Programming Languages and Compilers (CS 421)

Sasa Misailovic
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha

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*

BNF Grammars

- BNF rules (aka *productions*) have form

  X ::= *y*

  where **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: <Sum>
- Start symbol = <Sum>

- <Sum> ::= 0
- <Sum >::= 1
- <Sum> ::= <Sum> + <Sum>
- <Sum> ::= (<Sum>)
- Can be abbreviated as

<Sum> ::= 0 | 1
| <Sum> + <Sum> | ( )

BNF Deriviations

- Given rules

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

we may replace **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 Semantics

- The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Start with the start symbol:

<Sum> =>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal

<Sum> =>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute:
  - <Sum> ::= <Sum> + <Sum>

<Sum> => <Sum> + <Sum >

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute:
  - <Sum> ::= (<Sum>)

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute:
  - <Sum> ::= <Sum> + <Sum>

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute:
  - <Sum >::= 1

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

=> (<Sum> + 1) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute:
  - <Sum >::= 0

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) + 0

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) + 0

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

- Pick a rule and substitute
  - <Sum> ::= 0

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

=> (<Sum> + 1) + <Sum>

=> (<Sum> + 1) 0

=> (0 + 1) + 0

BNF Derivations

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

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

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) + 0
=> (0 + 1) + 0

Regular Grammars

- Subclass of BNF
- Only rules of form
  <nonterminal>::=<terminal><nonterminal> or
  <nonterminal>::=<terminal> or
  <nonterminal>::=ε
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)

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::=vV

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:

  <exp> ::= <factor>
  | <factor> + <factor>
  <factor> ::= <bin>
  | <bin> * <exp>
  <bin> ::= 0 | 1

- Problem: Build parse tree for 1 * 1 + 0 as an <exp>

Example cont.

- 1 * 1 + 0:   <exp>

<exp> is the start symbol for this parse tree

Example cont.

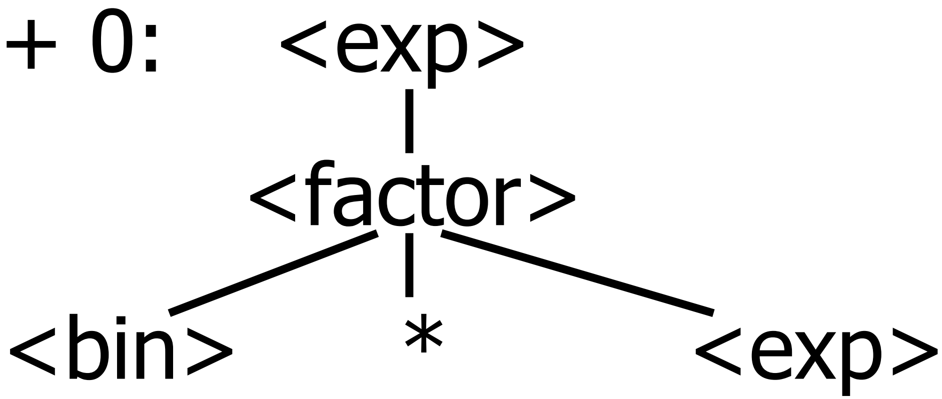
- 1 * 1 + 0: <exp>

```
        <exp>
          |
       <factor>
```

Use rule: <exp> ::= <factor>

Example cont.

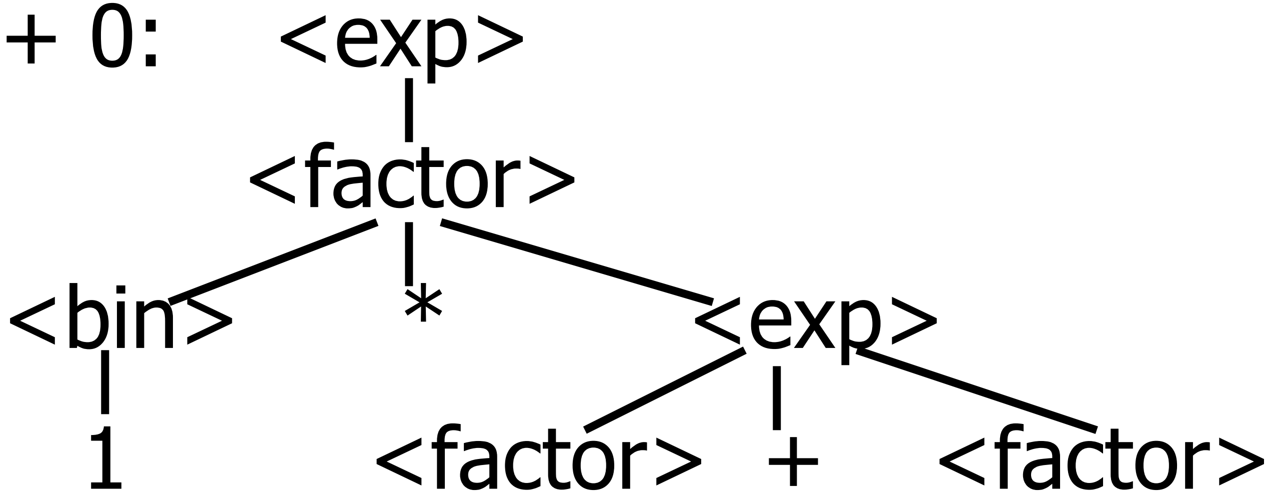
- 1 * 1 + 0:



Use rule: <factor> ::= <bin> * <exp>

Example cont.

- 1 * 1 + 0:

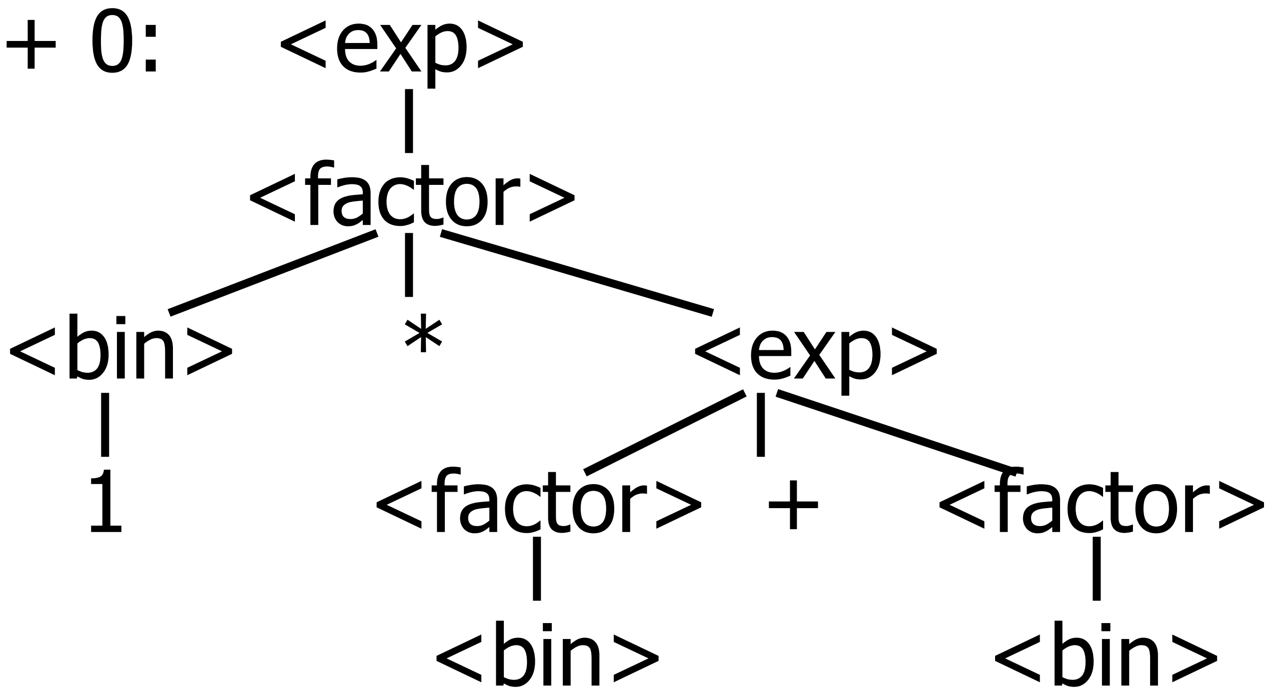


Use rules: <bin> ::= 1 and

<exp> ::= <factor> + <factor>

Example cont.

