Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2024/CS421C

Based on slides by Elsa Gunter, which are based in part on previous slides by Mattox Beckman and updated by Vikram Adve and Gul Agha

- Last time we were doing context-free grammars
- Some grammars can have ambiguity (multiple parse trees)
- We will continue with disambiguation on Thursday.
- Today, let's see how we can parse "good" grammars

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!

Where do we stand?

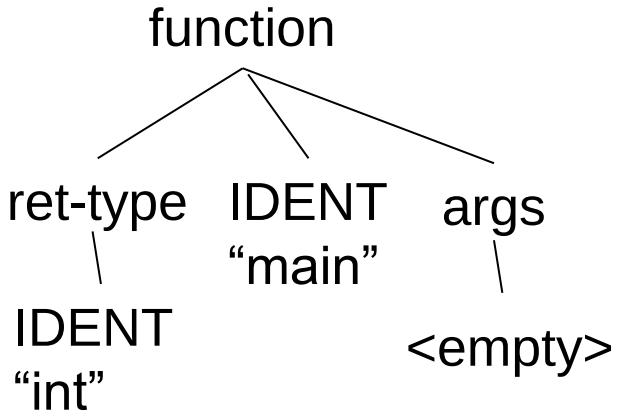
- Conceptually:

Input: “int main () { }”

Lexer*: [“int”, “main”, “(“, “)”, “{“, “}”]

Lexer: [IDENT “int”, IDENT “main”, LPAREN, RPAREN, LCURLY, RCURLY]

Parser: turns this list into a tree, so we can more easily work with it.



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
```

Parser's role

- Parser's role: Converts text from the grammar to a datastructure defined by this base type:

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

- Example: "X + Y"

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 - 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}
```

The lexer uses the tokes defined in the parser file (see next slide)

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)

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

```
expr:
    term①
      { Term_as_Expr $1 }

  | term① Plus_token expr③
      { Plus_Expr ($1, $3) }

  | term① Minus_token expr③
          { 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)

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

```
term:
    factor
      { Factor_as_Term $1 }
  | factor Times_token term
      { Mult_Term ($1, $3) }
  | factor Divide_token term
      { Div_Term ($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)
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

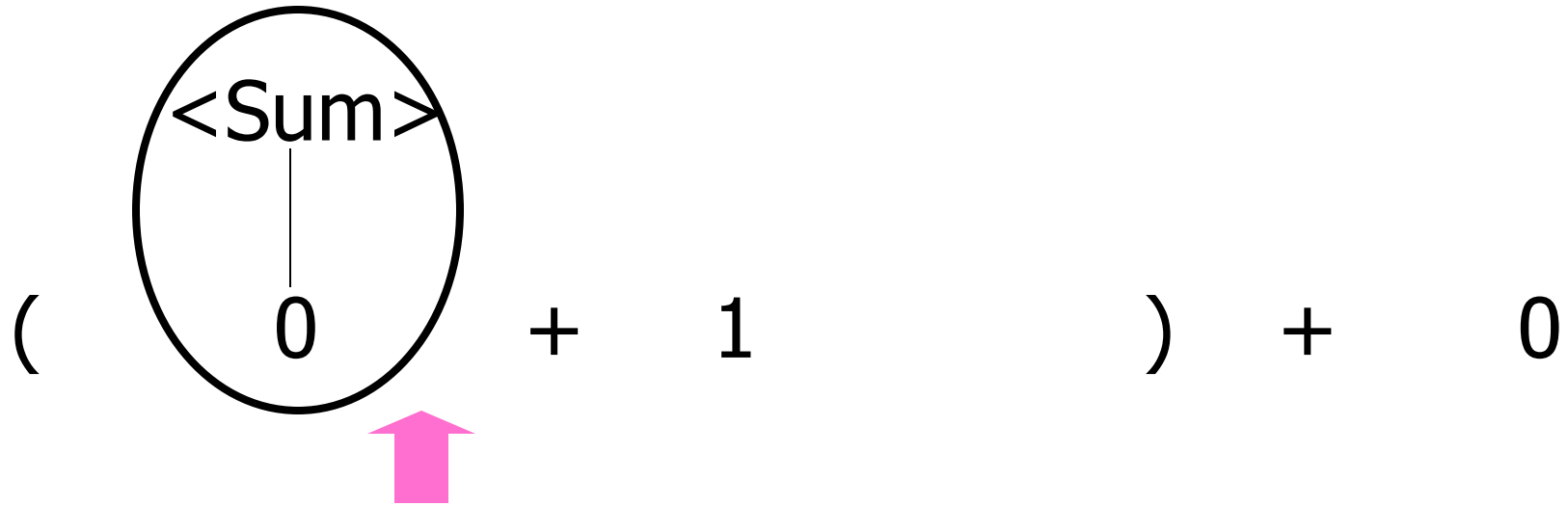
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>

