Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- **Reminder:** We want to turn strings (code) into computer instructions
- Done in **phases**
  - Turn strings into abstract syntax trees (**parse**)
  - Translate abstract syntax trees into executable instructions (**interpret** or **compile**)
- Thursday, we showed much of **parsing**, including how to use a **parser generator**
- Today we will learn **the algorithm beneath** the generated parser

Objectives for Today (TODO update)

- **Reminder:** We want to turn strings (code) into computer instructions
- Done in **phases**
  - Turn strings into abstract syntax trees (**parse**)
  - Translate abstract syntax trees into executable instructions (**interpret** or **compile**)
- Thursday, we showed much of **parsing**, including how to use a **parser generator**
- Today we will learn **the algorithm beneath** the generated parser

Questions from last week?

Reminder: Implementing Parsers

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) }
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")))
```

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")))
```

How did the parser generator actually generate something that parses input strings like this, given the grammar we provided?

The Parsing Algorithm

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 rightmost
  - 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

Parsing Algorithm

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 rightmost
  - 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

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 rightmost
  - 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

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 rightmost
  - 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

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 rightmost
  - 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

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 rightmost
  - 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

More Details Later

LR Parsing Example

Example: Sums of 0s and 1s

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

Problem: How can we derive (0 + 1) + 0 : <Sum>?

Example: Sums of 0s and 1s

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

Problem: How can we derive (0 + 1) + 0 : <Sum>?

Work from the **bottom up**

LR Parsing Example

Example: Sums of 0s and 1s

<Sum>   =>

Work from the **bottom up**

- (0 + 1) + 0

Example: Sums of 0s and 1s

<Sum> =>

Work from the **bottom up**

(□ 0 + 1) + 0

= □ (0 + 1) + 0 **shift**

LR Parsing Example

Example: Sums of 0s and 1s

<Sum> =>

(0 □ + 1) + 0

= (□ 0 + 1) + 0 **shift**

= □ (0 + 1) + 0 shift

Example: Sums of 0s and 1s

<Sum> =>

Now we want to replace

	(0 □ + 1) + 0	
=	(□ 0 + 1) + 0	shift
=	□ (0 + 1) + 0	shift

Example: Sums of 0s and 1s

<Sum> =>

(<Sum> □ + 1) + 0

=> (0 □ + 1) + 0 **reduce**

= (□ 0 + 1) + 0 shift

= □ (0 + 1) + 0 shift

Example: Sums of 0s and 1s

<Sum> =>

Keep working **up**

(<Sum> □ + 1) + 0

=> (0 □ + 1) + 0 **reduce**

= (□ 0 + 1) + 0 shift

= □ (0 + 1) + 0 shift

Example: Sums of 0s and 1s

<Sum> =>

Keep working **up**

(<Sum> + □ 1) + 0
= (<Sum> □ + 1) + 0 **shift**
=> (0 □ + 1) + 0 reduce
= (□ 0 + 1) + 0 shift
= □ (0 + 1) + 0 shift

Example: Sums of 0s and 1s

<Sum> =>

	(<Sum> + 1 □) + 0	
=	(<Sum> + □ 1) + 0	**shift**
=	(<Sum> □ + 1) + 0	shift
=>	(0 □ + 1) + 0	reduce
=	(□ 0 + 1) + 0	shift
=	□ (0 + 1) + 0	shift

Example: Sums of 0s and 1s

<Sum> =>

Now what?

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

LR Parsing Example

Example: Sums of 0s and 1s

<Sum> =>

(<Sum> + <Sum> □) + 0

=> (<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: Sums of 0s and 1s

<Sum> =>

	(<Sum> □) + 0	
=>	(<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 Parsing Example

Example: Sums of 0s and 1s

<Sum> =>

	(<Sum>) □ + 0	
=	(<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: Sums of 0s and 1s

<Sum> =>

<Sum> □ + 0	
=> (<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: Sums of 0s and 1s

<Sum> =>

	<Sum> + □ 0	
=	<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: Sums of 0s and 1s

<Sum>	=>		
		<Sum> + 0 □	
	=	<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: Sums of 0s and 1s

<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: Sums of 0s and 1s

<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: Sums of 0s and 1s

<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

Questions so far?