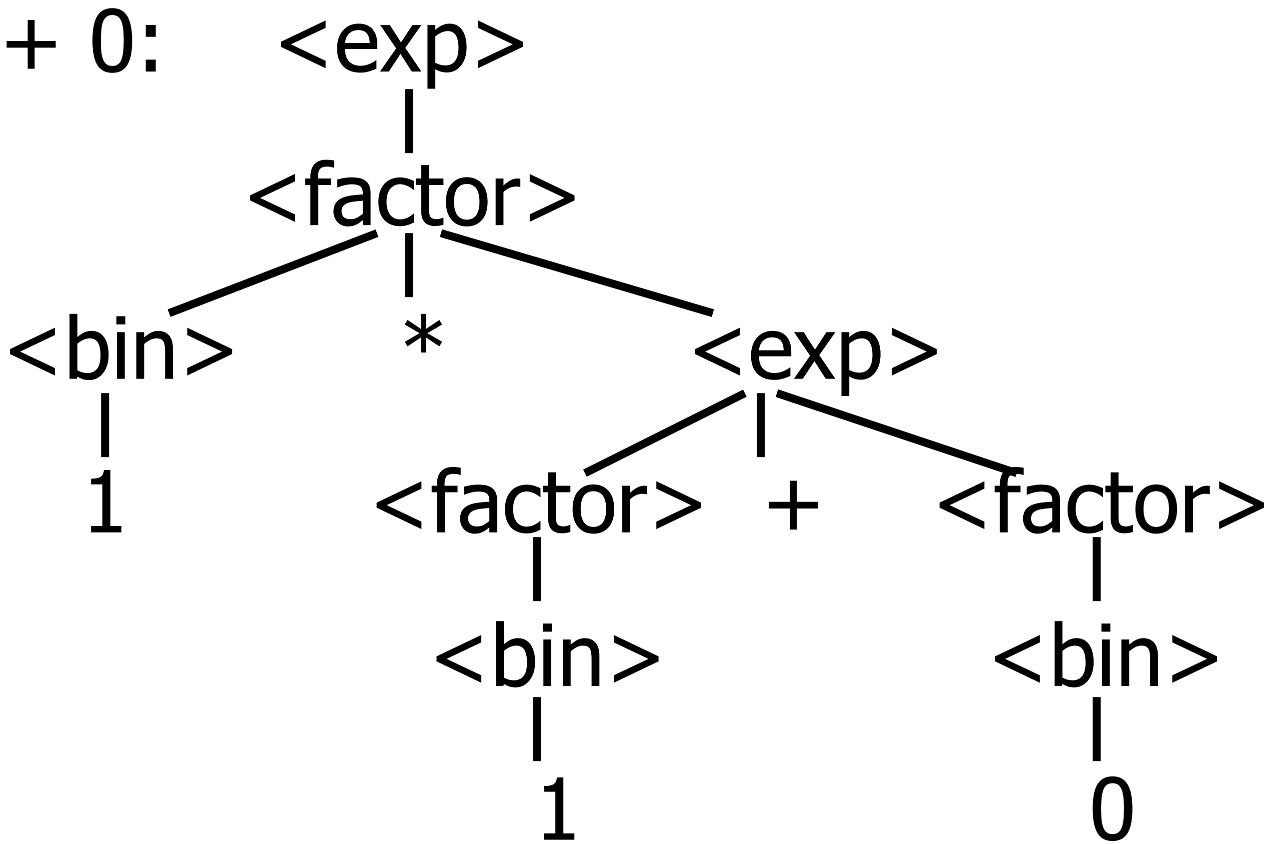
- 1 * 1 + 0:



Use rule: <factor> ::= <bin>

Example cont.

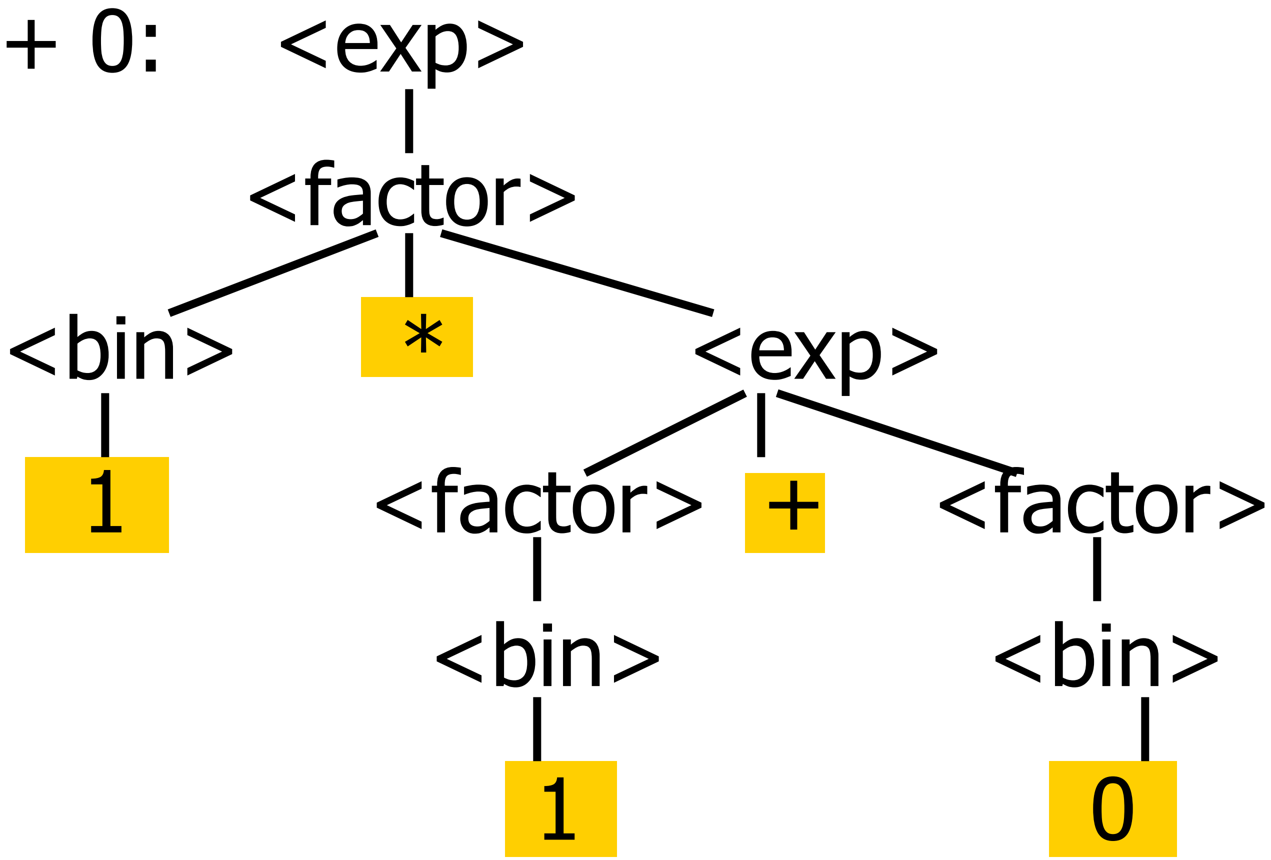
- 1 * 1 + 0:



Use rules: <bin> ::= 1 | 0

Example cont.

- 1 * 1 + 0:



Fringe of tree is string generated by grammar

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:

```
<exp>    ::= <factor> | <factor> + <factor>
<factor> ::=  <bin> |  <bin> * <exp>
<bin>    ::=  0  | 1
```

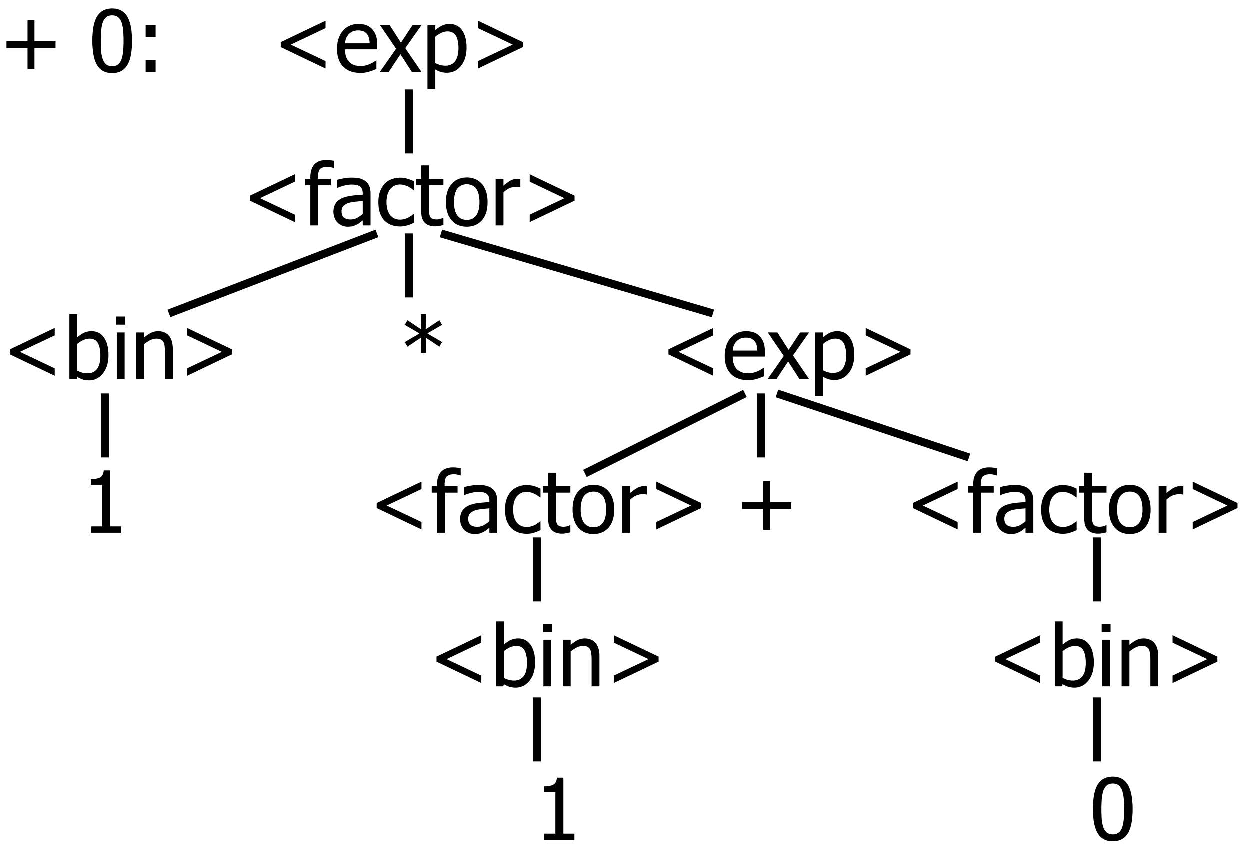
- Represent as Abstract Data Types:

- ```
  type exp      = Factor2Exp of factor
                | Plus of factor * factor
   and factor   = Bin2Factor of bin
                | Mult of bin * exp
   and bin      = Zero | One
  ```
  ```

Example cont.

