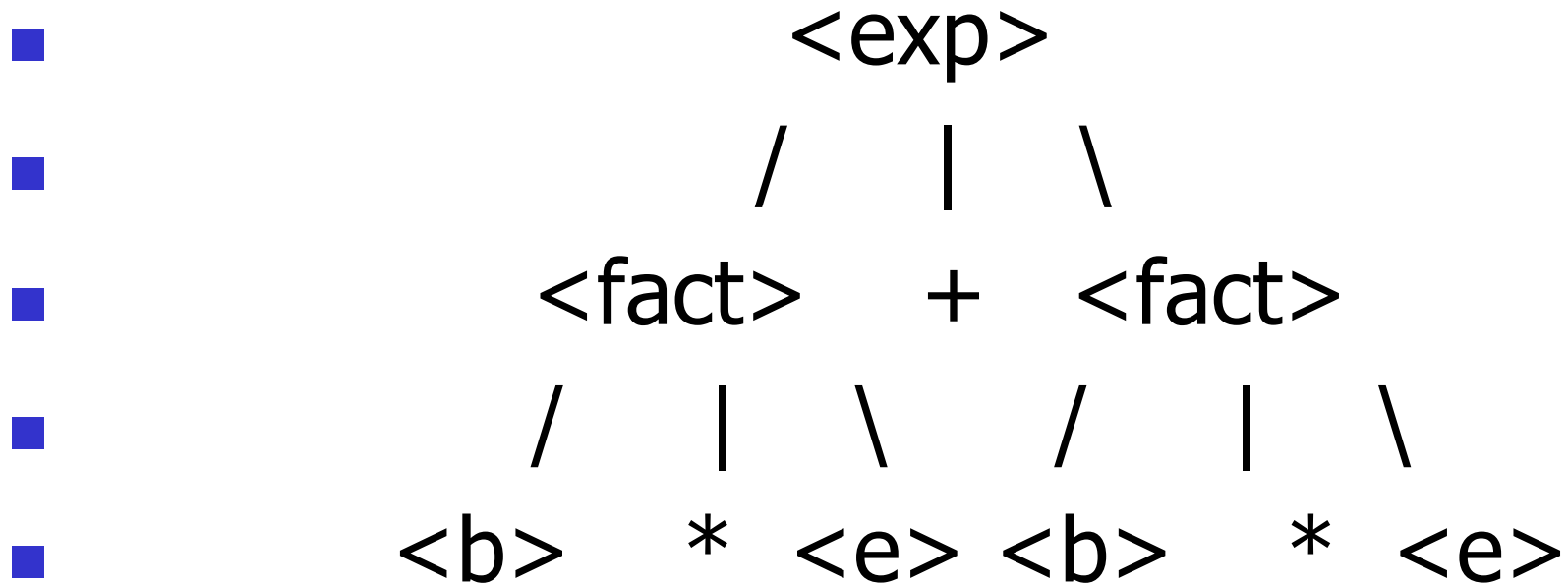
Example cont.

- 1 * 1 + 0:



Fringe of tree is string generated by grammar

Your Turn: $1 * 0 + 0 * 1$





Parse Tree Data Structures

- Parse trees may be represented by OCaml datatypes
- One datatype for each nonterminal
- One constructor for each rule
- Defined as mutually recursive collection of datatype declarations



Example

- Recall grammar:

$\langle \text{exp} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle + \langle \text{factor} \rangle$

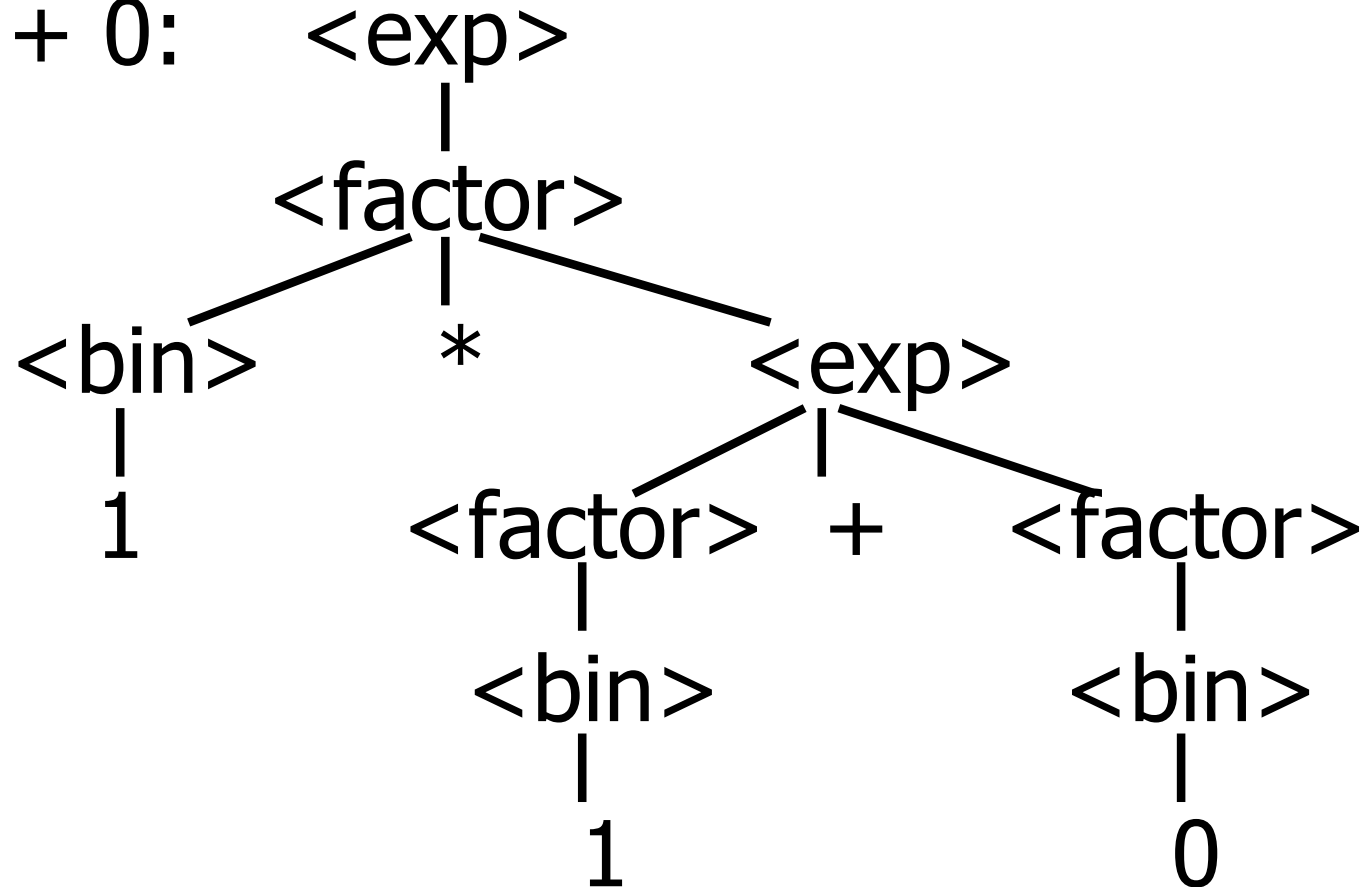
$\langle \text{factor} \rangle ::= \langle \text{bin} \rangle \mid \langle \text{bin} \rangle * \langle \text{exp} \rangle$

$\langle \text{bin} \rangle ::= 0 \mid 1$

- type $\text{exp} = \text{Factor2Exp}$ of factor
 - | Plus of $\text{factor} * \text{factor}$
- and $\text{factor} = \text{Bin2Factor}$ of bin
 - | Mult of $\text{bin} * \text{exp}$
- and $\text{bin} = \text{Zero} \mid \text{One}$

Example cont.