Building the Parse Tree

Example: Sums of 0s and 1s

(0 + 1) + 0

Example: Sums of 0s and 1s

(0 + 1) + 0

LR Parsing Example

Example: Sums of 0s and 1s

(0 + 1) + 0

Example: Sums of 0s and 1s

<Sum>

(0 + 1) + 0

Example: Sums of 0s and 1s

<Sum>

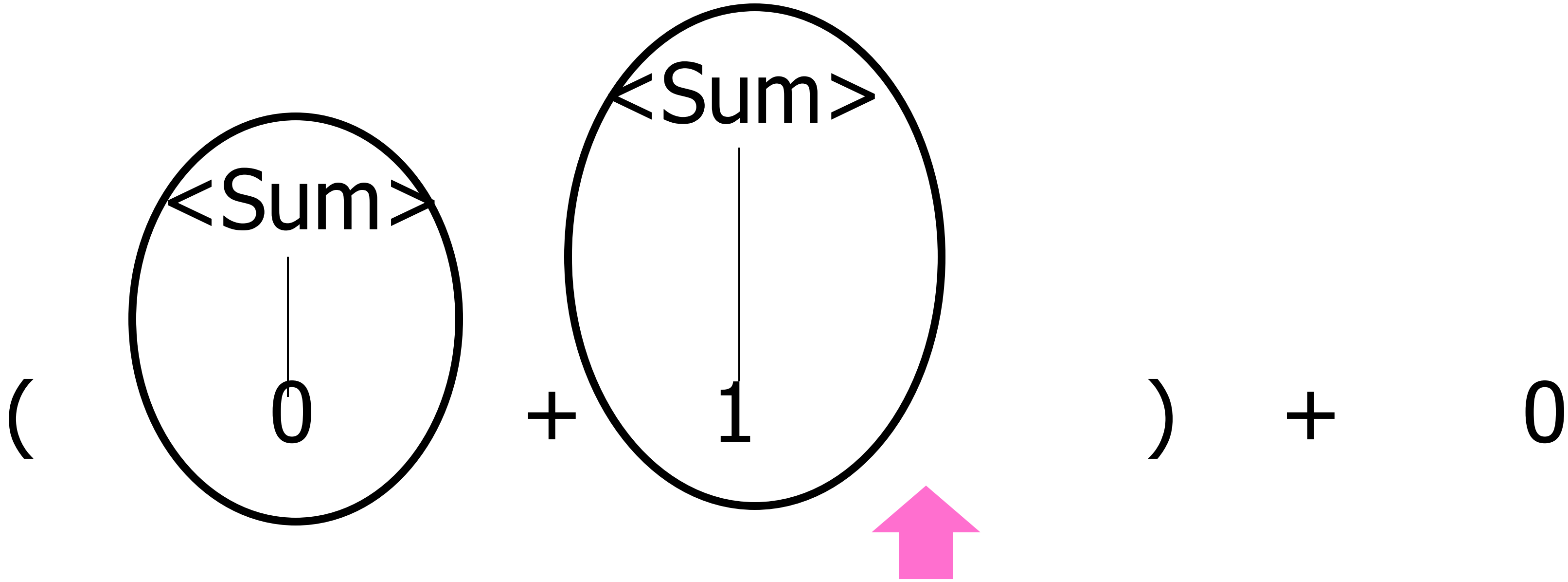
(0 + 1) + 0