```
type exp = Factor2Exp of factor
         | Plus of factor * factor
  and factor = Bin2Factor of bin
             | Mult of bin * exp
  and bin = Zero | One
```

- 1 * 1 + 0:



Example cont.

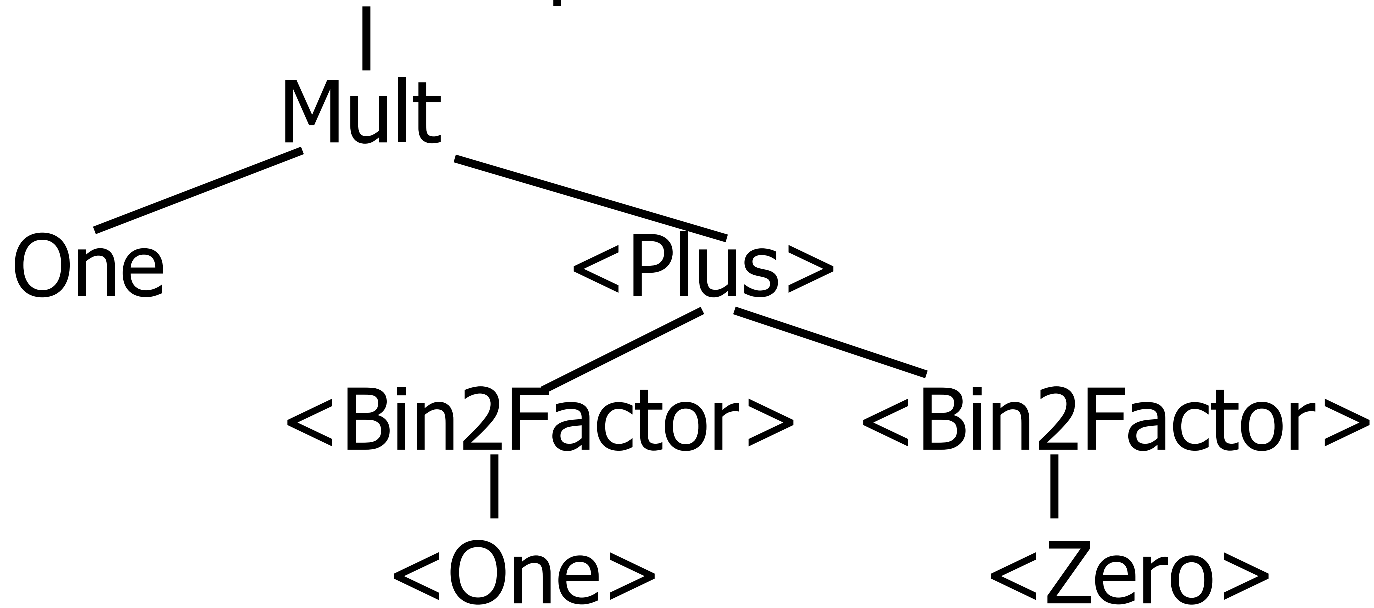
- Can be represented as

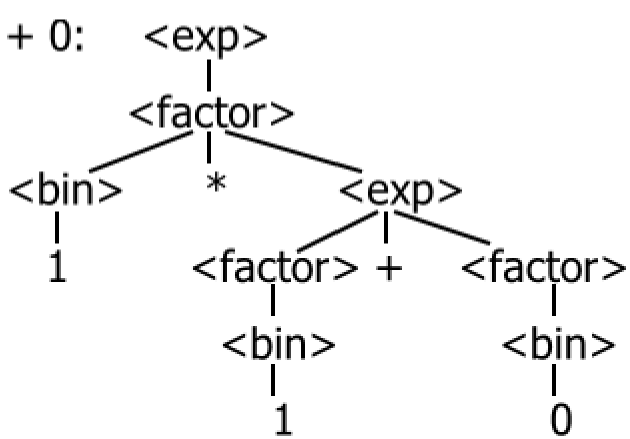
```
Factor2Exp
(Mult(One,
          Plus(Bin2Factor One,
                   Bin2Factor Zero)))
```

Example cont.

```
type exp = Factor2Exp of factor
           | Plus of factor * factor
  and factor = Bin2Factor of bin
               | Mult of bin * exp
  and bin = Zero | One
```

- 1 * 1 + 0: Factor2Exp



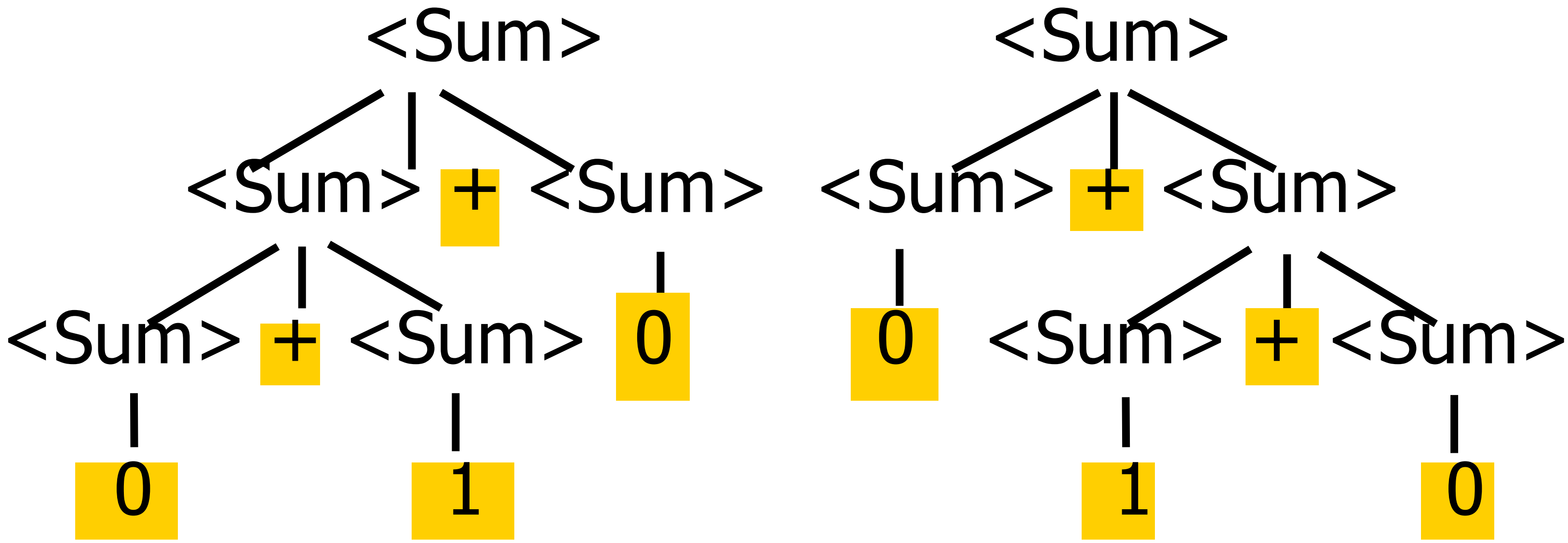


Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree
- If all BNFs 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)
- Replace old rules to use new non-terminals
- Rinse and repeat

Example

- Ambiguous grammar:

  <exp> ::= 0 | 1 | <exp> + <exp>

  | <exp> * <exp>

- String with more then one parse:

  0 + 1 + 0

  1 * 1 + 1

- Source of ambiguity: associativity and precedence

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 assoicativity, left-most one for left assoiciativity

Example

- <Sum> ::= 0 | 1 | <Sum> + <Sum>
  | (<Sum>)
- Becomes
  - <Sum> ::= <Num> | <Num> + <Sum>
  - <Num> ::= 0 | 1 | (<Sum>)

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).

  For instance multiplication (*) has higher precedence than addition (+)

- Needs to be reflected in grammar

Predence in Grammar

- Higher precedence translates to longer derivation chain

- Example:

  <exp> ::= 0 | 1 | <exp> + <exp> | <exp> * <exp>

- Becomes

  <exp> ::= <mult_exp>| <exp> + <mult_exp>
  <mult_exp> ::= <id> | <mult_exp> * <id>
  <id> ::= 0 | 1

Disambiguating a Grammar

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** to have higher precedence than ***b***, which in turn has higher precedence than ***m***, and such that ***m*** associates to the left.

Disambiguating a Grammar

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** to have higher precedence than ***b***, which in turn has higher precedence than ***m***, and such that ***m*** associates to the left.

- <exp> ::= <exp> m <not_m> | <not_m>
- <not_m> ::= b <not_m> | <not_b_m>
- <not_b_m> ::= <not_b_m>a | 0 | 1

Disambiguating a Grammar – Take 2

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***b*** to have <u>higher precedence</u> than ***m***, which in turn has <u>higher precedence</u> than ***a***, and such that ***m*** <u>associates to the right</u>.

Disambiguating a Grammar – Take 2

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***b*** has <u>higher precedence</u> than ***m***, which in turn has <u>higher precedence</u> than ***a***, and such that ***m*** <u>associates to the right</u>.
- <exp> ::=
  <no_a_m> | <no_m> m <no_a>| <exp> a
- <no_a> ::= <no_a_m> | <no_a_m> m <no_a>
- <no_m> ::= <no_a_m> | <exp> a
- <no_a_m> ::= b <no_a_m> | 0 | 1

Disambiguating a Grammar – Take 3

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** has higher precedence than ***m***, which in turn has higher precedence than ***b***, and such that ***m*** associates to the right.

- For you…

How do we disambiguate in this case?

- Our old friend:

