Programming Languages and Compilers (CS 421)

Reza Zamani

http://www.cs.uiuc.edu/class/cs421/

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

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

<Sum> =>

= | (0 + 1) + 0 shift

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

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

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

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0

Example

<Sum>

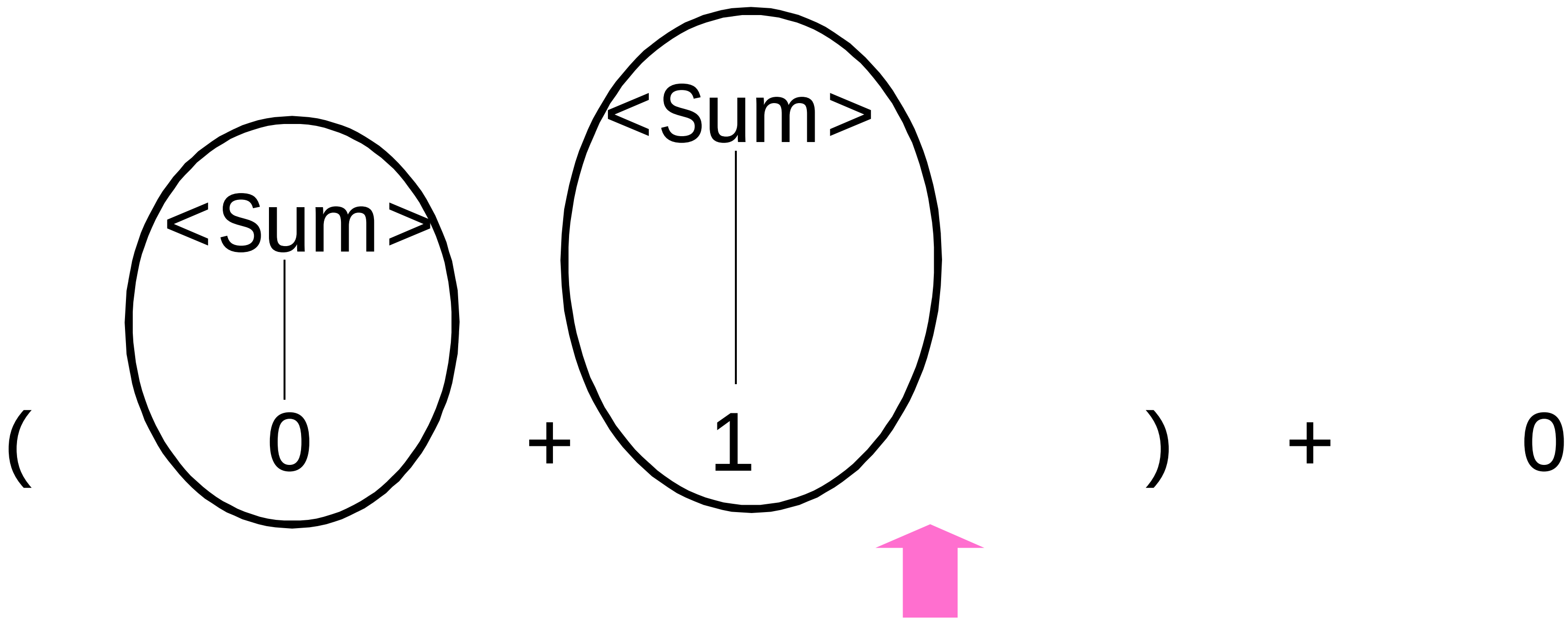
(0 + 1) + 0

Example

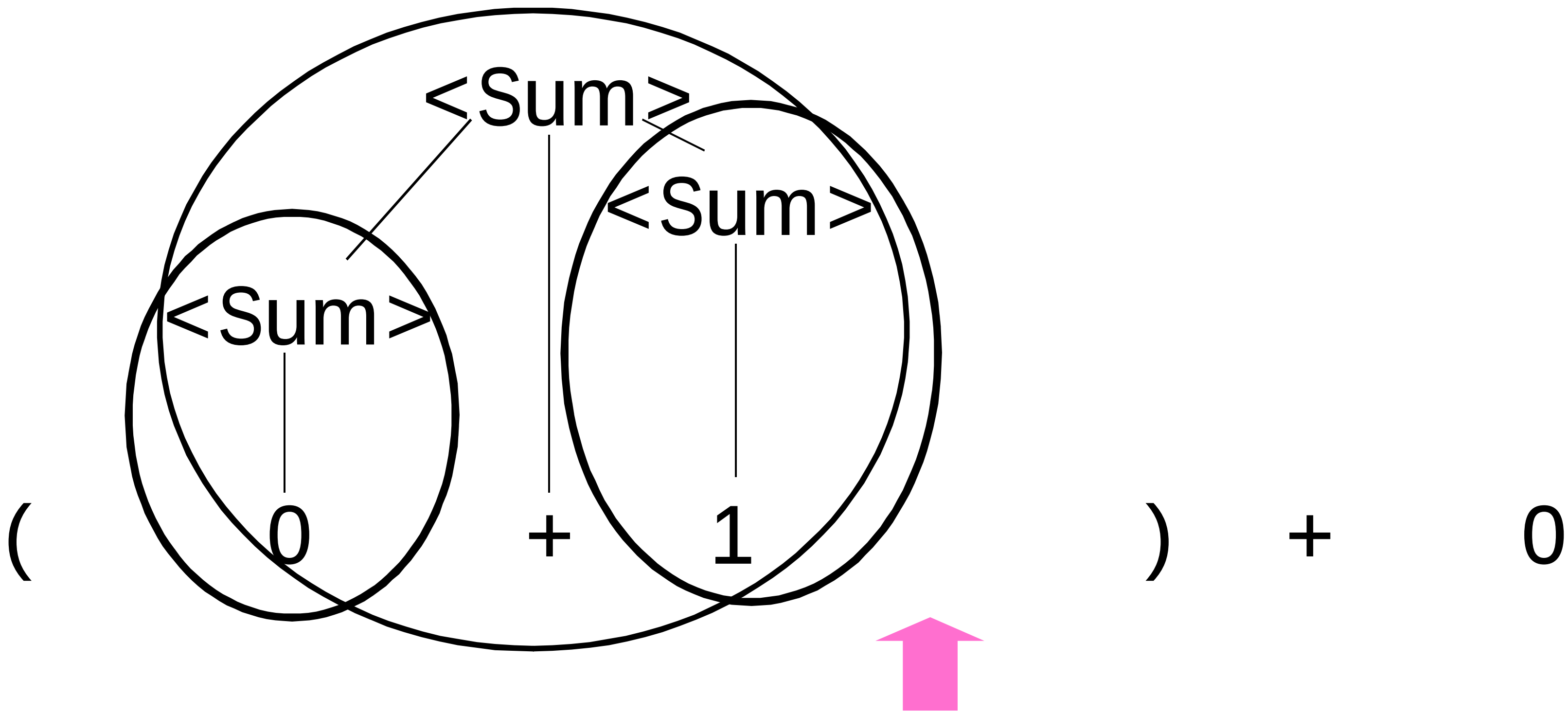
<Sum>

(0 + 1) + 0

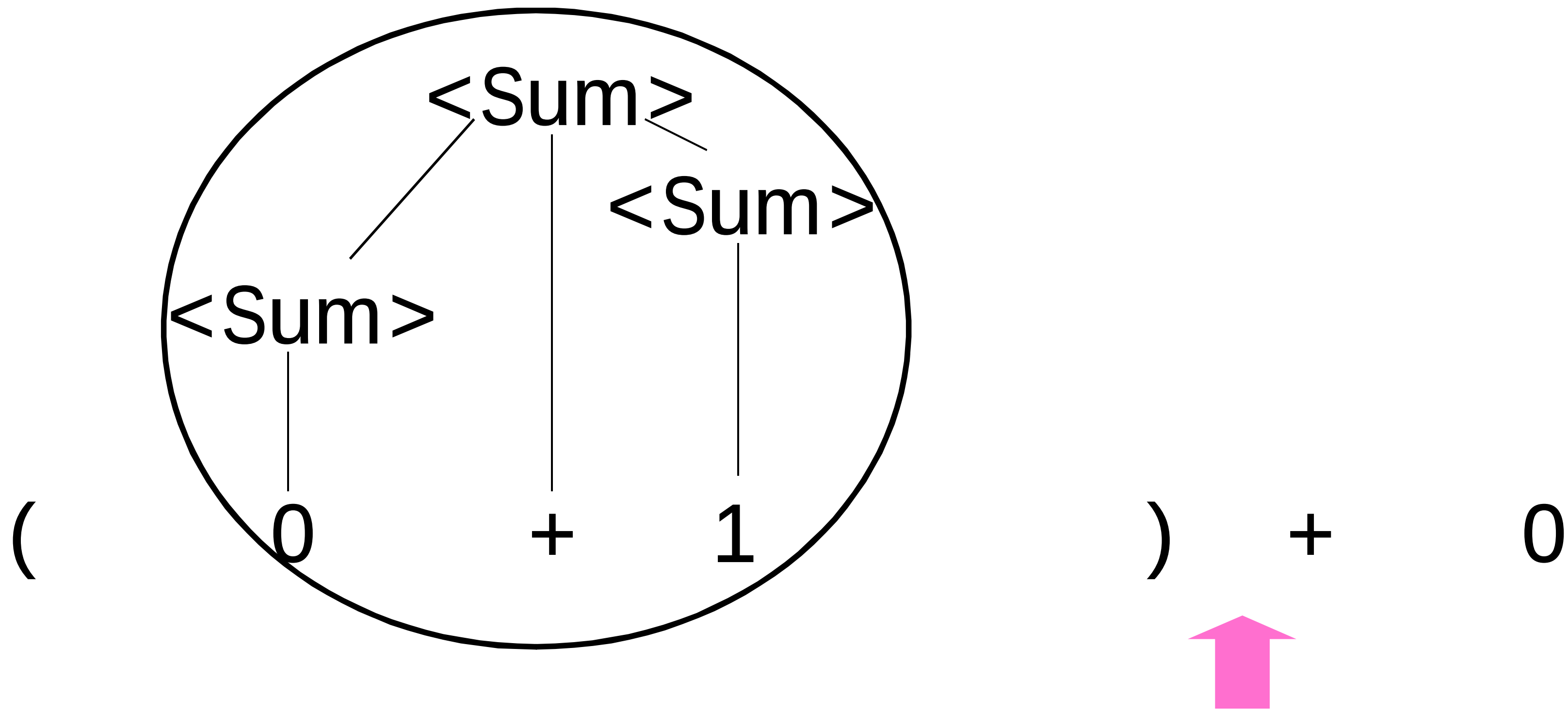