Example: Sums of 0s and 1s

<Sum>

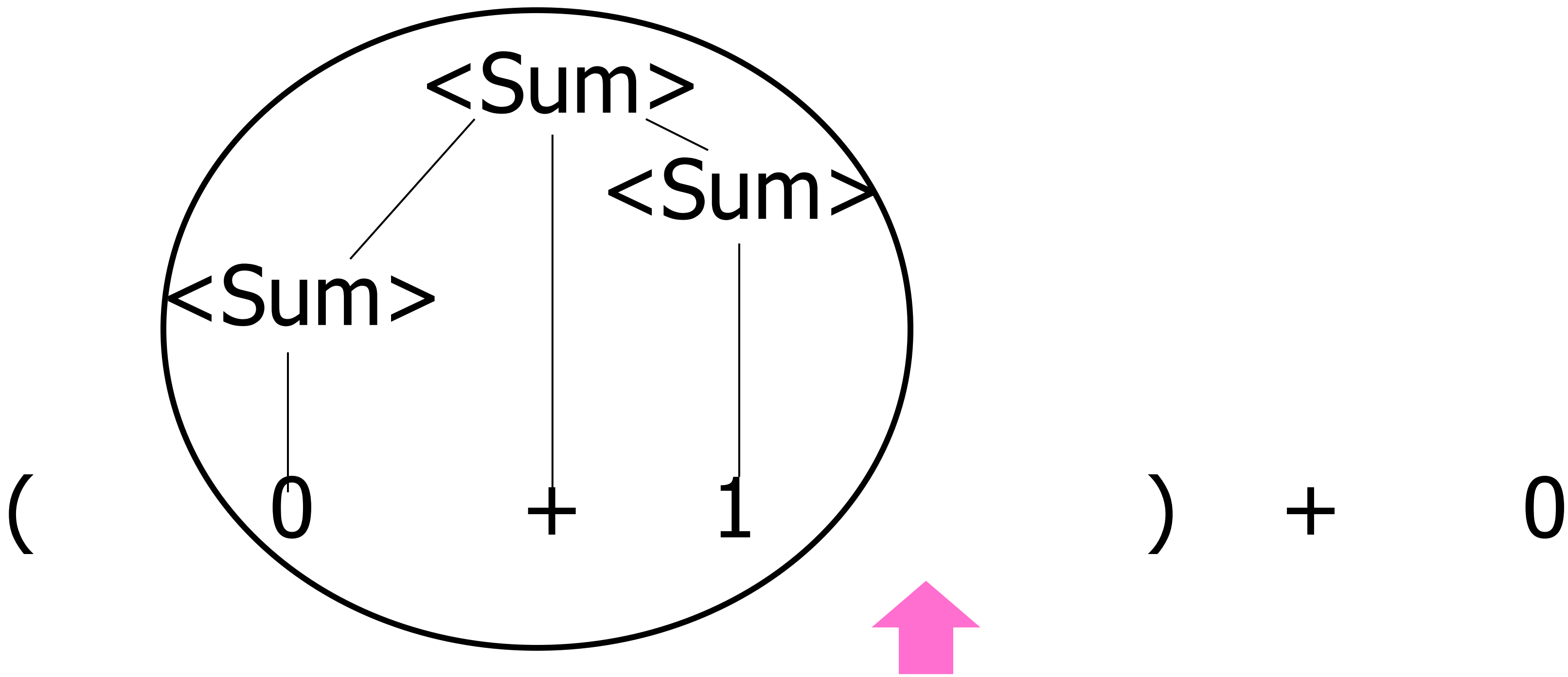
(0 + 1) + 0

Example: Sums of 0s and 1s

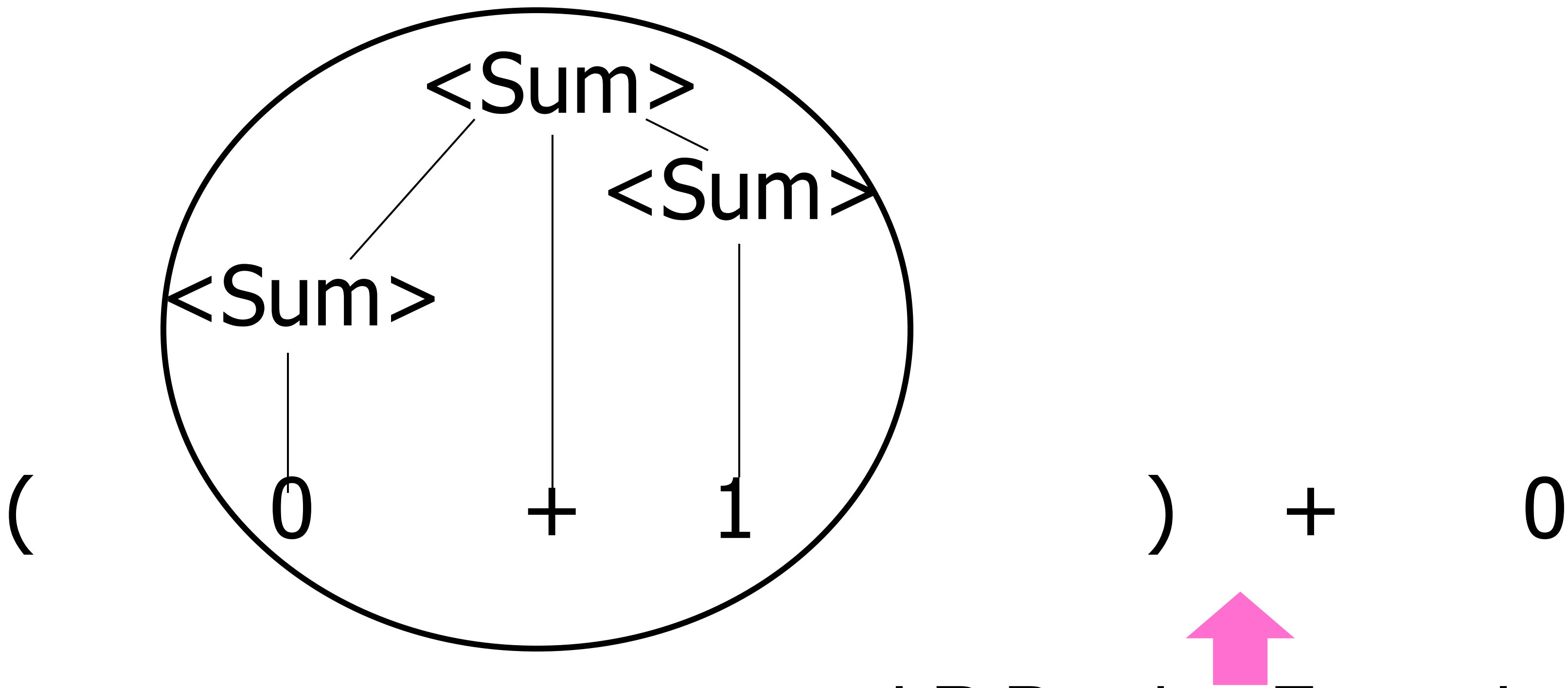