- $1 * 1 + 0$:





Example cont.

- Can be represented as

Factor2Exp

(Mult(One,

Plus(Bin2Factor One,

Bin2Factor Zero)))

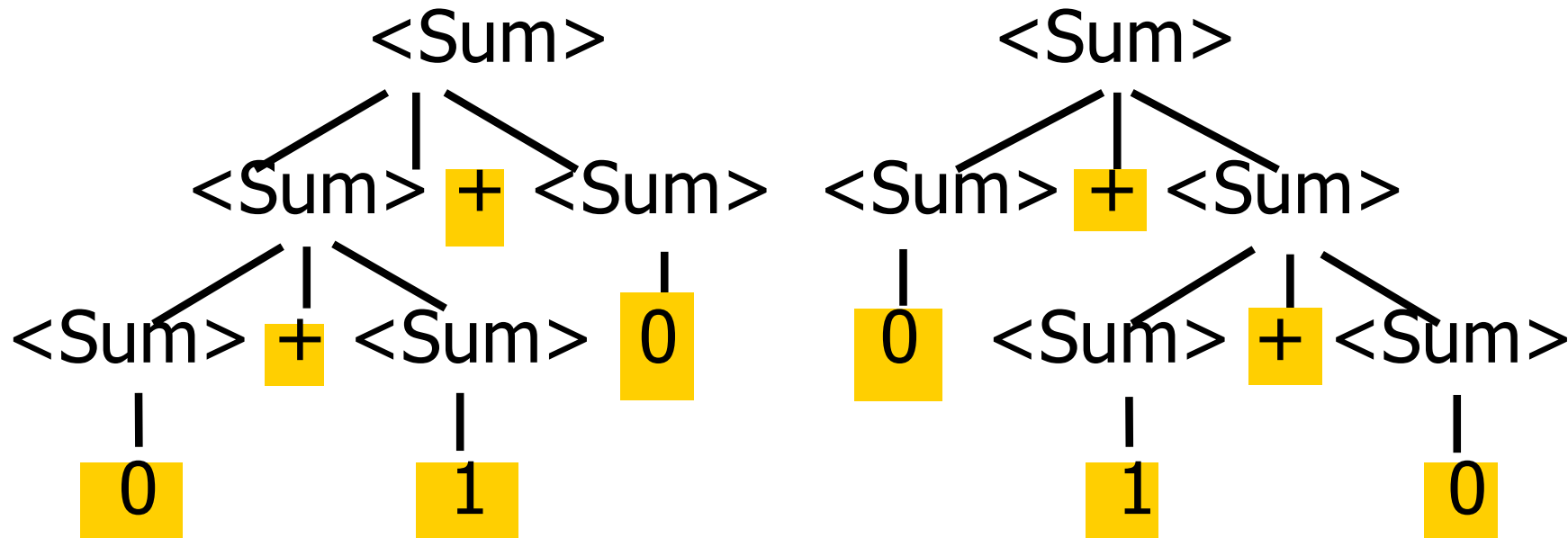


Ambiguous Grammars and Languages

- 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 result for:

$$3 + 4 * 5 + 6$$



Example

- What is the result for:

$$3 + 4 * 5 + 6$$

- Possible answers:

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

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

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

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



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

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



Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity



Disambiguating a Grammar

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



Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
- 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



Example

- Ambiguous grammar:

$$\langle \text{exp} \rangle ::= 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \\ \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle$$

- String with more than one parse:

$$0 + 1 + 0$$
$$1 * 1 + 1$$

- Source of ambiguity: associativity and precedence



Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity



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

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



First Example Again

- In any above language, $3 + 4 * 5 + 6 = 29$
- In APL, all infix operators have same precedence
 - Thus we still don't know what the value is (handled by associativity)
- How do we handle precedence in grammar?



Precedence in Grammar

- Higher precedence translates to longer derivation chain

- Example:

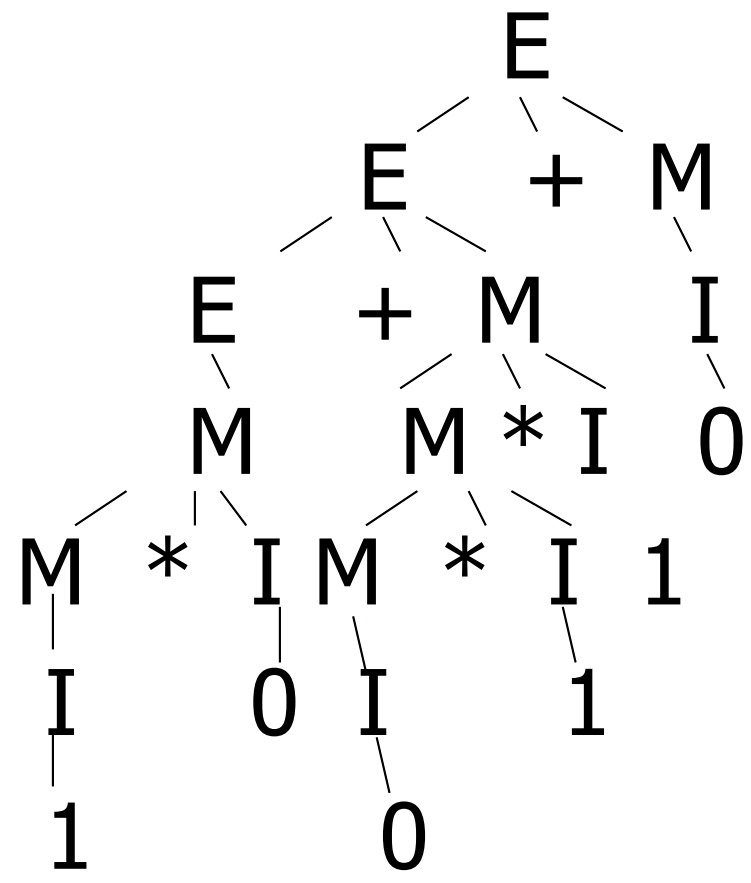
$$\langle \text{exp} \rangle ::= 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \\ \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle$$