(0 + 1) + 0

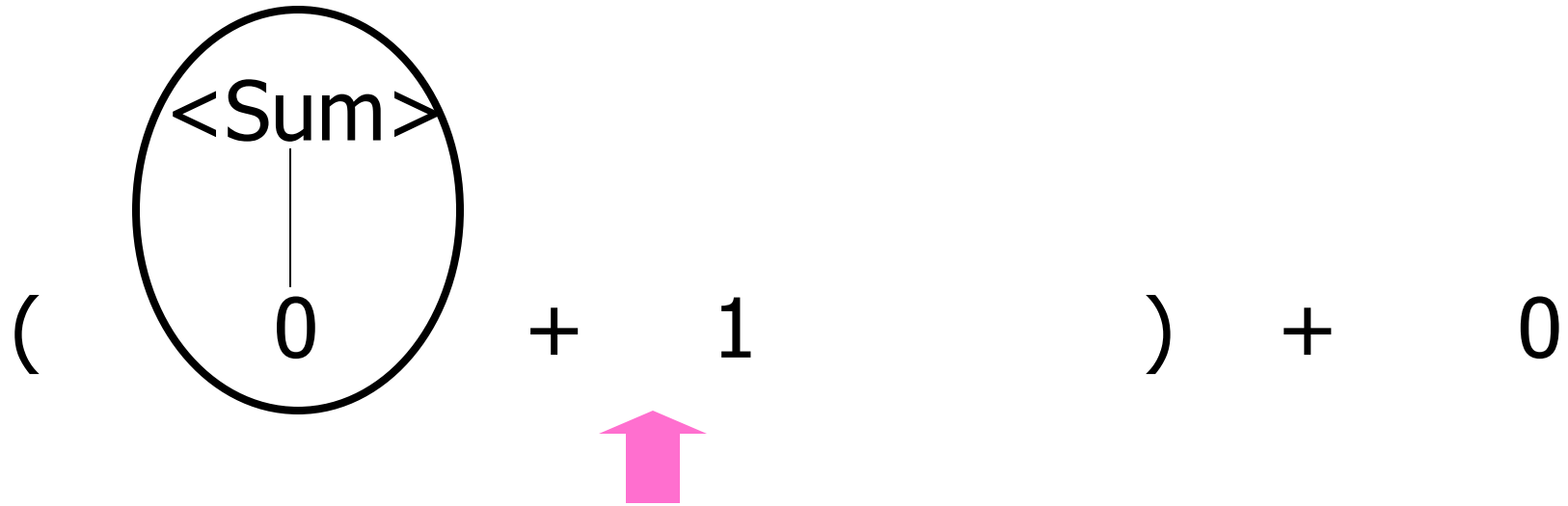
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>