```
<exp>     ::= <factor>
            |  <factor> +  <factor>

<factor>  ::=  <bin>
            |   <bin>  *  <exp>

<bin>     ::=  0  | 1
```

- How do we make multiplication have higher precedence than addition?

Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., 1 * (0 + 1)

```
<exp>      ::= <factor>
             |  <factor> +  <exp>

<factor>   ::=  <bin>
             |   <bin>  *  <factor>

<bin>      ::=  0  | 1
```

Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., 1 * (0 + 1)

```
<exp>      ::= <factor>
             |  <factor> +  <exp>

<factor>   ::=  <bin>
             |   <bin>  *  <factor>

<bin>      ::=  0  | 1 | ( <exp> )
```

Moving On With Richer Expressions

- How do we extend the grammar to support other operations, subtraction and division?

```
<exp>      ::= <factor>
             |  <factor> +  <exp> | <factor> - <exp>

<factor>   ::=  <bin>
             |   <bin>  *  <exp> | <bin> / <factor>

<bin>      ::=  0  | 1 | ( <exp> )
```

Disambiguating Grammars – Dangling Else

- stmt ::= …
  | **if** (expr) stmt
  | **if** (expr) stmt **else** stmt

- How can we parse
  if (e1) if (e2) s1 else s2 ?

Disambiguating Grammars – Dangling Else

- Try: let us try to differentiate if we have **if** inside the **then** branch or not….
- stmt = open_stmt | closed_stmt
- open_stmt ::= **if** (expr) stmt
                    | **if** (expr) closed_stmt **else** open_stmt
- closed_stmt ::= non_if_statement
                    | **if** (expr) closed_stmt **else** closed_stmt

- How can we parse **if** (e1) **if** (e2) s1 **else** s2 now ?

Disambiguating Grammars – Overlapping

- seq = ε | may_word | word seq
- may_word = ε | "word"

- How do you parse "word"? And ε?

- How do you fix it?

How do you know you have ambiguity?

- The Ocaml parser generator (ocamlyacc) will report ambiguity in the grammar as “conflicts”:
- ***Shift/reduce:*** Usually caused by lack of associativity or precedence information in grammar
- ***Reduce/reduce:*** can’t decide between two different rules to reduce by; Not always clear what the problem is, but often right-hand side of one production is the suffix of another
- We will explain what these conflicts mean next time!

Parser Code

- Ocamlyacc is a parser generator for Ocaml
  - Similar generators exist for other languages
  - Search under: Yacc, Bison, Menhir…
  - Another family: Antlr
- Input: high level specification (<grammar>.mly file)
- Output: tokens (<*grammar*>.mli) and generated parser (<*grammar*>.ml)
  - <*grammar*>.ml defines a 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

- <*grammar*>.mly 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
- <*trailer*> similar. Possibly used to call parser

Ocamlyacc Input

- <*grammar*>.mly File format:

```
%{
    <header>
%}
    <declarations>
%%
    <rules>
%%
    <trailer>
```

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 Input

- <*grammar*>.mly File format:

%{

<*header*>

%}

<*declarations*>

%%

<*rules*>

%%

<*trailer*>

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 - Grammar

A slight variation of what we've seen earlier:

Expr ::= Term | Term + Expr | Term – Expr

Term ::= Factor | Factor * Term | Factor / Term

Factor ::= Id | (Expr)

Example - Base types

```
Expr   ::= Term | Term + Expr | Term – Expr
Term   ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
```

```
(* 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

Expr ::= Term | Term + Expr | Term – Expr
Term ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | (Expr)

```
{ 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 + Expr | Term – Expr
Term   ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
```

```
expr:
    term
      { Term_as_Expr $1 }
  | term Plus_token expr
      { Plus_Expr ($1, $3) }
  | term Minus_token expr
            { Minus_Expr ($1, $3) }
```

Example - Base types

```
(* File: expr.ml *)
type expr =
   Term_as_Expr of term
 | Plus_Expr of (term * expr)
 | Minus_Expr of (term * expr)
```

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) }
```

Expr	::= Term | Term + Expr | Term – Expr
Term	::= Factor | Factor * Term | Factor / Term
Factor	::= Id | (Expr)

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)
```

Example - Parser (exprparse.mly)

```
Expr   ::= Term | Term + Expr | Term – Expr
Term   ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
```

```
factor:
    Id_token
       { Id_as_Factor $1 }
  | Left_parenthesis expr Right_parenthesis
       {Parenthesized_Expr_as_Factor $2 }

main:
  | expr EOL
        { $1 }
```

Recall, we previously defined:

```
  %start main
  %type <expr> main
```

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
```

- Call:
  - `$ ocamlyacc options exprparse.mly`
- Get:
  - Tokens: exprparse.mli (can be used in lexer)
  - Parser: exprparse.ml
    (included in the rest of code)

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"))
  )
```