- Becomes

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

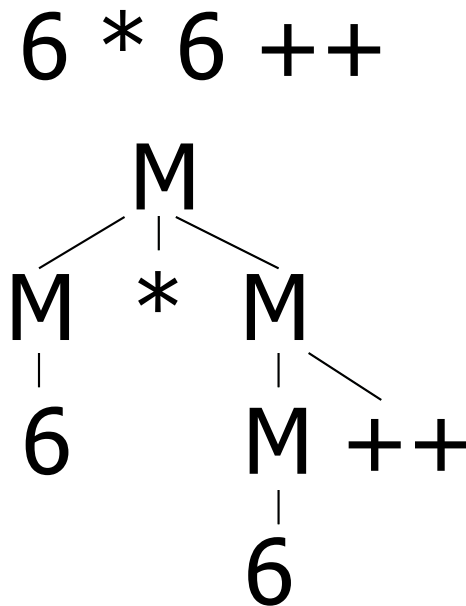
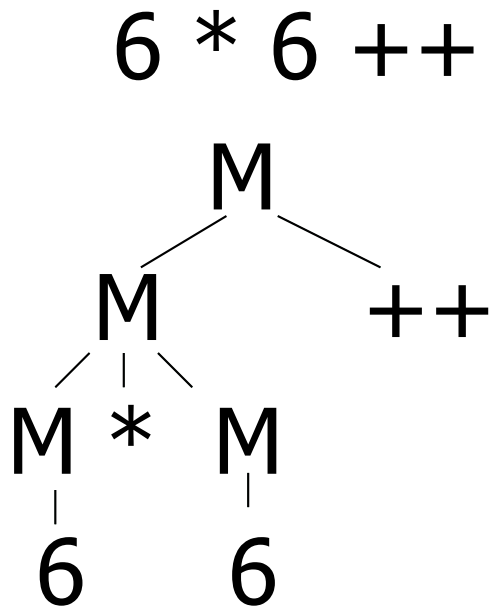