LR Parsing Example

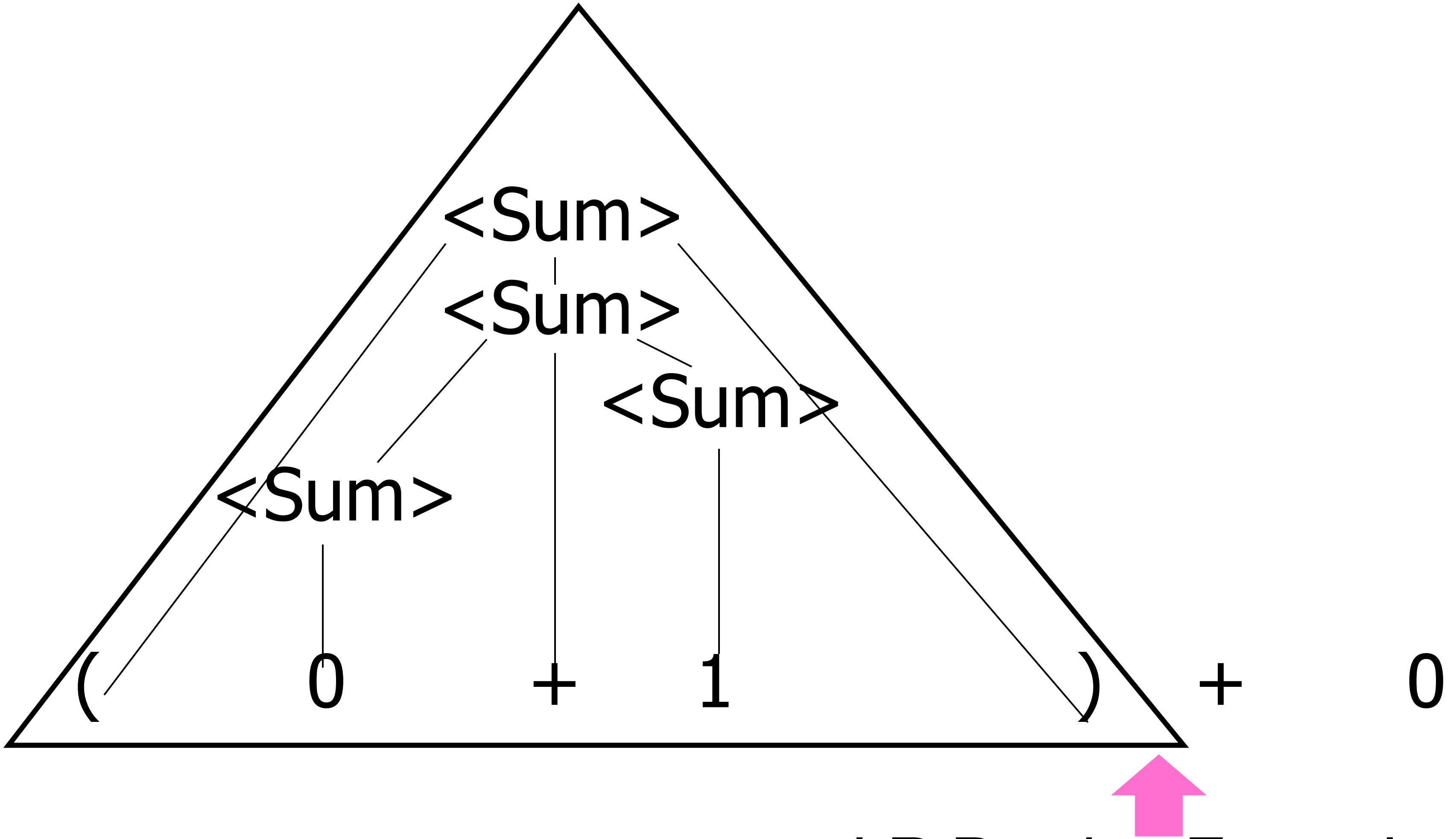
Example: Sums of 0s and 1s



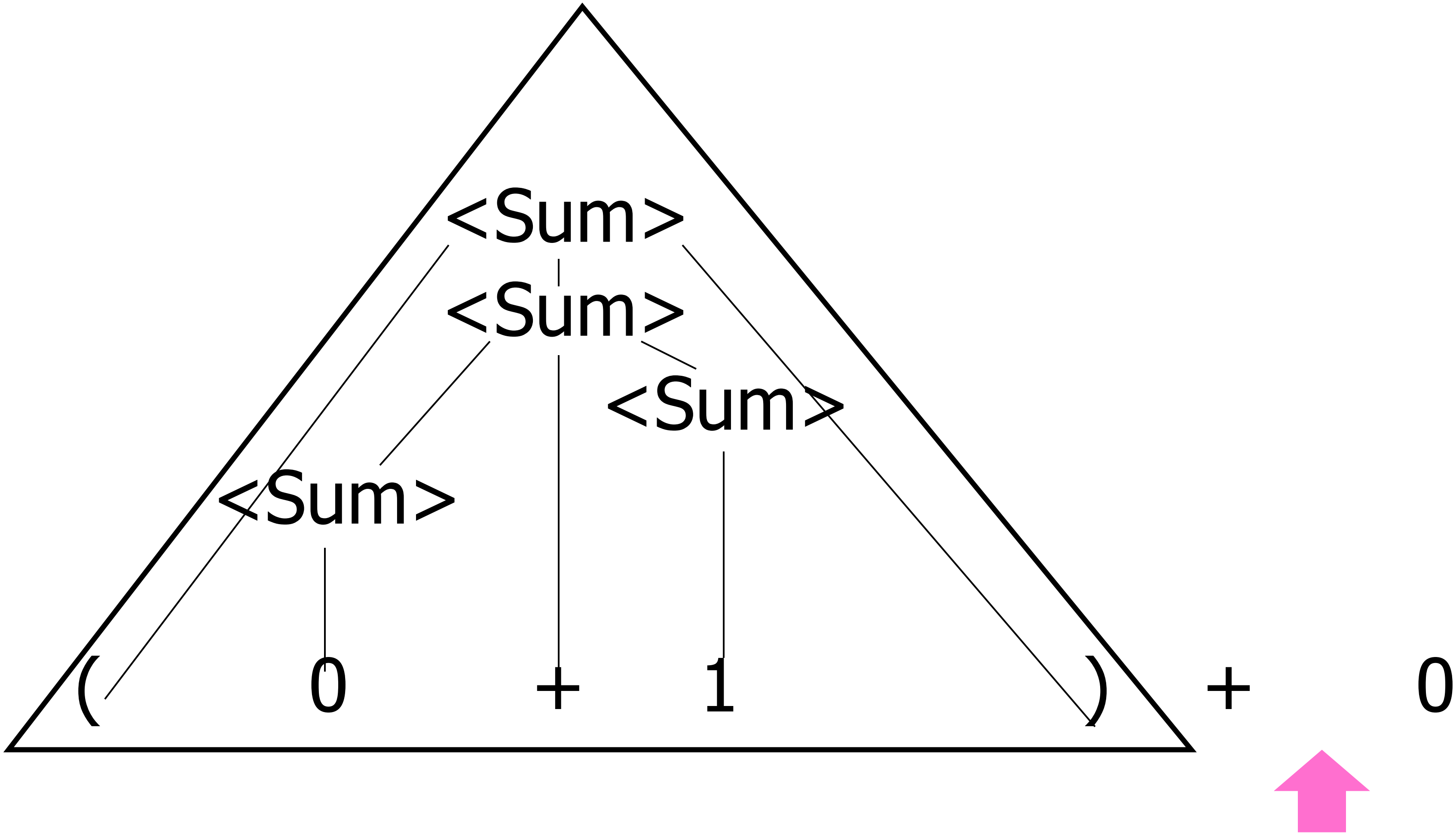