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
```

LR Parsing

General plan:

- 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: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>

(0 + 1) + 0

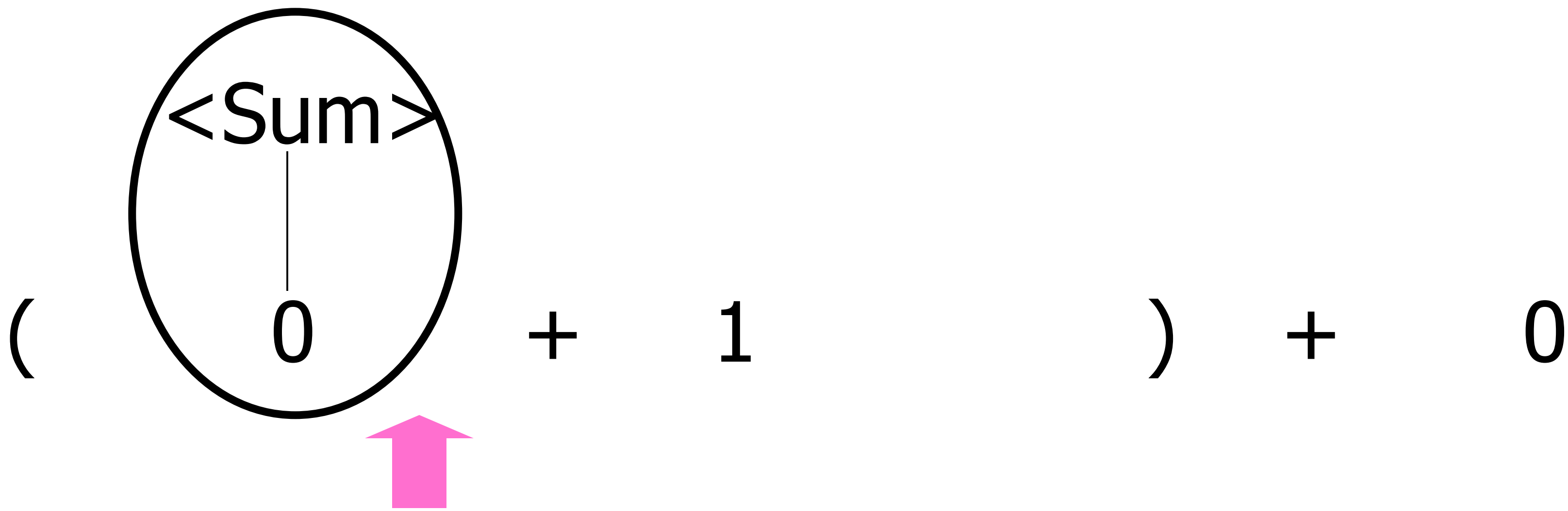
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>

(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>



Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>

<Sum>

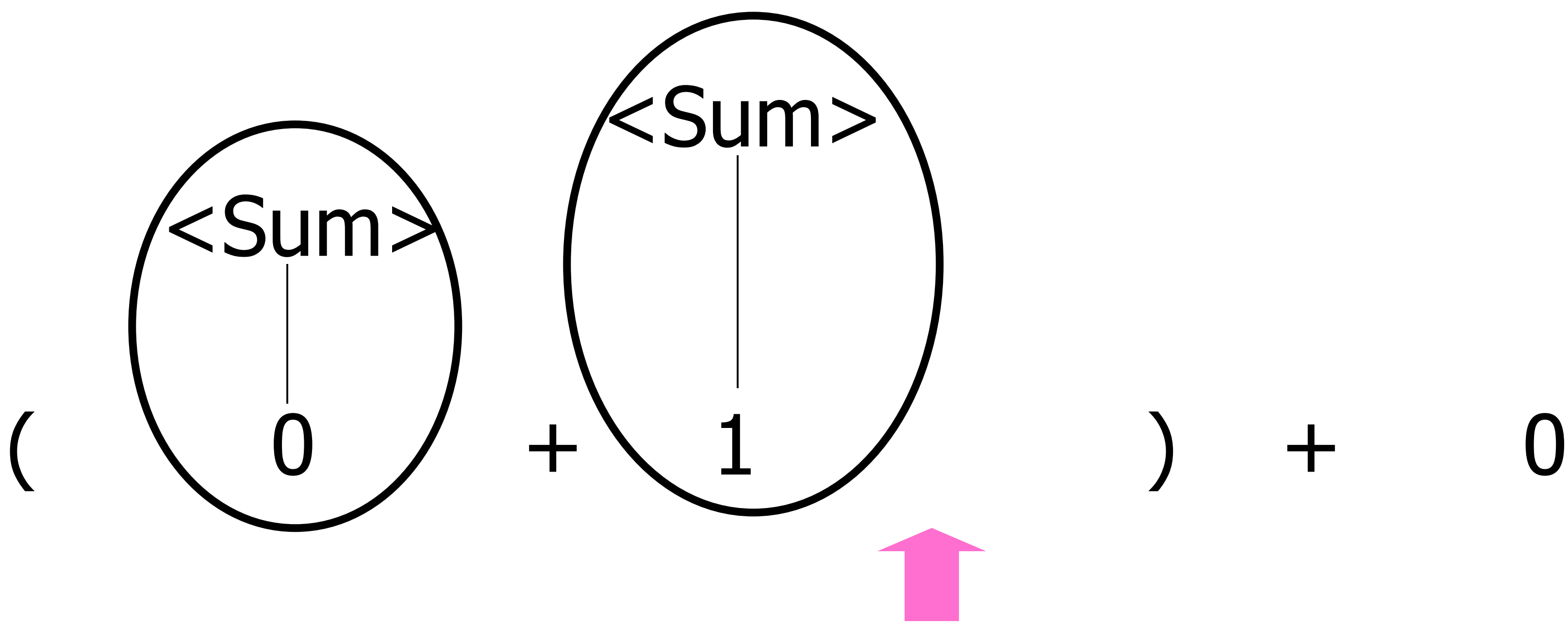
(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

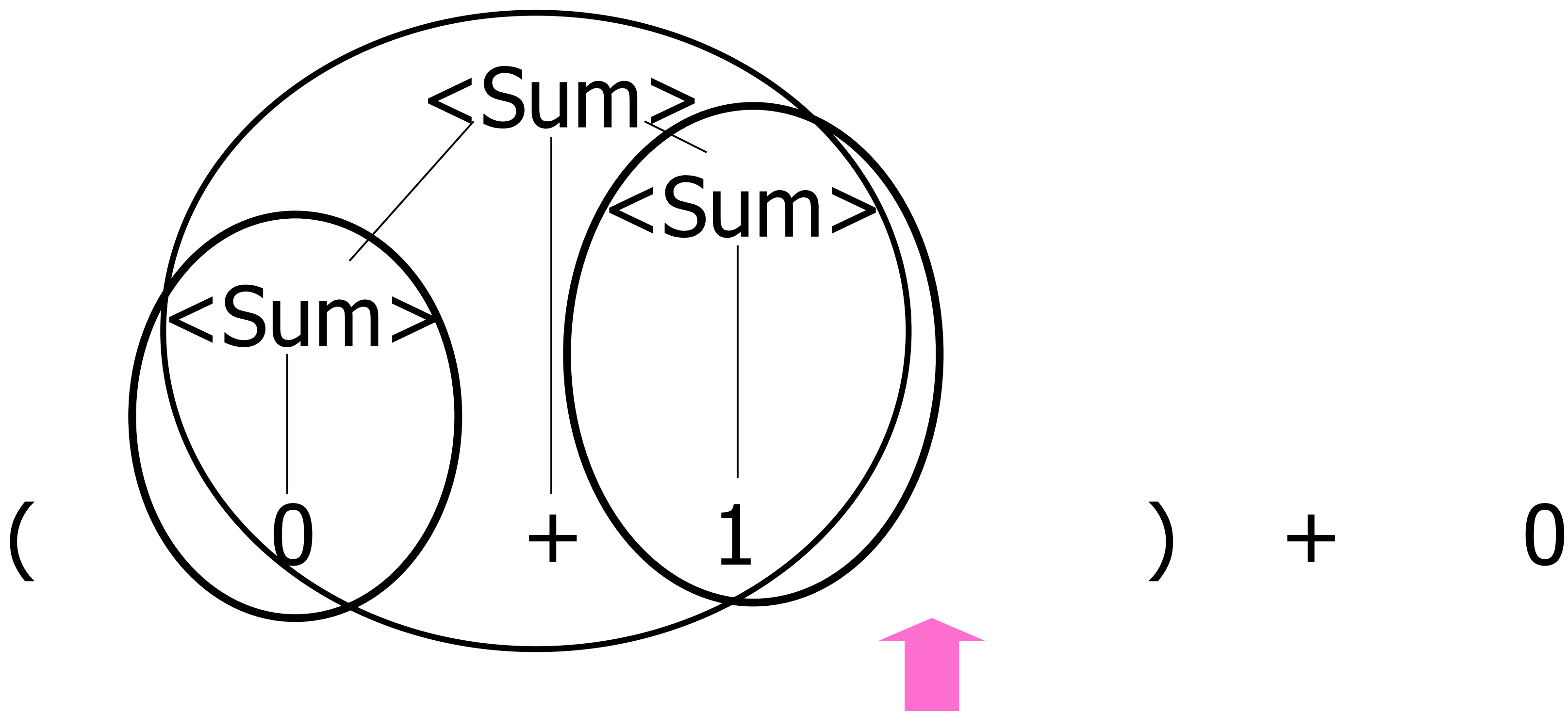
<Sum>

(0 + 1) + 0

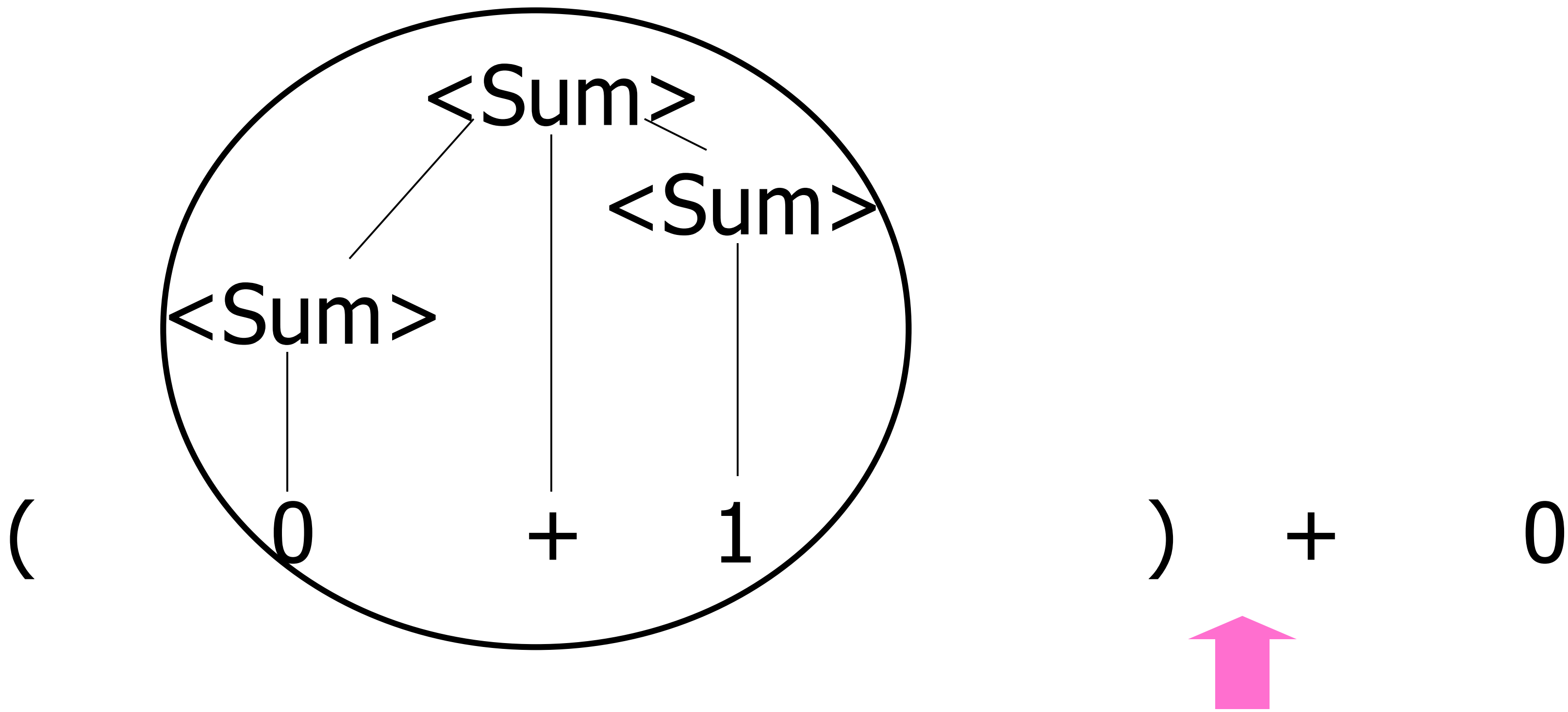
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



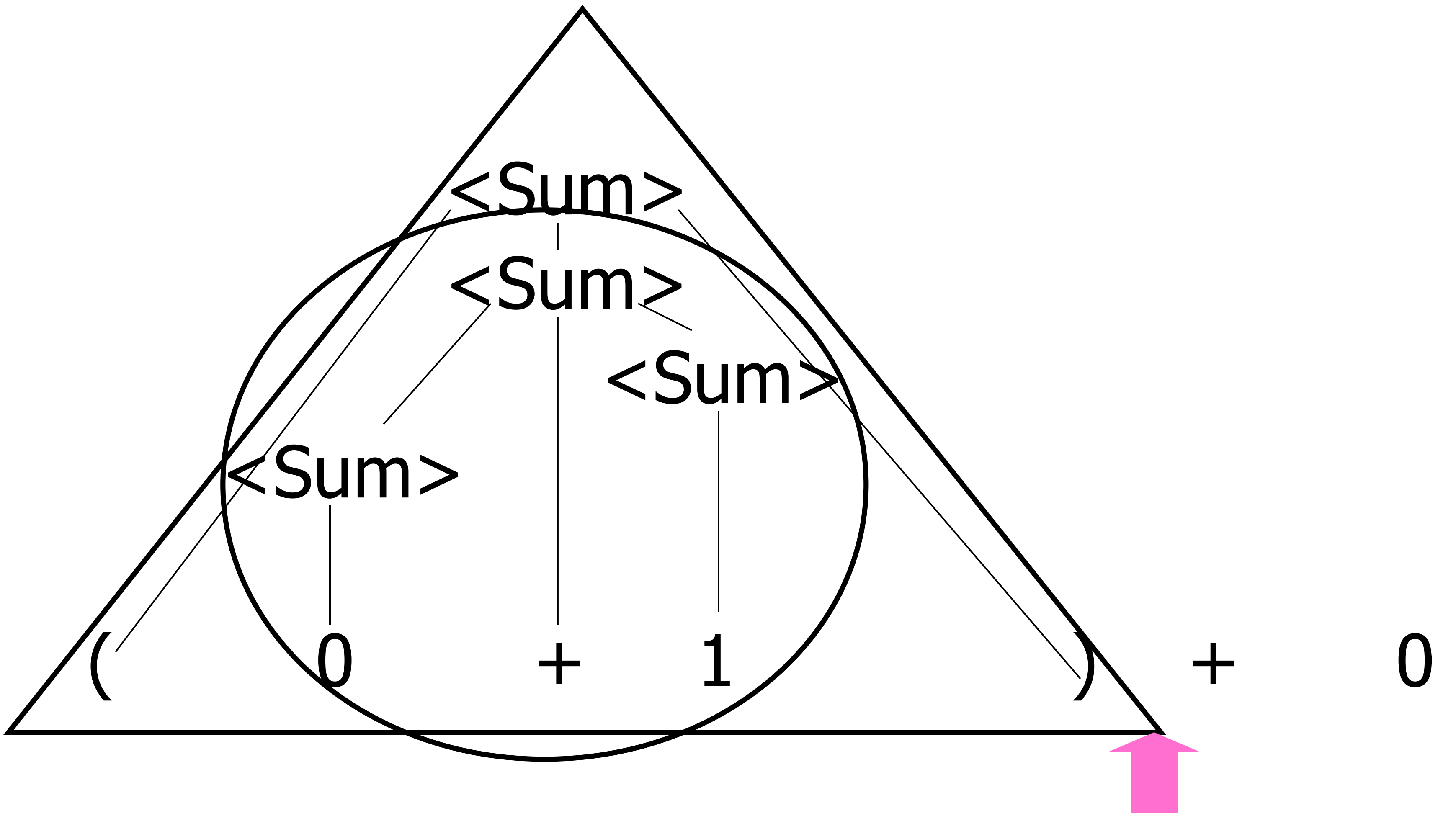
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



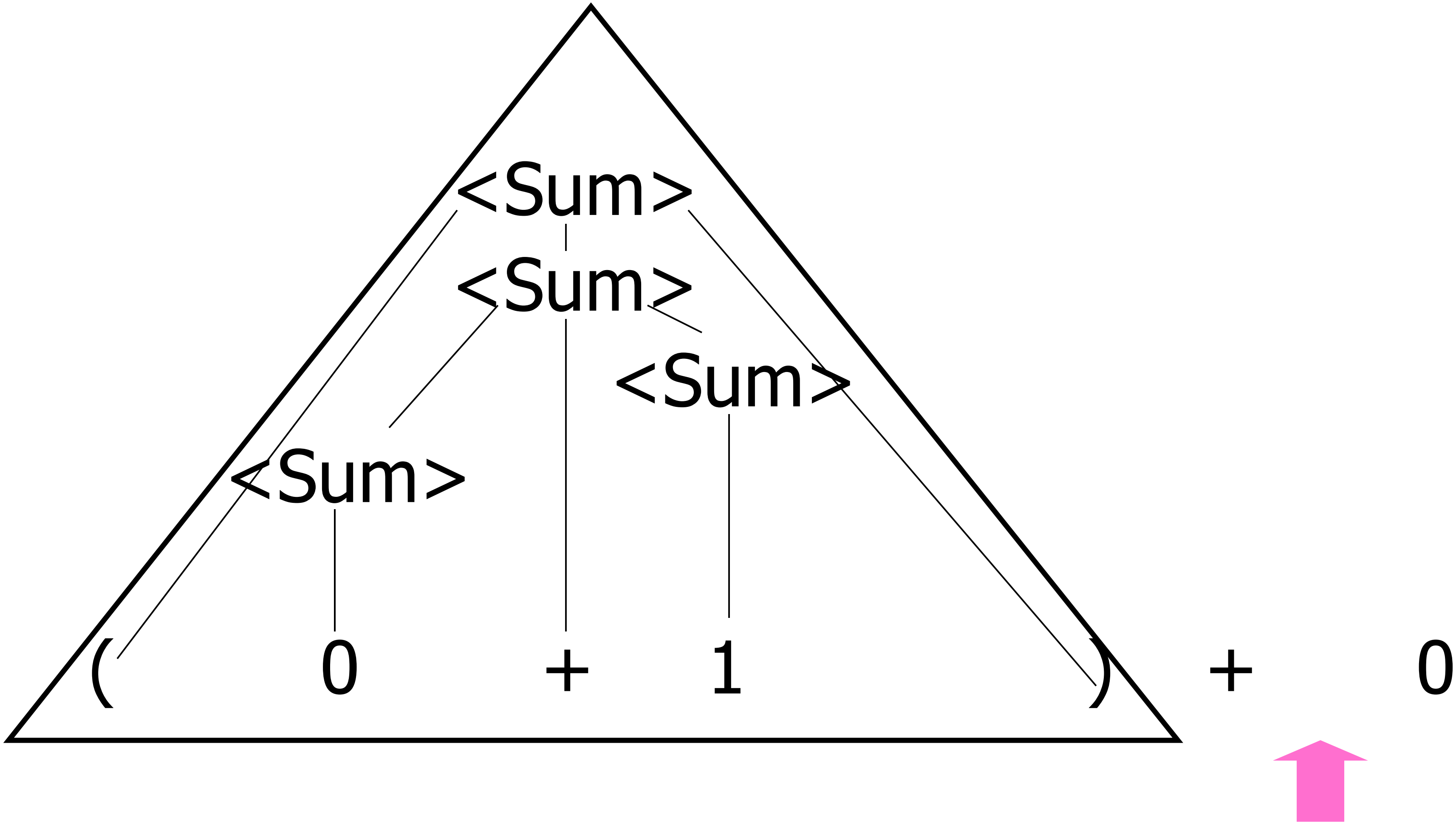
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>



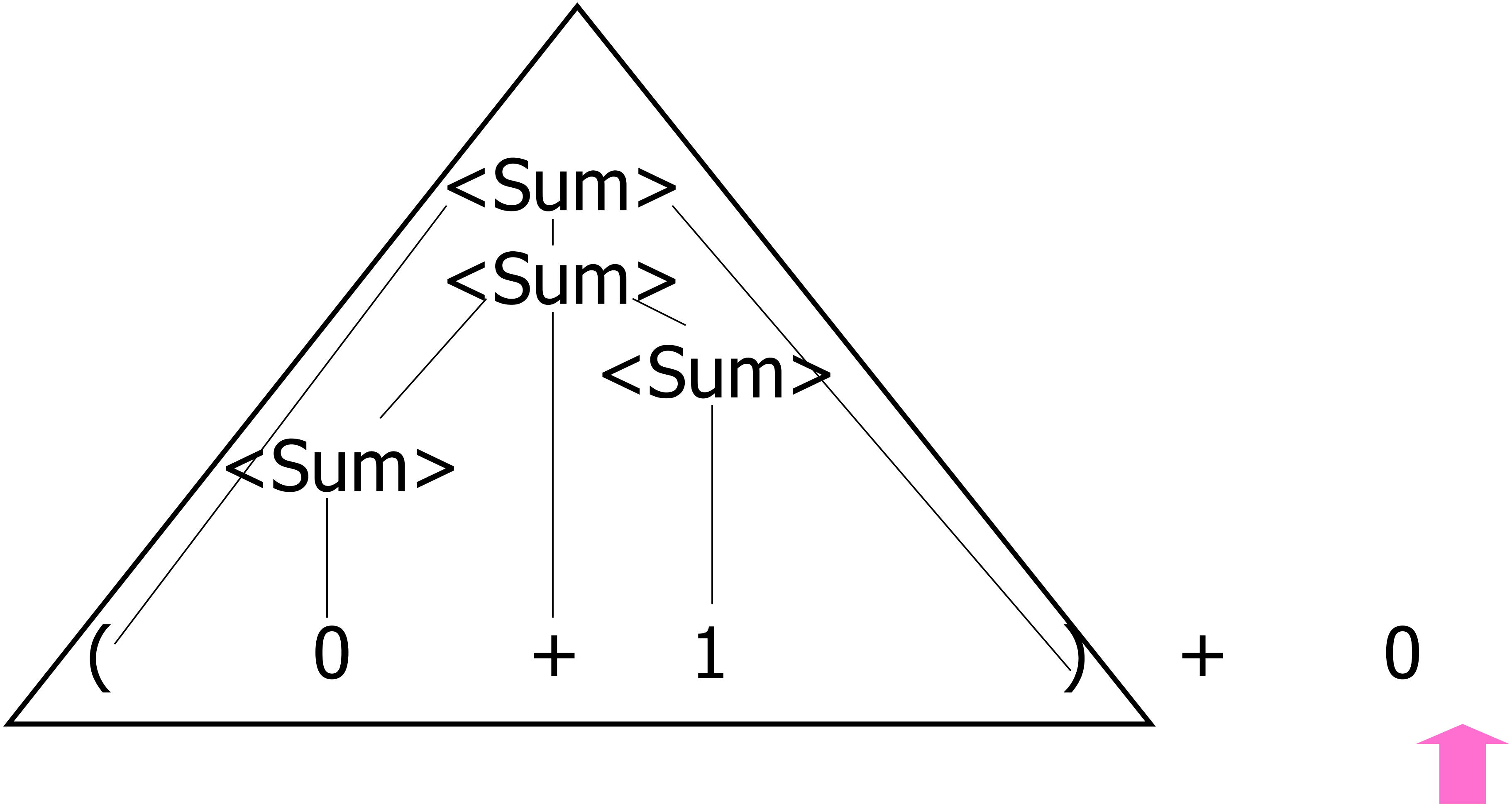
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



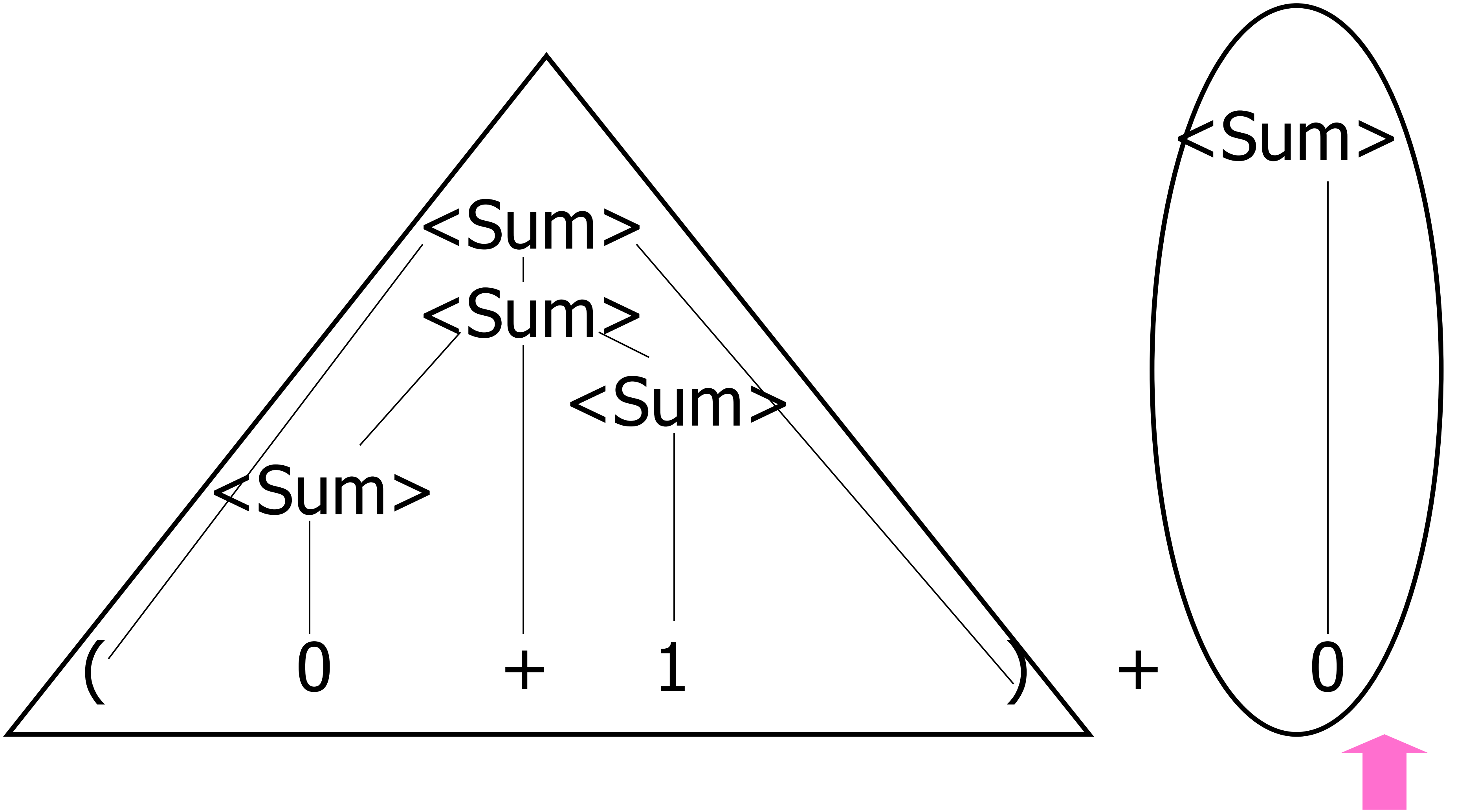
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



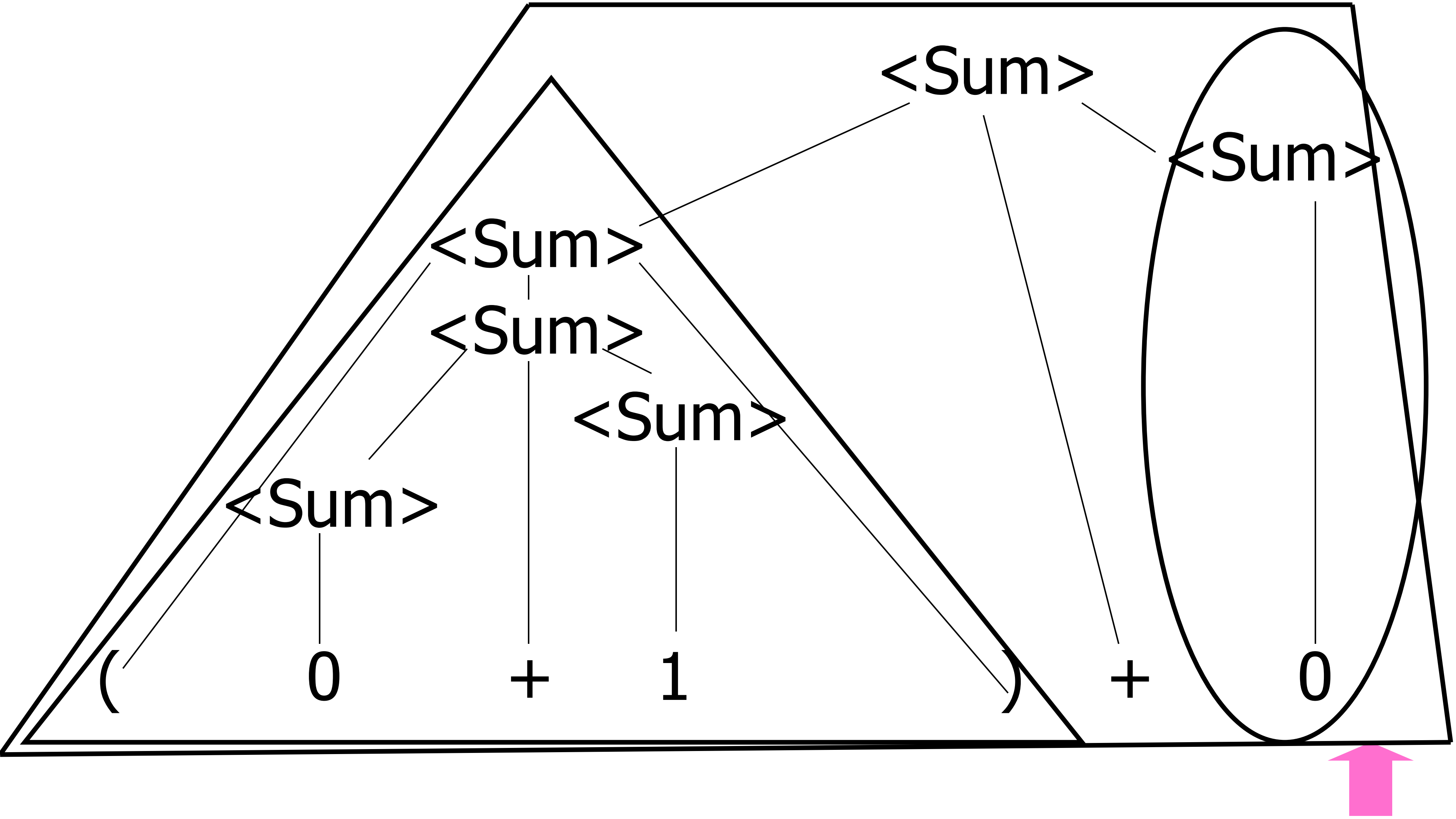