Example



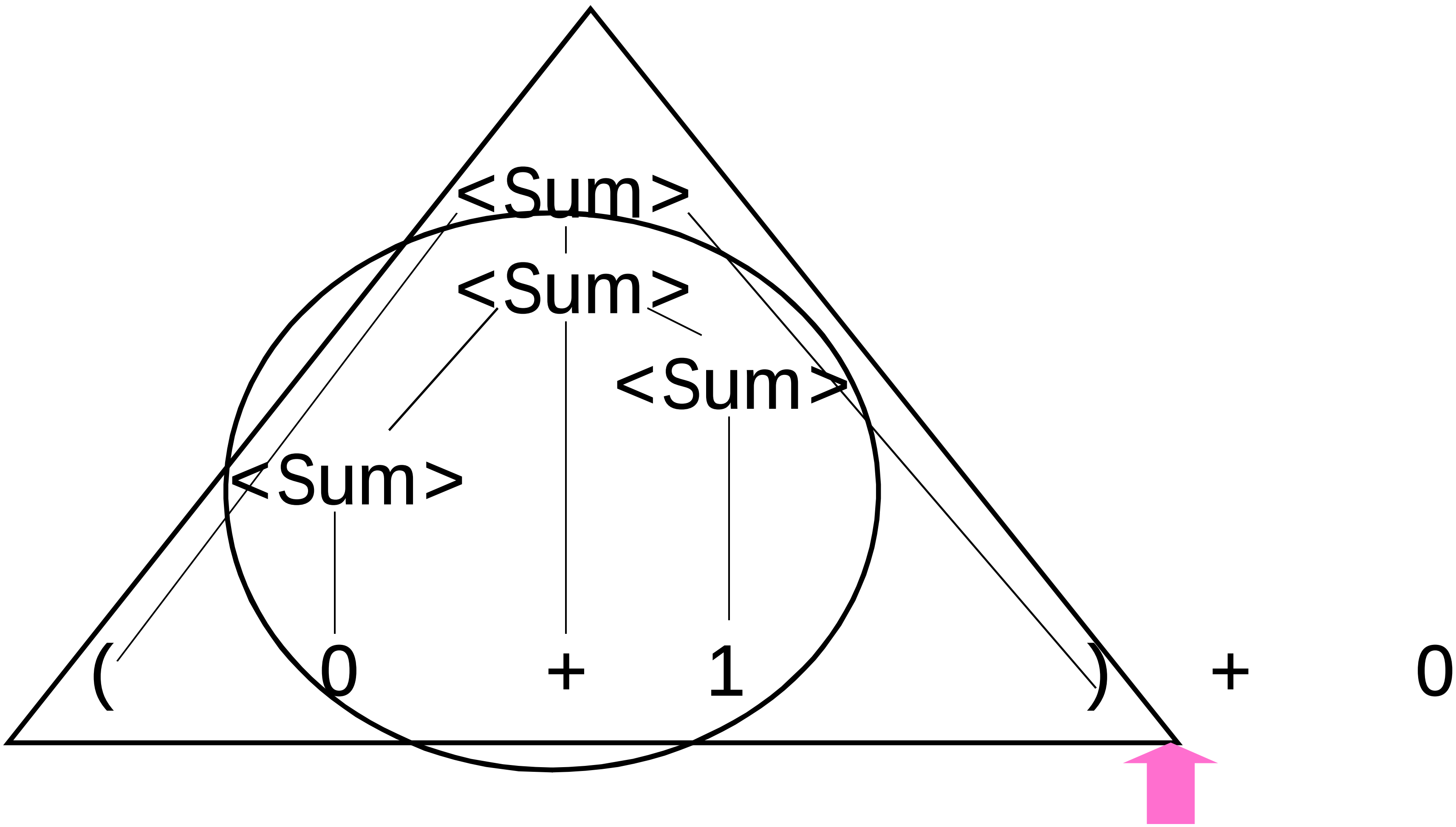
Example



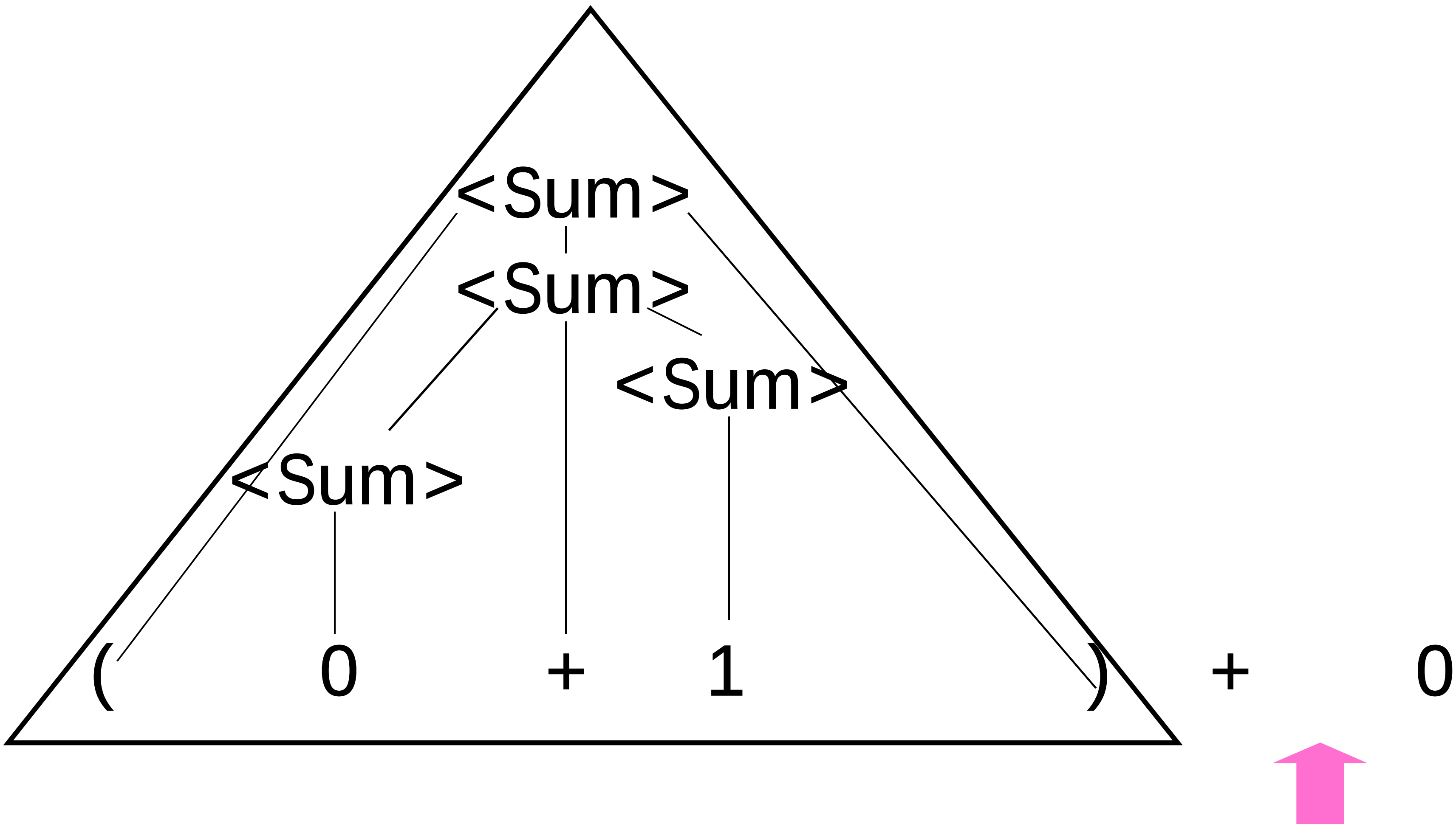
Example



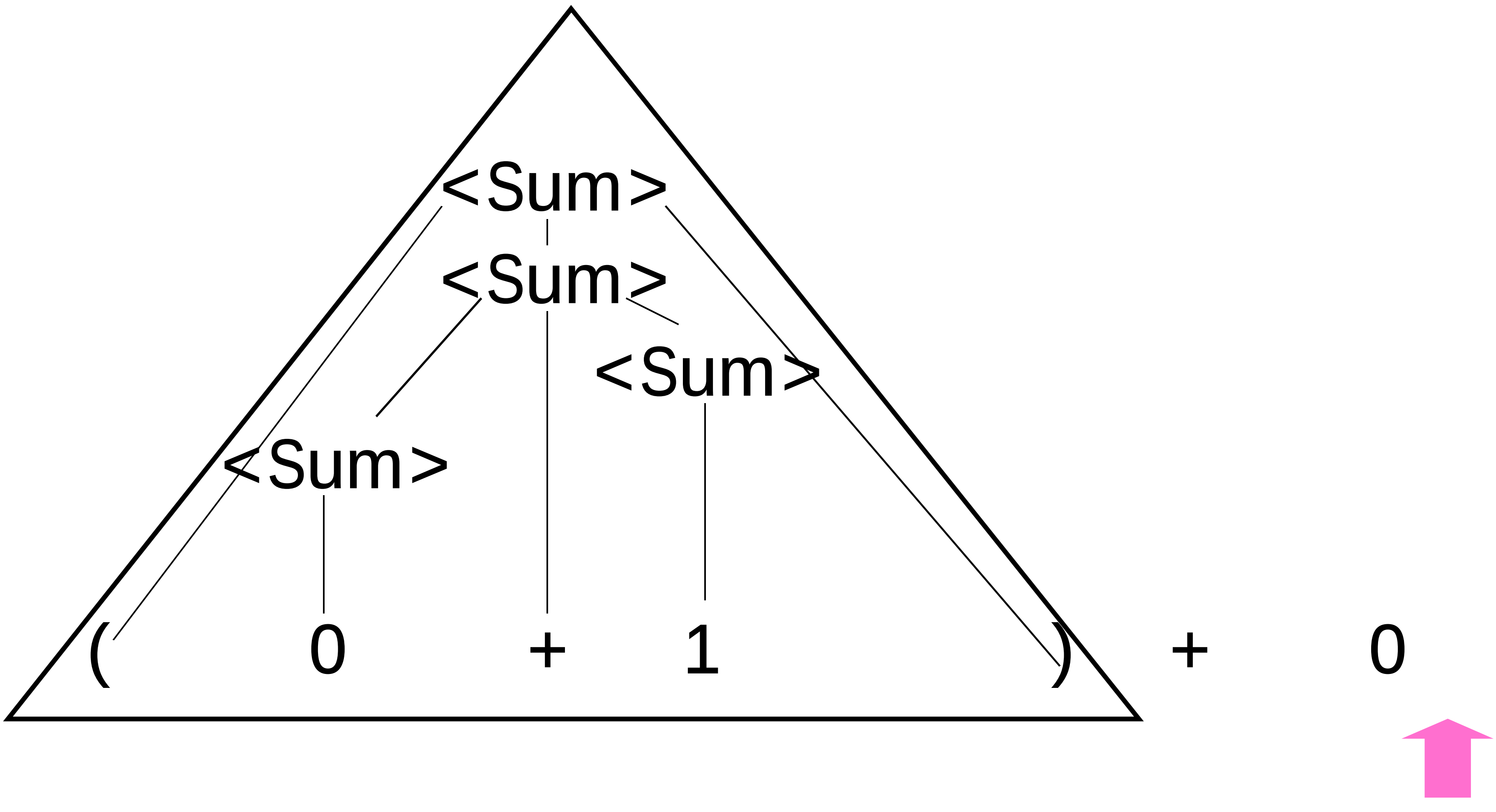
Example



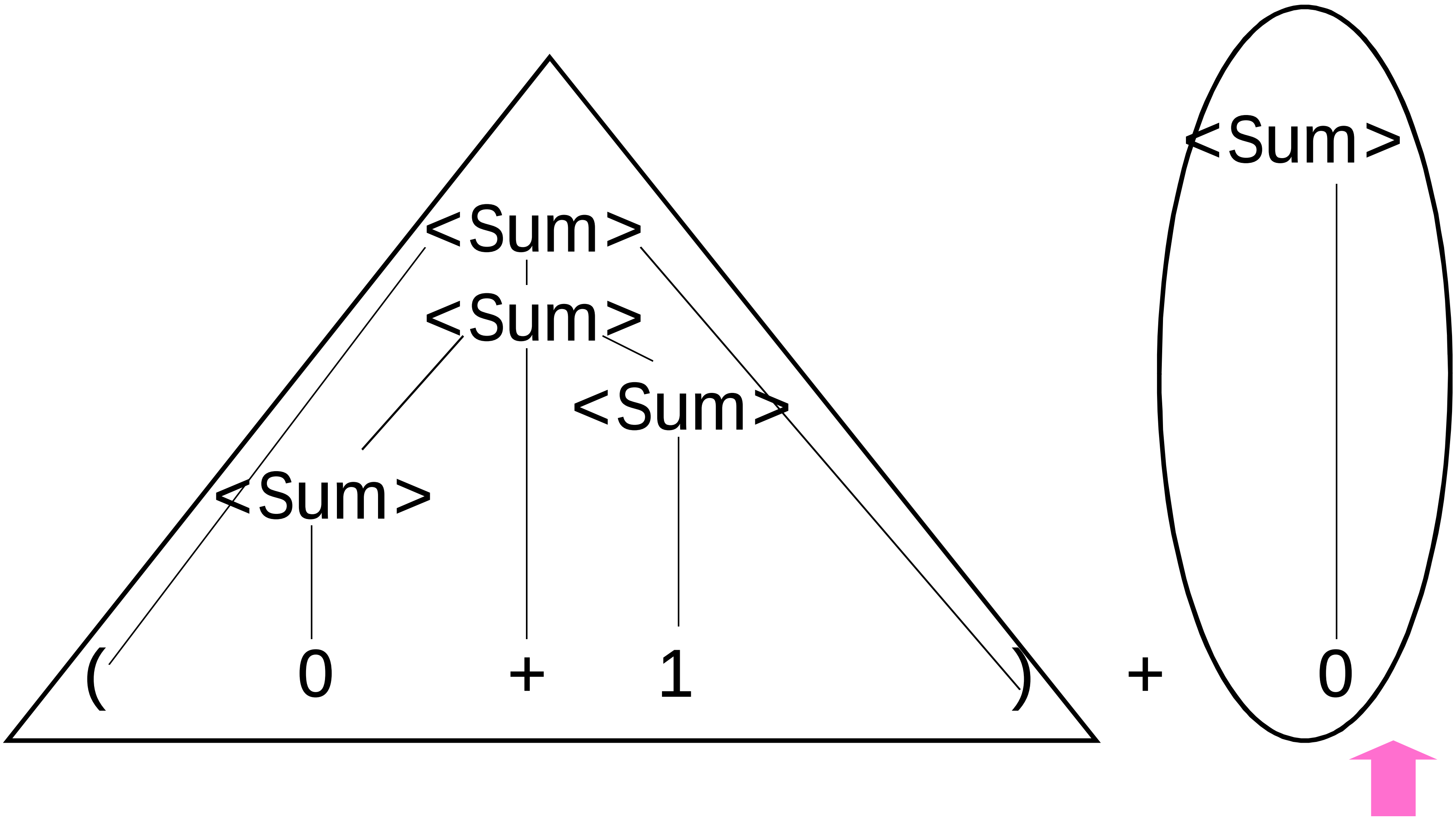
Example