$$1 * 0 + 0 * 1 * 1 + 0$$



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 --- You choose



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

- Think about $6 * 6 ++ * 6 * 6 ++$



Parser Code

- `<grammar>.mly` 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 =
```

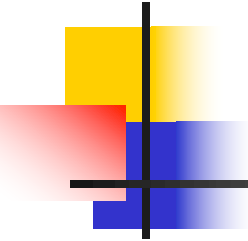
```
Plus_Expr
```

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




LR Parsing

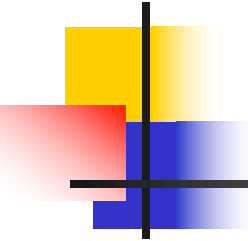
- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there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



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

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

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



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

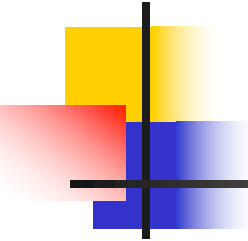
$\langle \text{Sum} \rangle \Rightarrow$

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

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

shift

shift



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

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

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

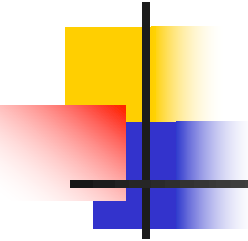
$$= (\bullet 0 + 1) + 0$$