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



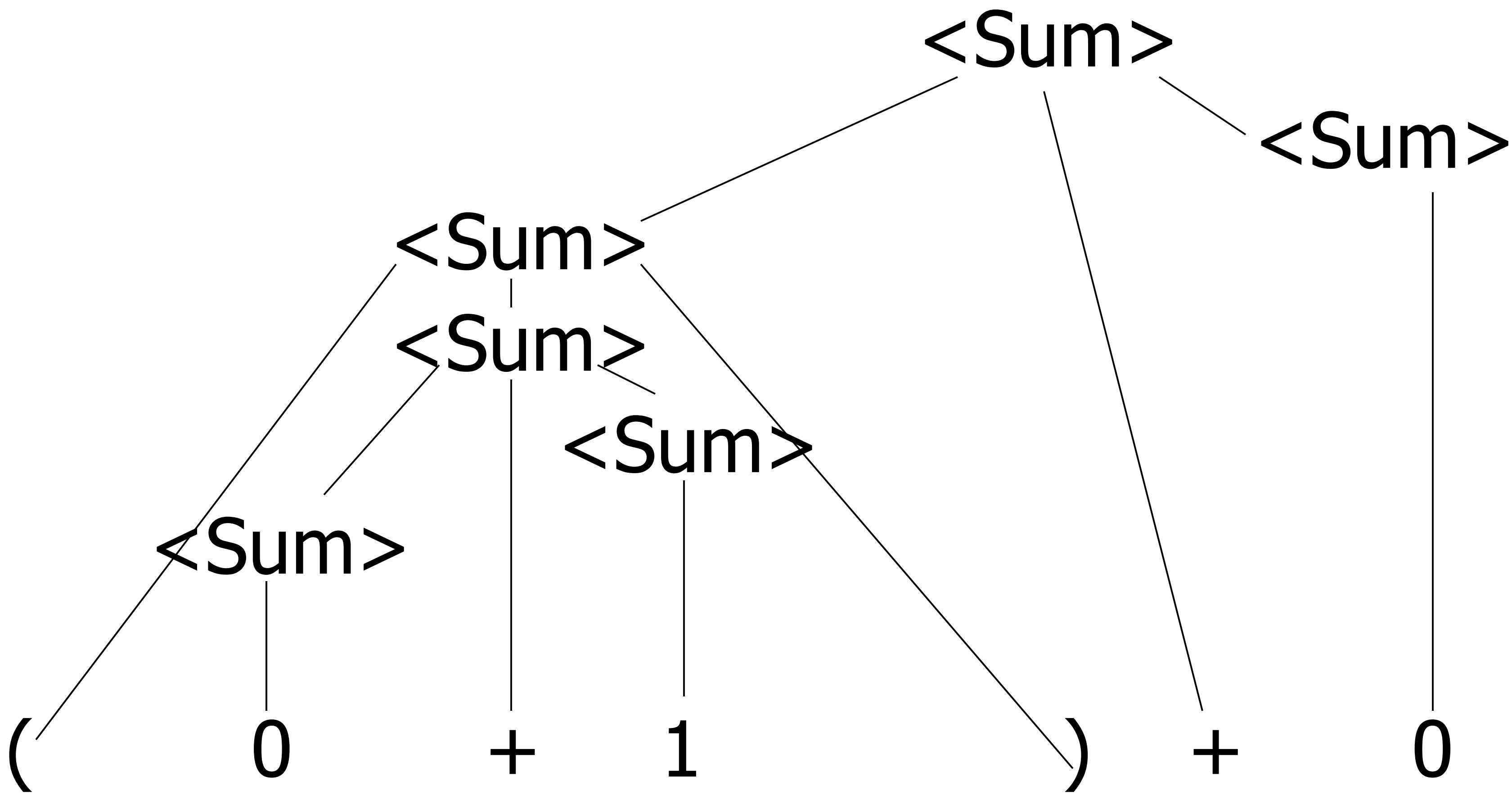
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>



Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



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 * 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 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

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= ● (0 + 1) + 0 shift

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*)

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= ● (0 + 1) + 0 shift

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

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * 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 Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum> + <Sum> ●) + 0 reduce
=> (<Sum> + 1 ●) + 0 reduce
= (<Sum> + ● 1) + 0 shift
= (<Sum> ● + 1) + 0 shift
=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is
   E ::= u
   a) Remove 2 * 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 Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> •) + 0	shift