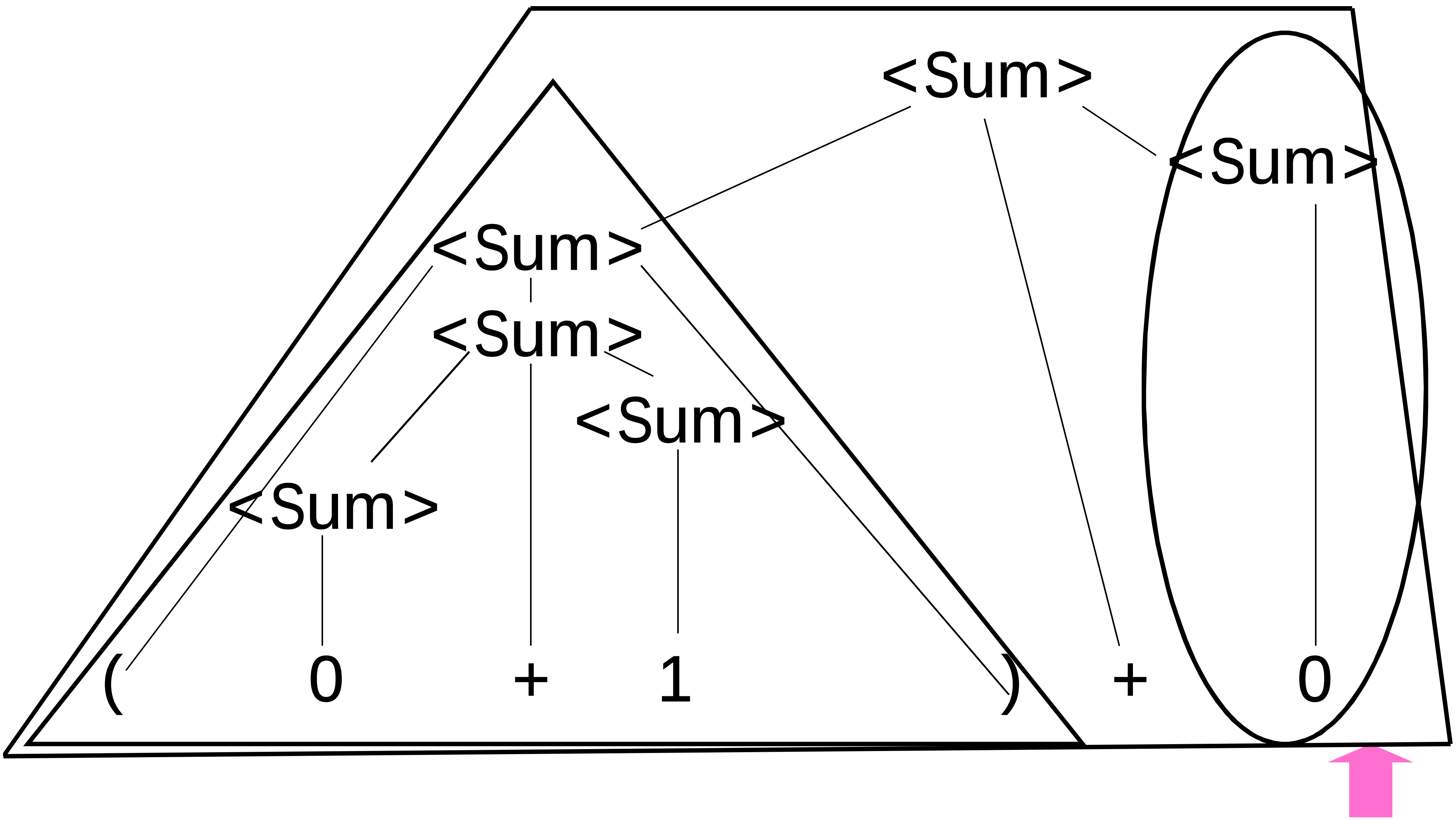
Example



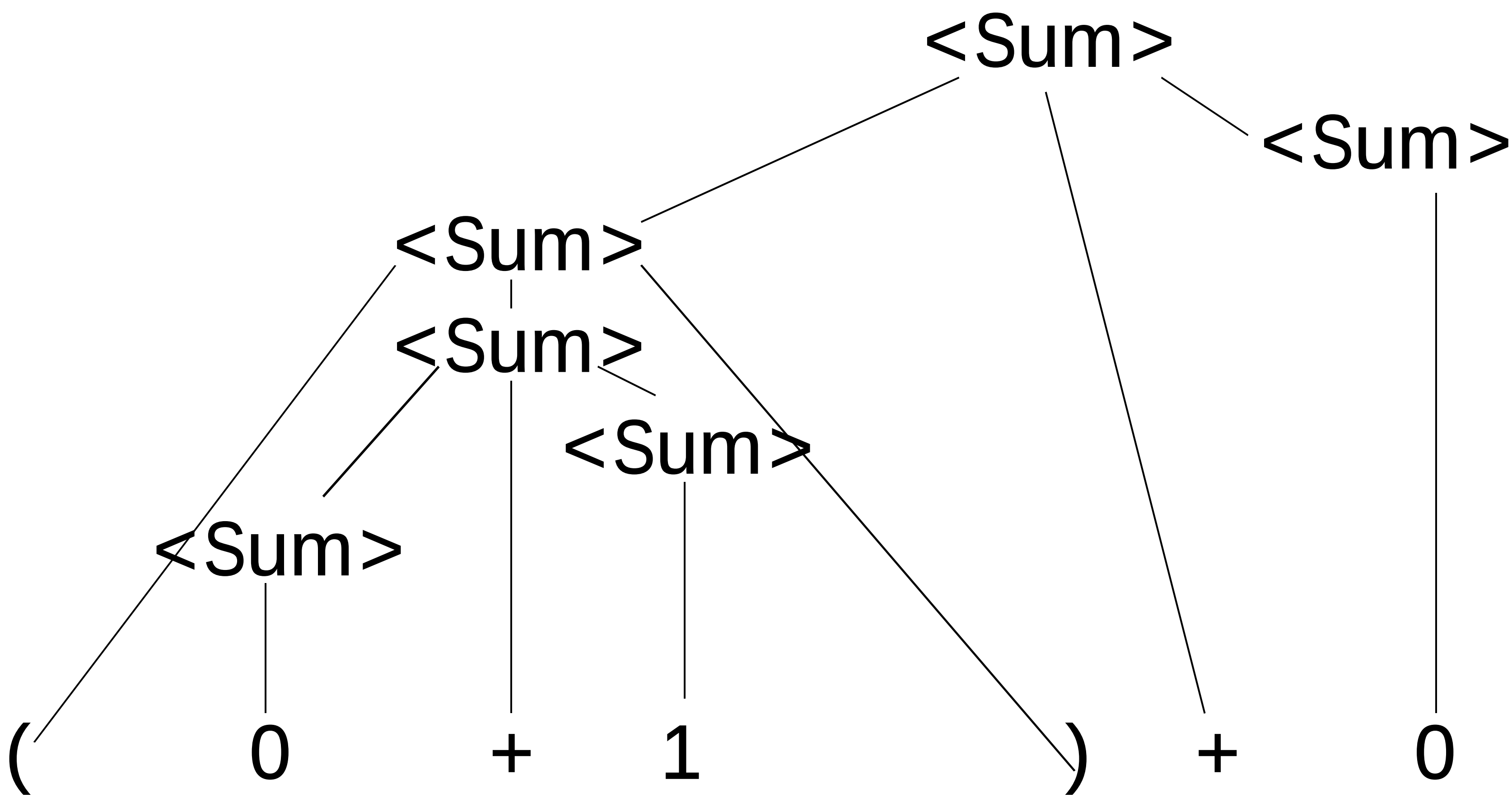
Example



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 * 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?

Using Ocamlyacc

- Input attribute grammar is put in file <*grammar*>.mly
- Execute

  ocamlyacc <*grammar*>.mly
- Produces code for parser in

  <*grammar*>.ml

  and interface (including type declaration for tokens) in

  <*grammar*>.mli

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 symbol
- %left *symbol … symbol*
- %right *symbol … symbol*
- %nonassoc *symbol … symbol*

  Associate precedences and associativities 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 values semantic attributes 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.mll)

```
{ (*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")))
```