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

reduce

shift

shift



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

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

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

shift

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

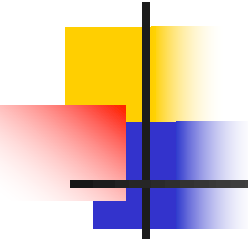
reduce

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

shift

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

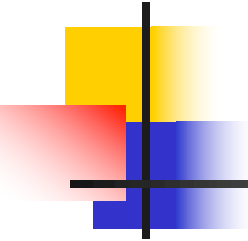
shift



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

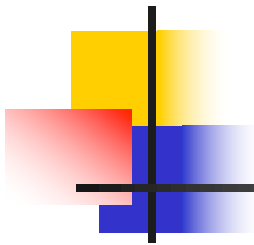
$\langle \text{Sum} \rangle \Rightarrow$
 $\Rightarrow \langle \text{Sum} \rangle + 0 \bullet$ reduce
 $= \langle \text{Sum} \rangle + \bullet 0$ shift
 $= \langle \text{Sum} \rangle \bullet + 0$ shift
 $\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$ reduce
 $= (\langle \text{Sum} \rangle \bullet)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

$$(0 + 1) + 0$$





Example

$$(0 + 1) + 0$$





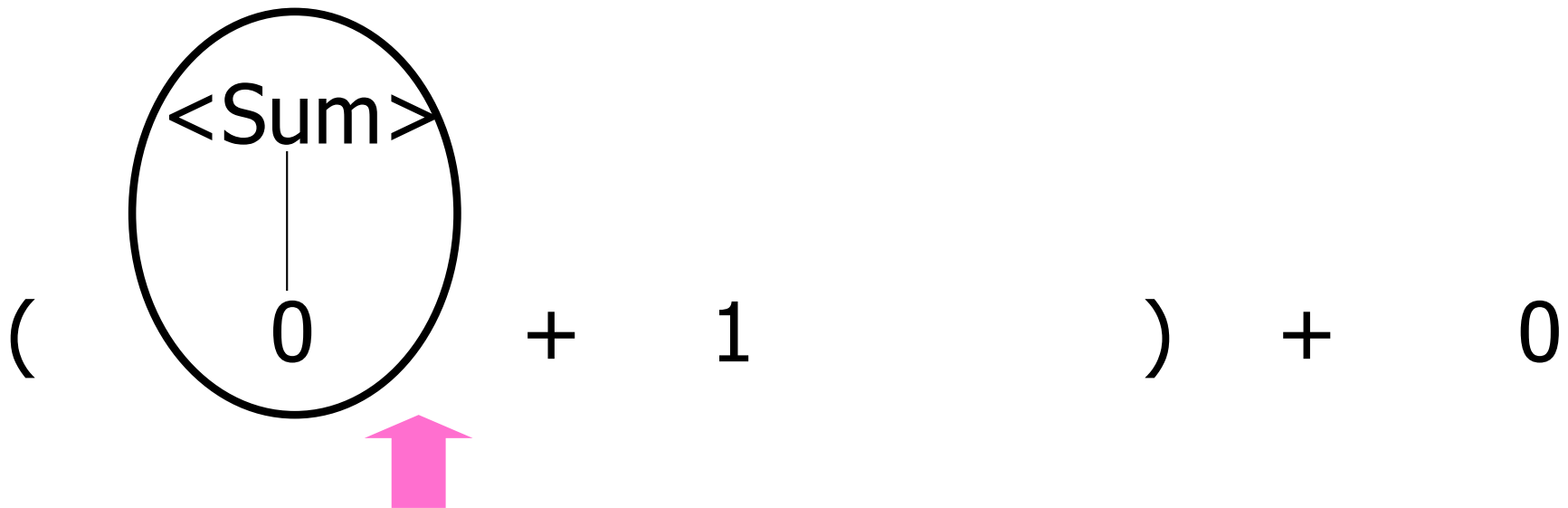
Example

(0 + 1) + 0



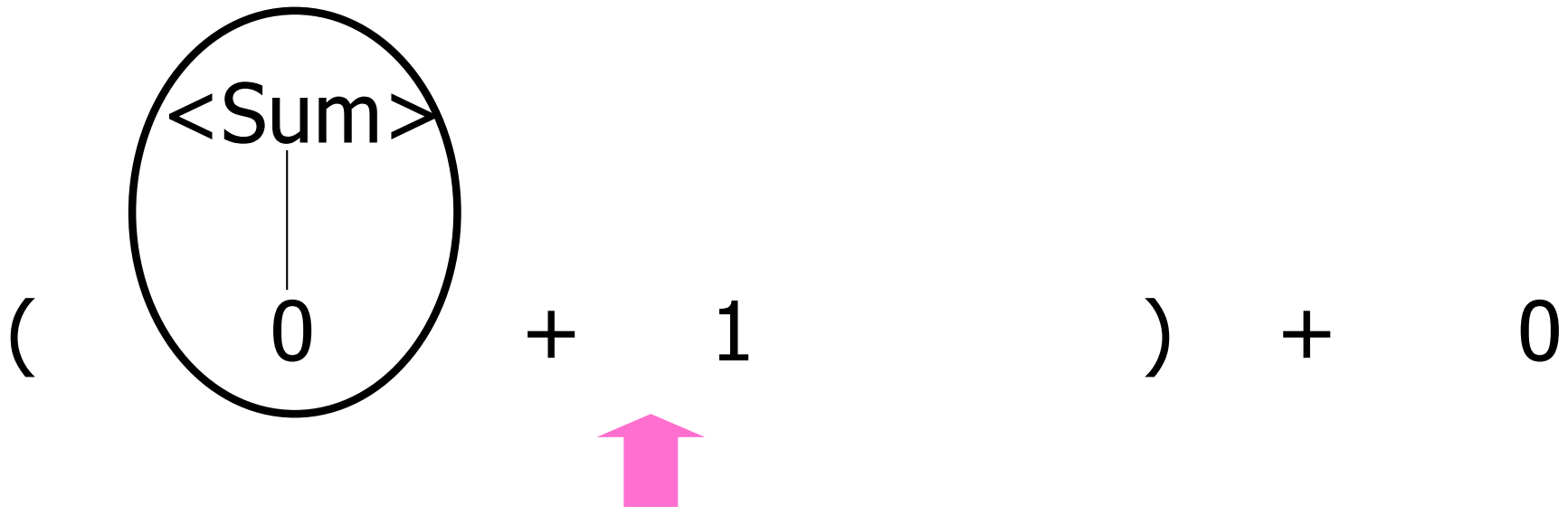


Example





Example



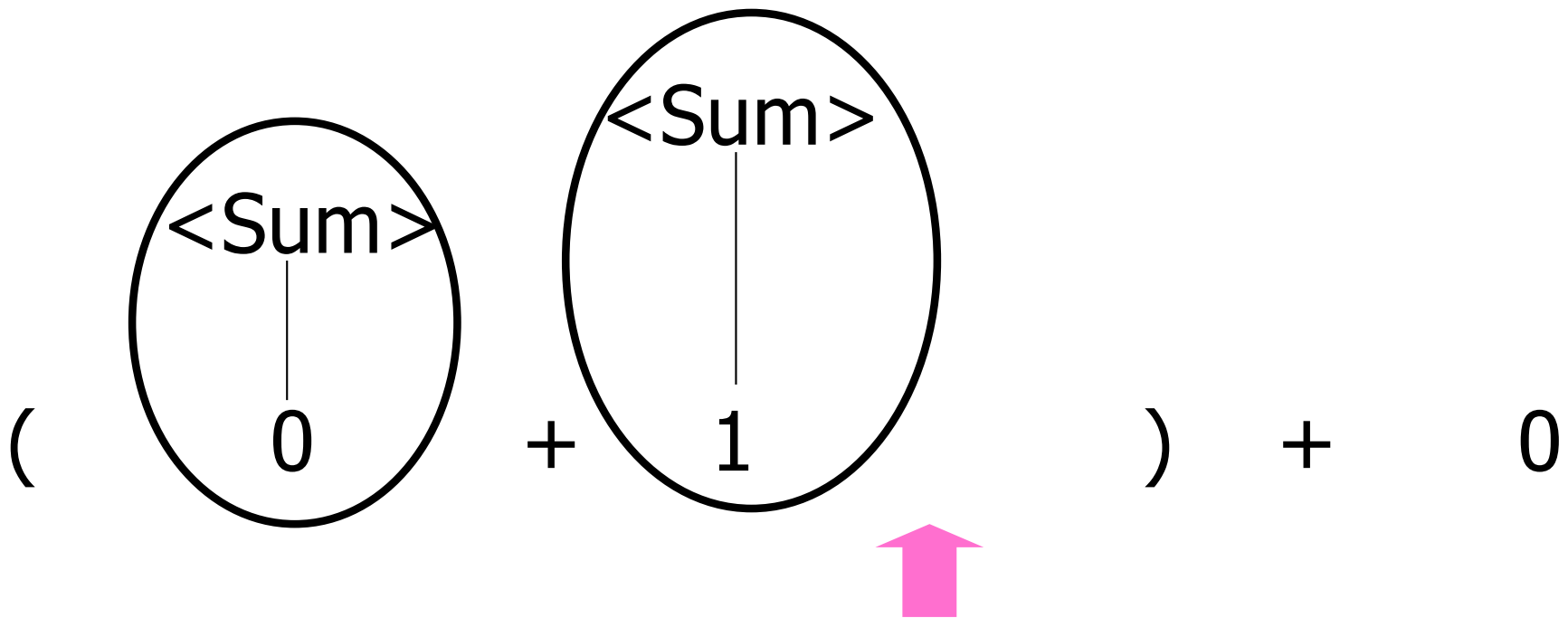


Example