=>	(<Sum> + <Sum> •) + 0	reduce
=>	(<Sum> + 1 •) + 0	reduce
=	(<Sum> + • 1) + 0	shift
=	(<Sum> • + 1) + 0	shift
=>	(0 • + 1) + 0	reduce
=	(• 0 + 1) + 0	shift
=	• (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum>) ● + 0	reduce
= (<Sum> ●) + 0	shift
=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	<Sum> ● + 0	shift
=>	(<Sum>) ● + 0	reduce
=	(<Sum> ●) + 0	shift
=>	(<Sum> + <Sum> ●) + 0	reduce
=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	<Sum> + ● 0	shift
=	<Sum> ● + 0	shift
=>	(<Sum>) ● + 0	reduce
=	(<Sum> ●) + 0	shift
=>	(<Sum> + <Sum> ●) + 0	reduce
=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum>	=>		
	=>	<Sum> + 0 ●	reduce
	=	<Sum> + ● 0	shift
	=	<Sum> ● + 0	shift
	=>	(<Sum>) ● + 0	reduce
	=	(<Sum> ●) + 0	shift
	=>	(<Sum> + <Sum> ●) + 0	reduce
	=>	(<Sum> + 1 ●) + 0	reduce
	=	(<Sum> + ● 1) + 0	shift
	=	(<Sum> ● + 1) + 0	shift
	=>	(0 ● + 1) + 0	reduce
	=	(● 0 + 1) + 0	shift
	=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum>	=>	<Sum> + <Sum > ●	reduce
	=>	<Sum> + 0 ●	reduce
	=	<Sum> + ● 0	shift
	=	<Sum> ● + 0	shift
	=>	(<Sum>) ● + 0	reduce
	=	(<Sum> ●) + 0	shift