Example: Sums of 0s and 1s



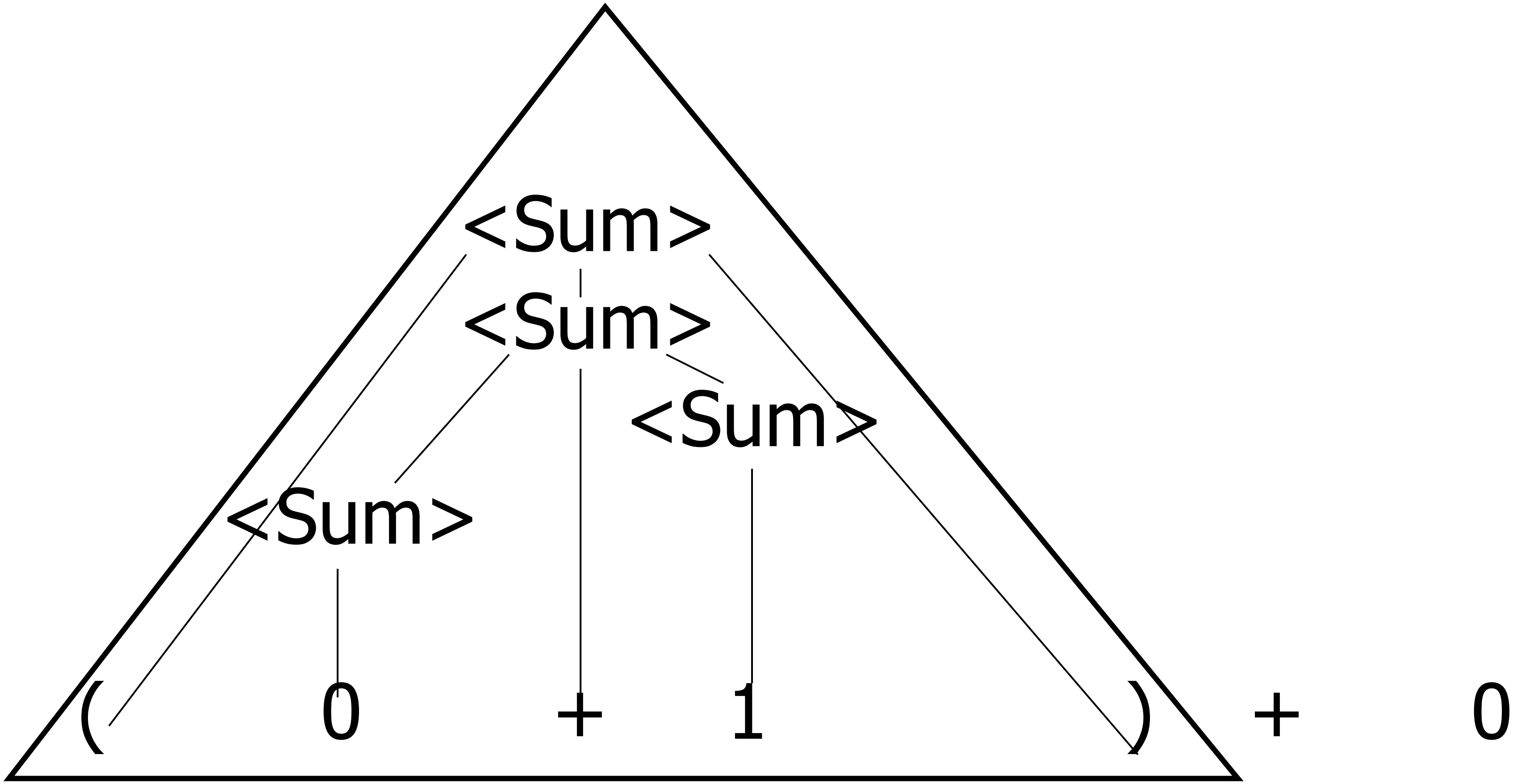
Example



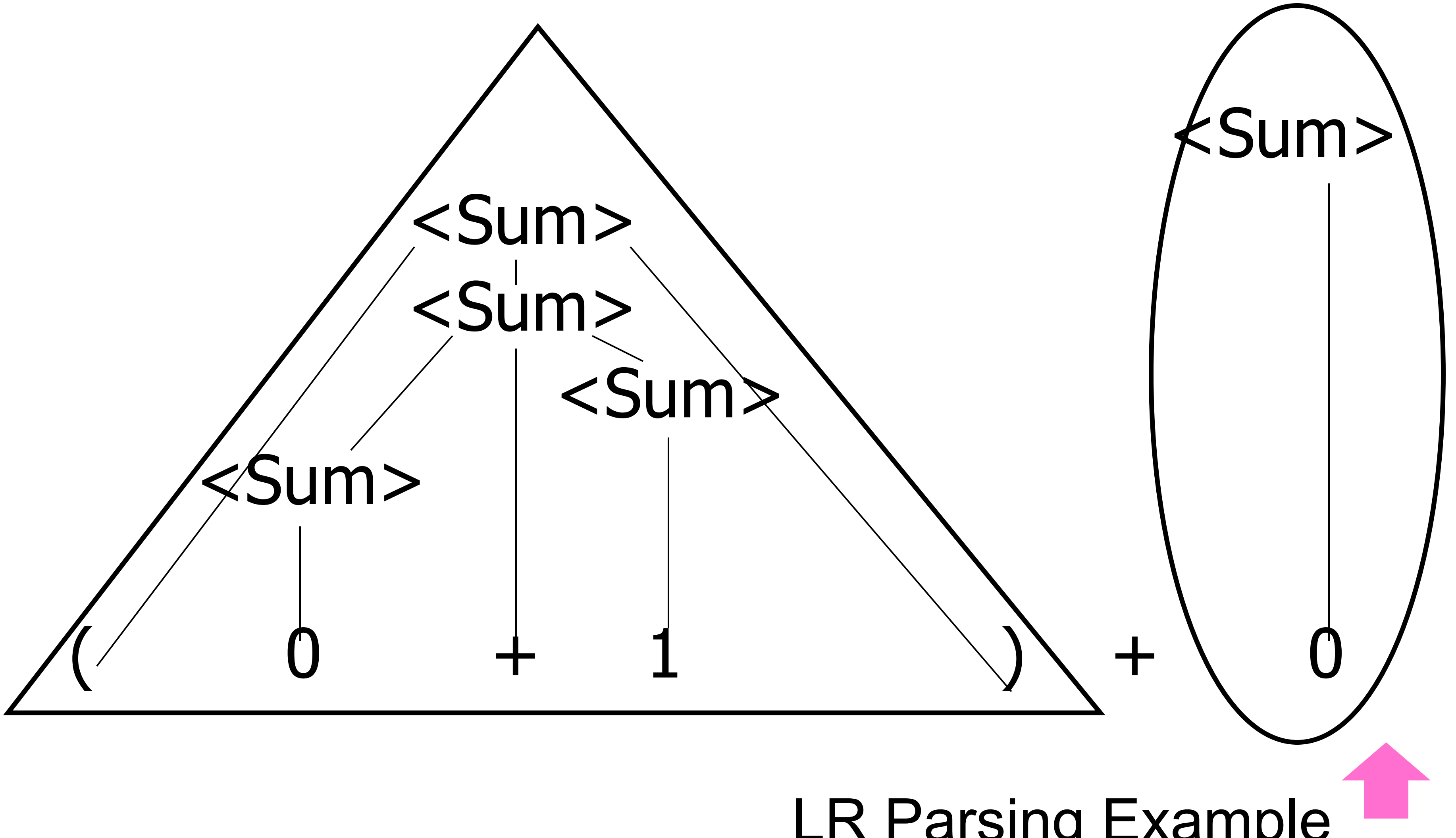
Example