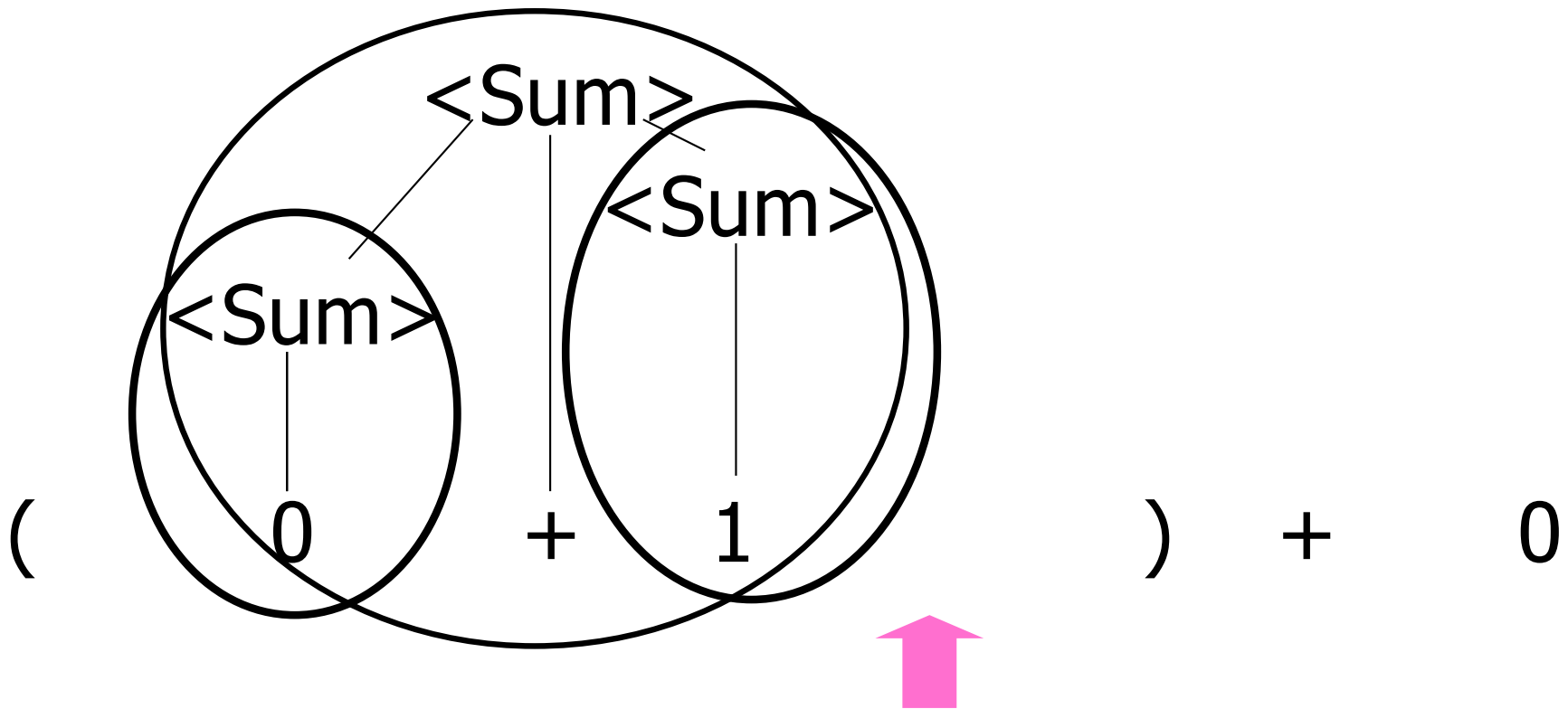


Example



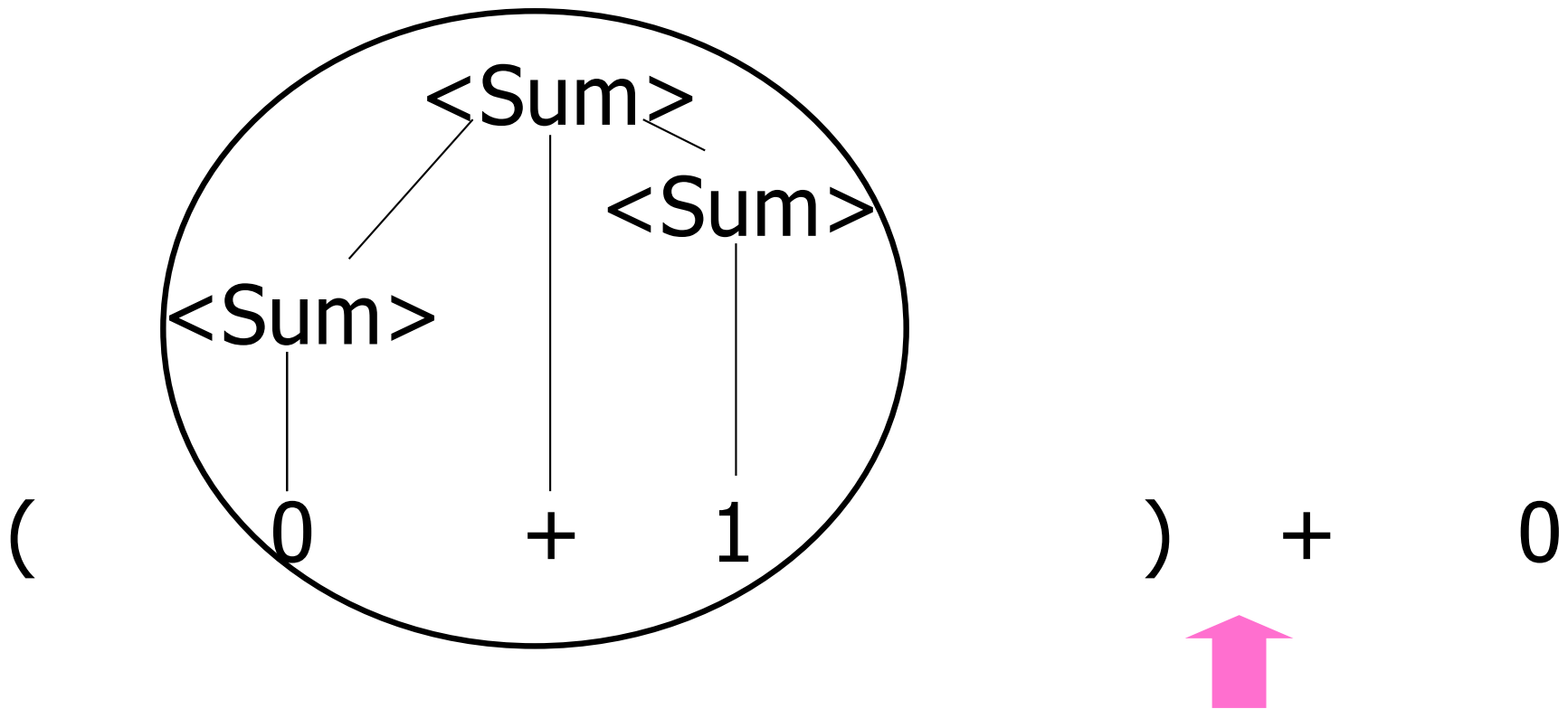


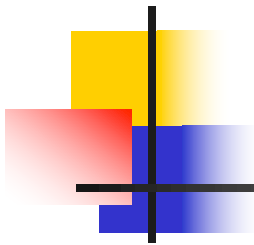
Example



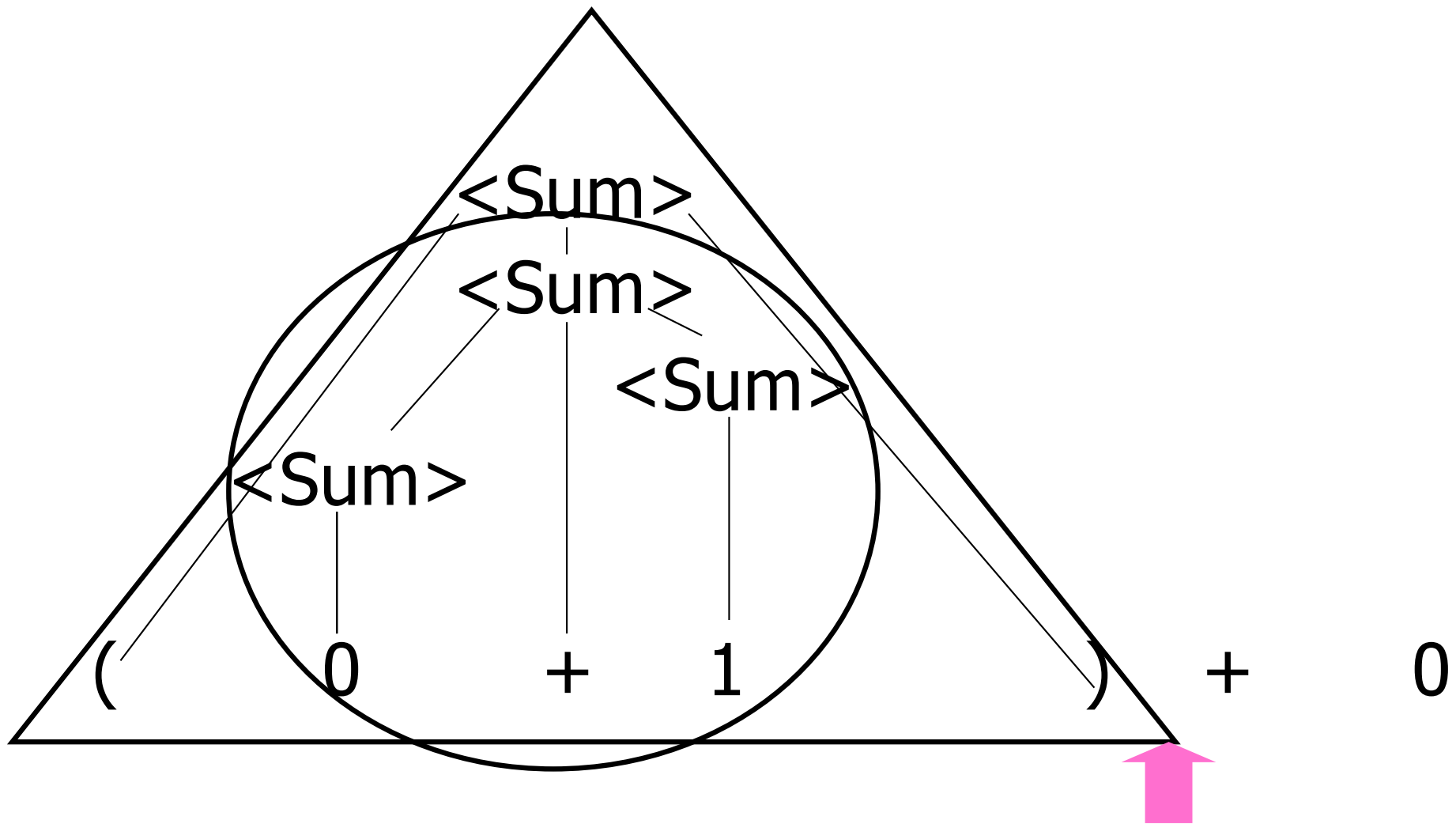


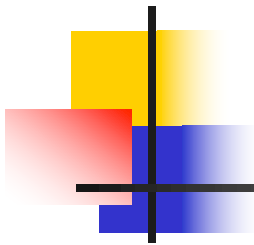
Example



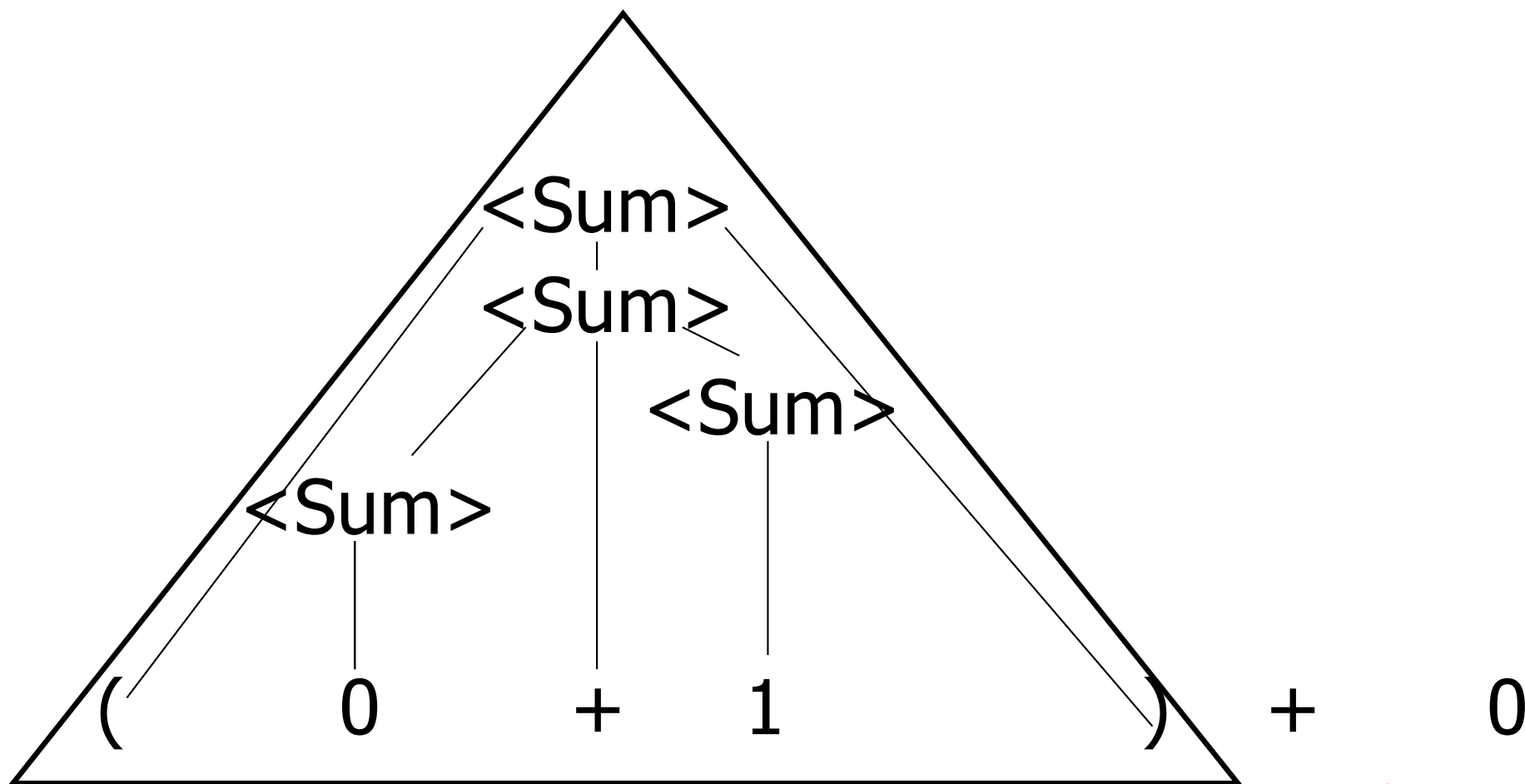


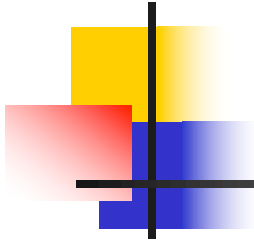
Example



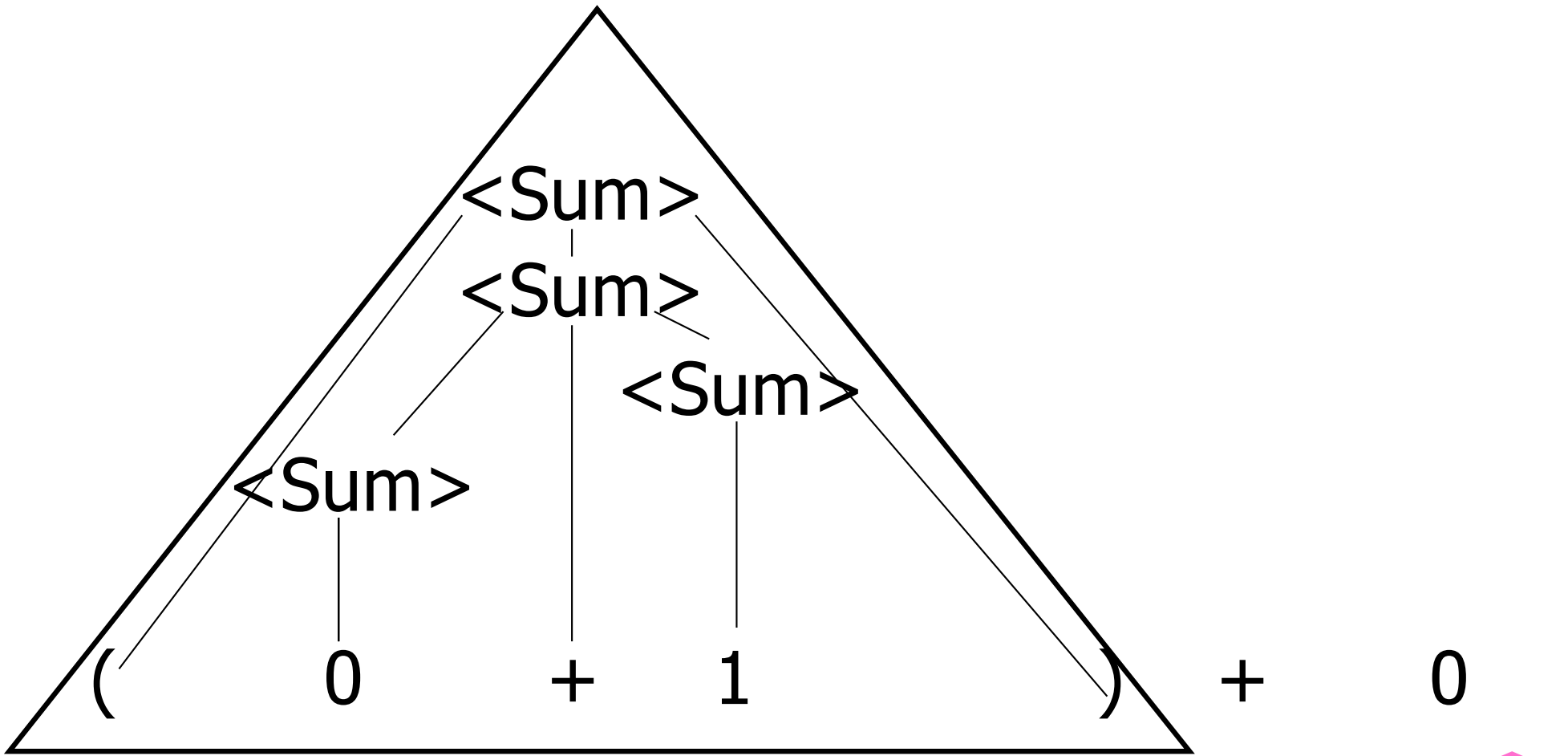


Example



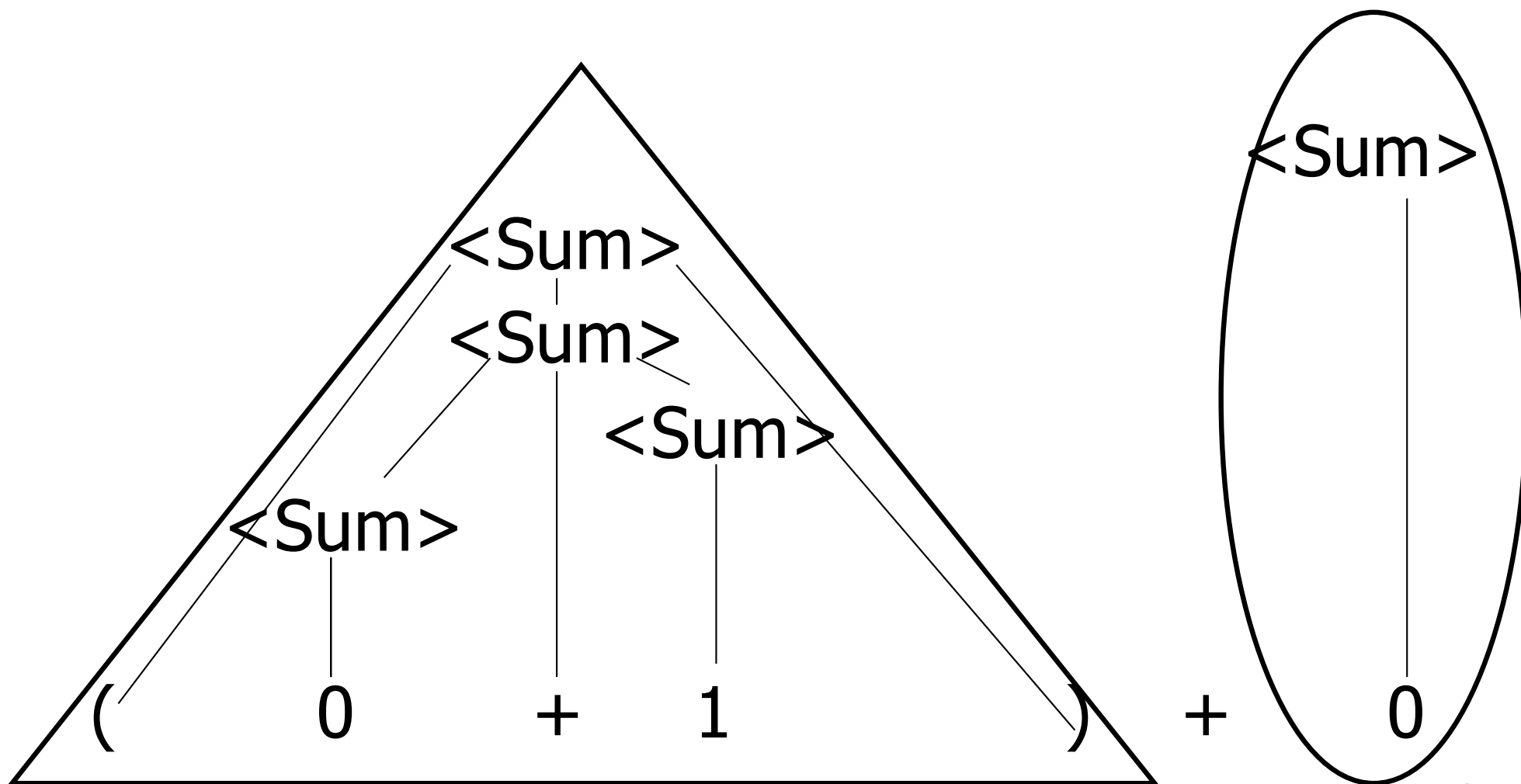


Example



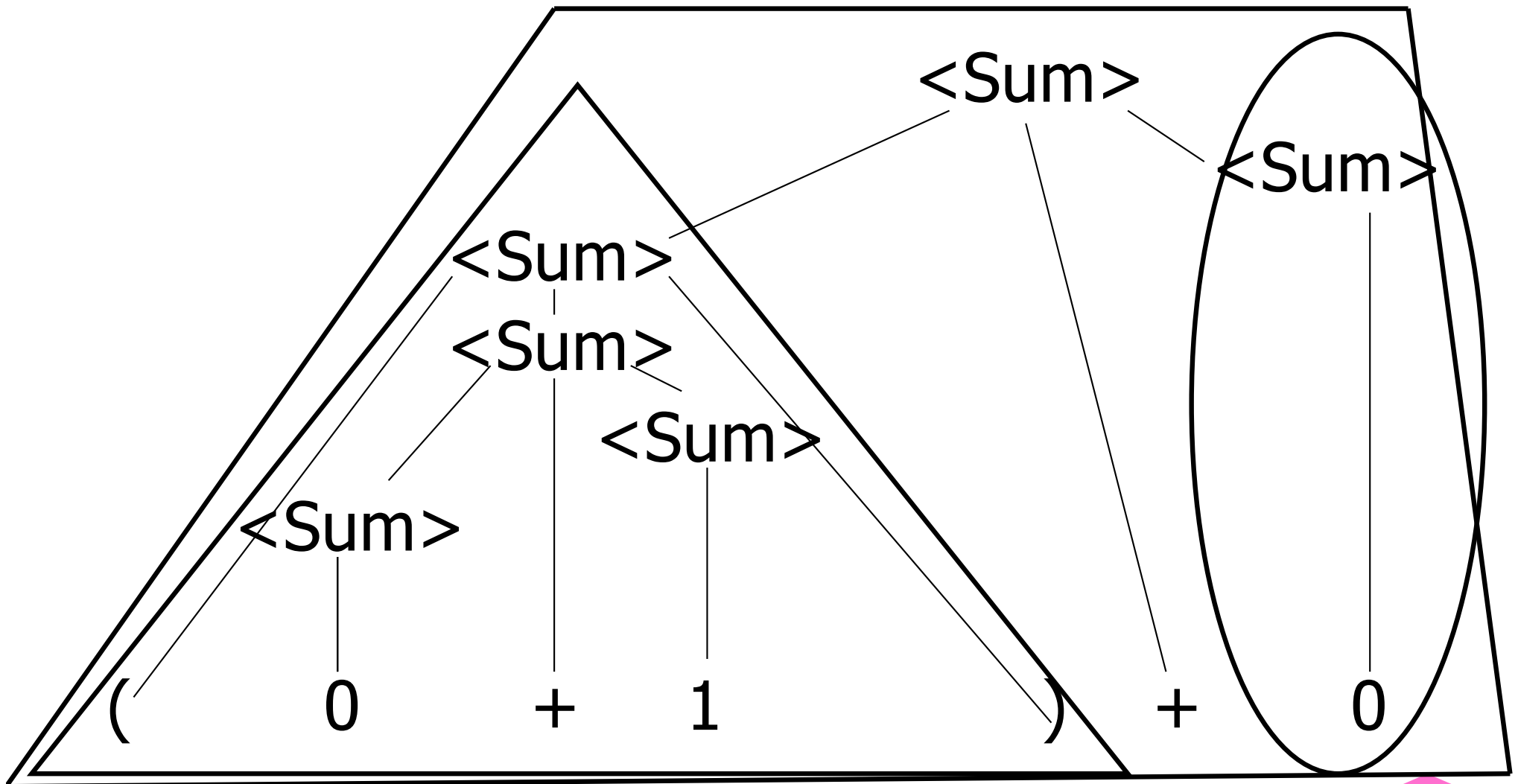


Example



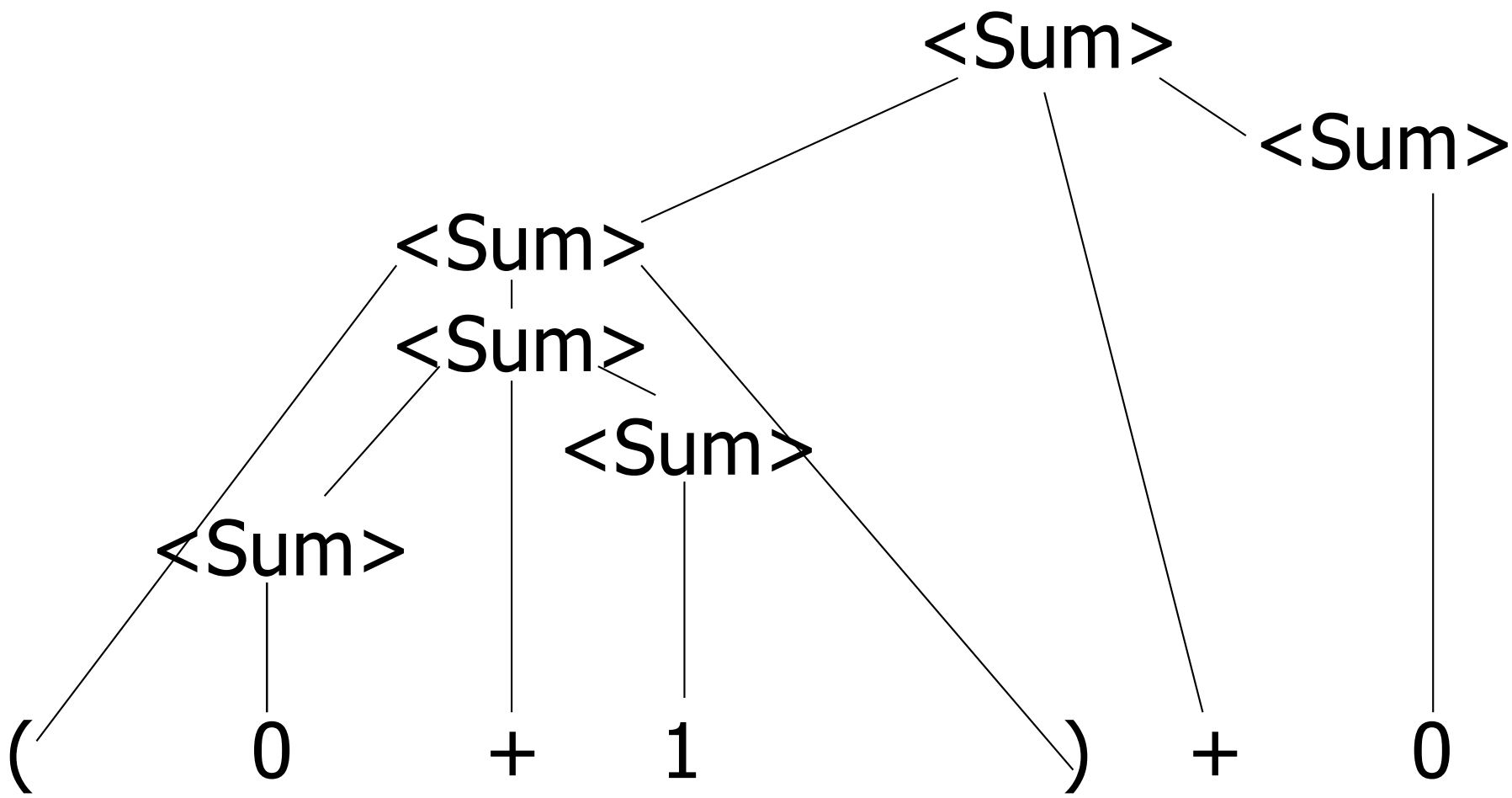


Example





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 $S ::= A$?