	=>	(<Sum> + <Sum> ●) + 0	reduce
	=>	(<Sum> + 1 ●) + 0	reduce
	=	(<Sum> + ● 1) + 0	shift
	=	(<Sum> ● + 1) + 0	shift
	=>	(0 ● + 1) + 0	reduce
	=	(● 0 + 1) + 0	shift
	=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> ●	=> <Sum> + <Sum > ●	reduce
	=> <Sum> + 0 ●	reduce
	= <Sum> + ● 0	shift
	= <Sum> ● + 0	shift
	=> (<Sum>) ● + 0	reduce
	= (<Sum> ●) + 0	shift
	=> (<Sum> + <Sum> ●) + 0	reduce
	=> (<Sum> + 1 ●) + 0	reduce
	= (<Sum> + ● 1) + 0	shift
	= (<Sum> ● + 1) + 0	shift
	=> (0 ● + 1) + 0	reduce
	= (● 0 + 1) + 0	shift
	= ● (0 + 1) + 0	shift

LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success
8. If action = **error**,
   - Stop parsing, return failure

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 * 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 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: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

● 0 + 1 + 0	shift
-> 0 ● + 1 + 0	reduce
-> <Sum> ● + 1 + 0	shift
-> <Sum> + ● 1 + 0	shift
-> <Sum> + 1 ● + 0	reduce
-> <Sum> + <Sum> ● + 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 | 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?