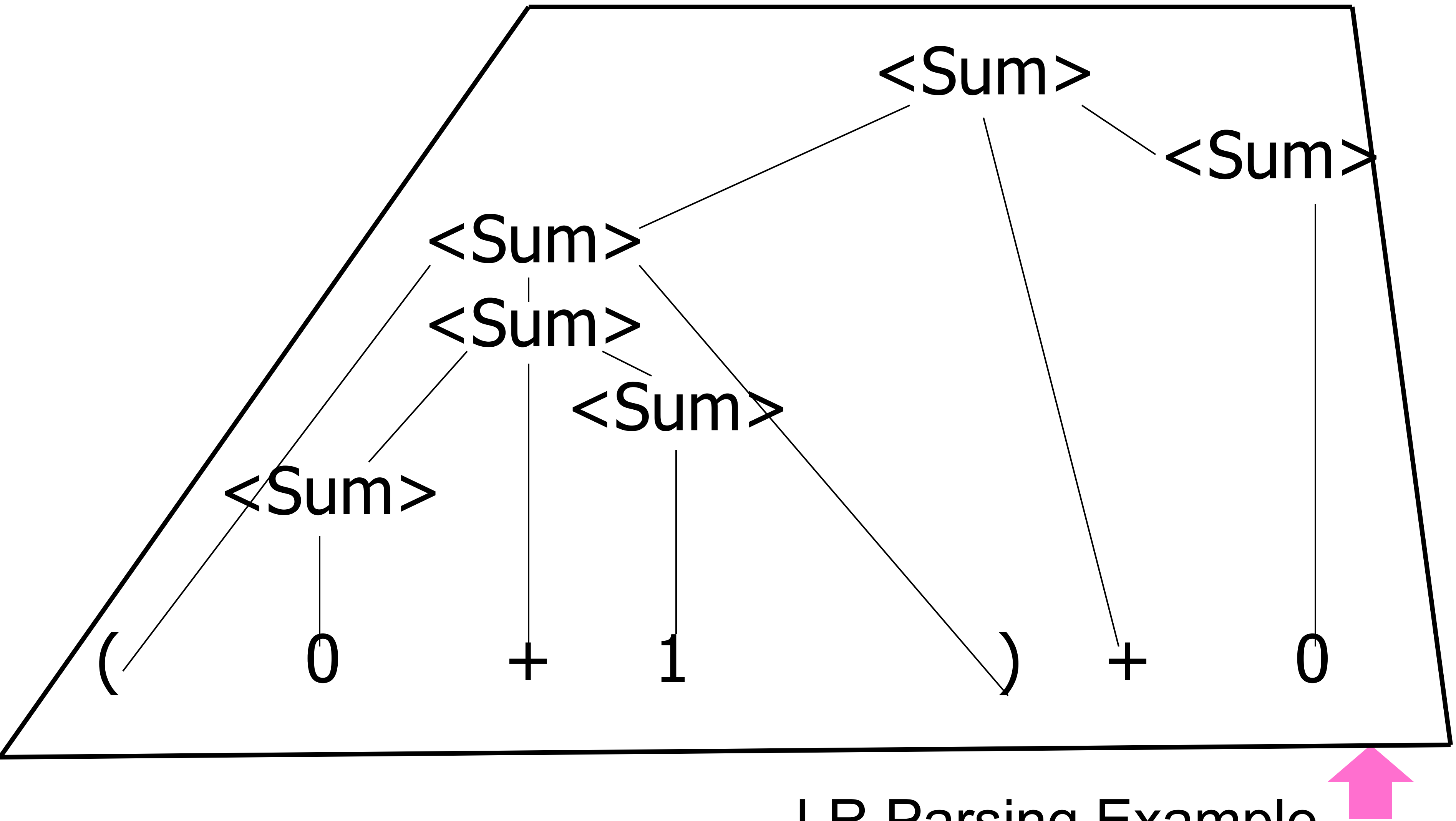
Example



Example

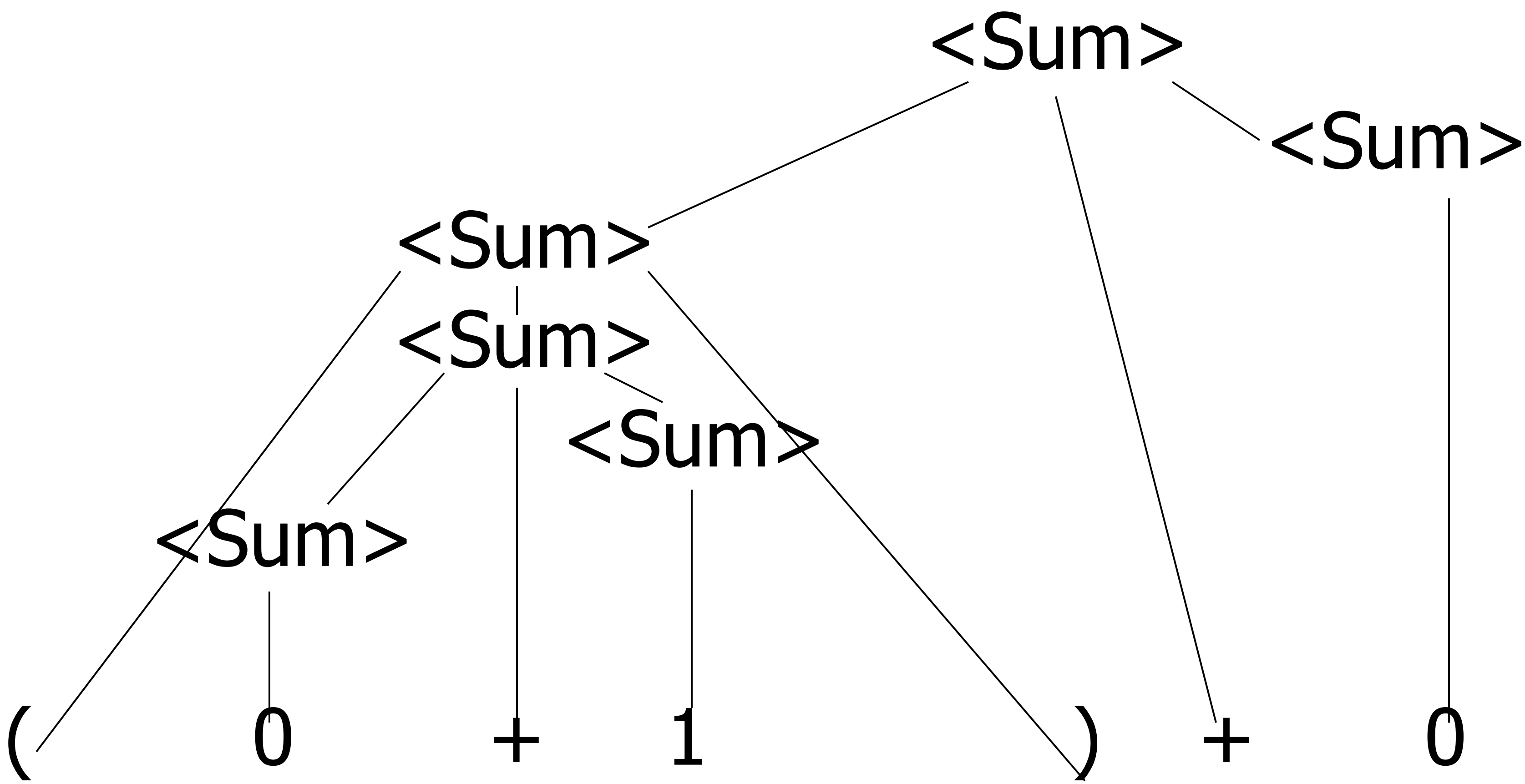


Example



LR Parsing Example

Example



Questions so far?

How LR Parsing Works

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

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*

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. Ensure 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

0. Ensure 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

0. Ensure 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

0. Ensure 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

0. Ensure 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 back 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

- AKA building the actual **parse tree** with the **values** it stores
- Add to each **reduce** a **rule** for calculating the new synthesized attribute from the component attributes
- Add to each **nonterminal** pushed onto the stack, the **attribute** calculated for it
- When performing a **reduce**,
  - gather the recorded attributes from each nonterminal popped from stack
  - Compute new attribute for nonterminal pushed onto stack

Adding Synthesized Attributes

- AKA building the actual **parse tree** with the **values** it stores
- Add to each **reduce** a **rule** for calculating the new synthesized attribute from the component attributes
- Add to each **nonterminal** pushed onto the stack, the **attribute** calculated for it
- When performing a **reduce**,
  - gather the recorded attributes from each nonterminal popped from stack
  - Compute new attribute for nonterminal pushed onto stack

Questions so far?

Dealing with Ambiguity

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: Sums of 0s and 1s

	□ 0 + 1 + 0	shift
->	0 □ + 1 + 0	reduce
->	<Sum> □ + 1 + 0	shift
->	<Sum> + □ 1 + 0	shift
->	<Sum> + 1 □ + 0	reduce
->	<Sum> + <Sum> □ + 0	

Example: Sums of 0s and 1s

□ 0 + 1 + 0 shift
-> 0 □ + 1 + 0 reduce
-> <Sum> □ + 1 + 0 shift
-> <Sum> + □ 1 + 0 shift
-> <Sum> + 1 □ + 0 reduce
-> <Sum> + <Sum> □ + 0

Example: Sums of 0s and 1s

	□ 0 + 1 + 0	shift
->	0 □ + 1 + 0	reduce
->	<Sum> □ + 1 + 0	shift
->	<Sum> + □ 1 + 0	shift
->	<Sum> + 1 □ + 0	reduce
->	<Sum> + <Sum> □ + 0	

Example: Sums of 0s and 1s

□ 0 + 1 + 0	shift
-> 0 □ + 1 + 0	reduce
-> <Sum> □ + 1 + 0	shift
-> <Sum> + □ 1 + 0	shift
-> <Sum> + 1 □ + 0	reduce
-> <Sum> + <Sum> □ + 0	

Example: Sums of 0s and 1s

□ 0 + 1 + 0	shift
-> 0 □ + 1 + 0	reduce
-> <Sum> □ + 1 + 0	shift
-> <Sum> + □ 1 + 0	shift
-> <Sum> + 1 □ + 0	reduce
-> <Sum> + <Sum> □ + 0	

Example: Sums of 0s and 1s

□ 0 + 1 + 0	shift
-> 0 □ + 1 + 0	reduce
-> <Sum> □ + 1 + 0	shift
-> <Sum> + □ 1 + 0	shift
-> <Sum> + 1 □ + 0	reduce
-> <Sum> + <Sum> □ + 0	

Example: Sums of 0s and 1s

□ 0 + 1 + 0 shift
-> 0 □ + 1 + 0 reduce
-> <Sum> □ + 1 + 0 shift
-> <Sum> + □ 1 + 0 shift
-> <Sum> + 1 □ + 0 reduce
-> <Sum> + <Sum> □ + 0

Do we **shift** or **reduce**?
We could do either.

Example: Sums of 0s and 1s

□ 0 + 1 + 0	shift
-> 0 □ + 1 + 0	reduce
-> <Sum> □ + 1 + 0	shift
-> <Sum> + □ 1 + 0	shift
-> <Sum> + 1 □ + 0	reduce
-> <Sum> + <Sum> □ + 0	

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 □	

Example

S ::= A | aB
A ::= abc
B ::= bc

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Example

S ::= A | aB
A ::= abc
B ::= bc

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Example

S ::= A | aB
A ::= abc
B ::= bc

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Example

S ::= A | aB
A ::= abc
B ::= bc

Which rule to reduce by?

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Example

S ::= A | **aB**
A ::= abc
B ::= bc

Which rule to reduce by?

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Example

S ::= A | aB
A ::= abc
B ::= bc

Which rule to reduce by?

□ abc	shift
a □ bc	shift
ab □ c	shift
abc □	

Questions?

Extra time?

Disambiguate <Sum>again, then run algorithm by hand on some strings to get shift/reduce sequences.

Next Class: More Disambiguation

Next Class

- **WA8** due next **Thursday**
- **MP9** due next **Tuesday**
- **Please sign up with CBTF** for **Midterm 3**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help