(0 + 1) + 0

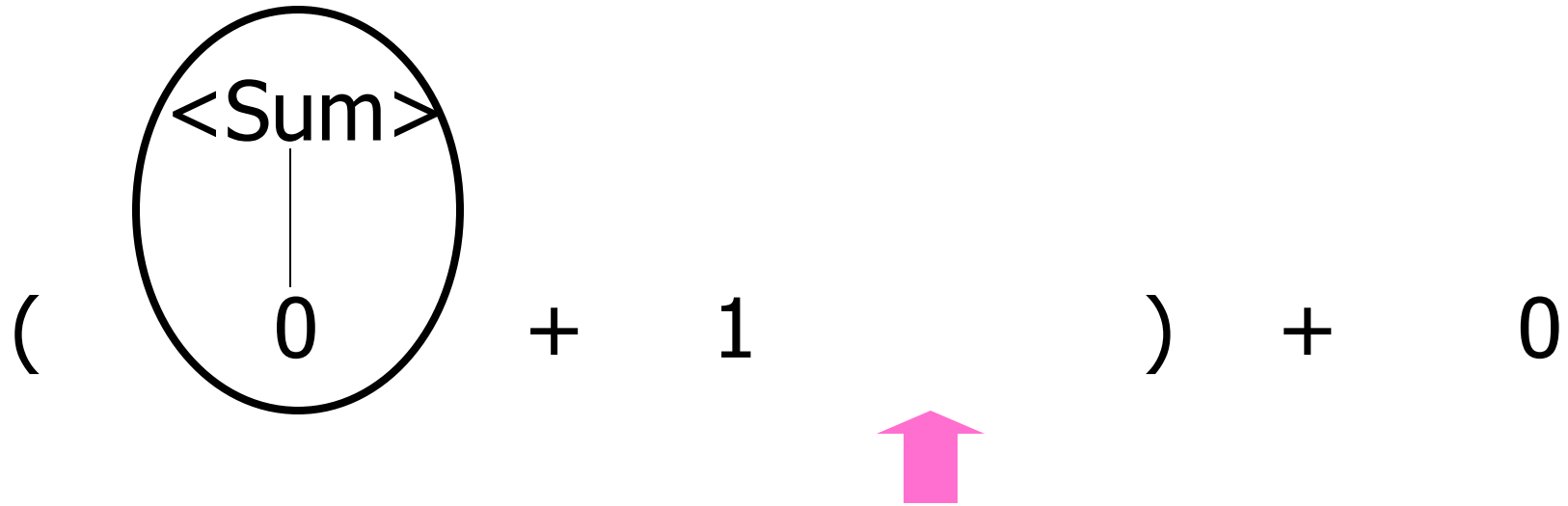
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



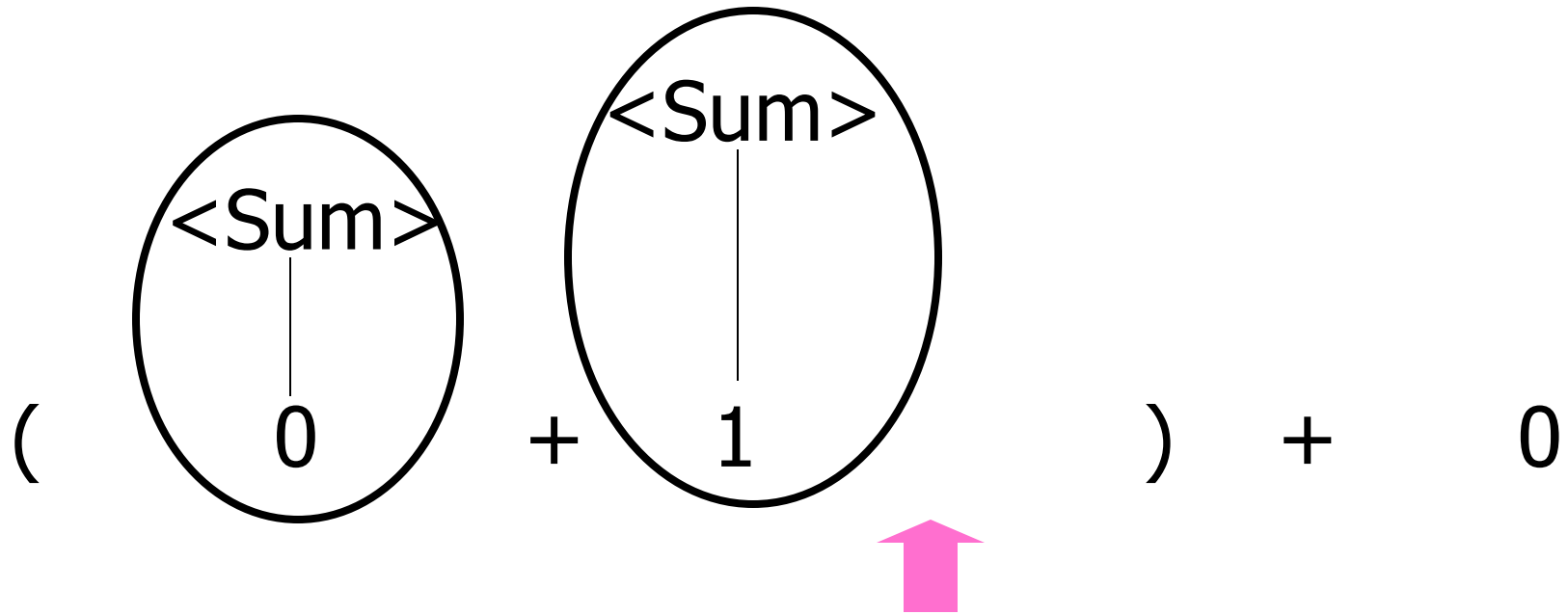
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



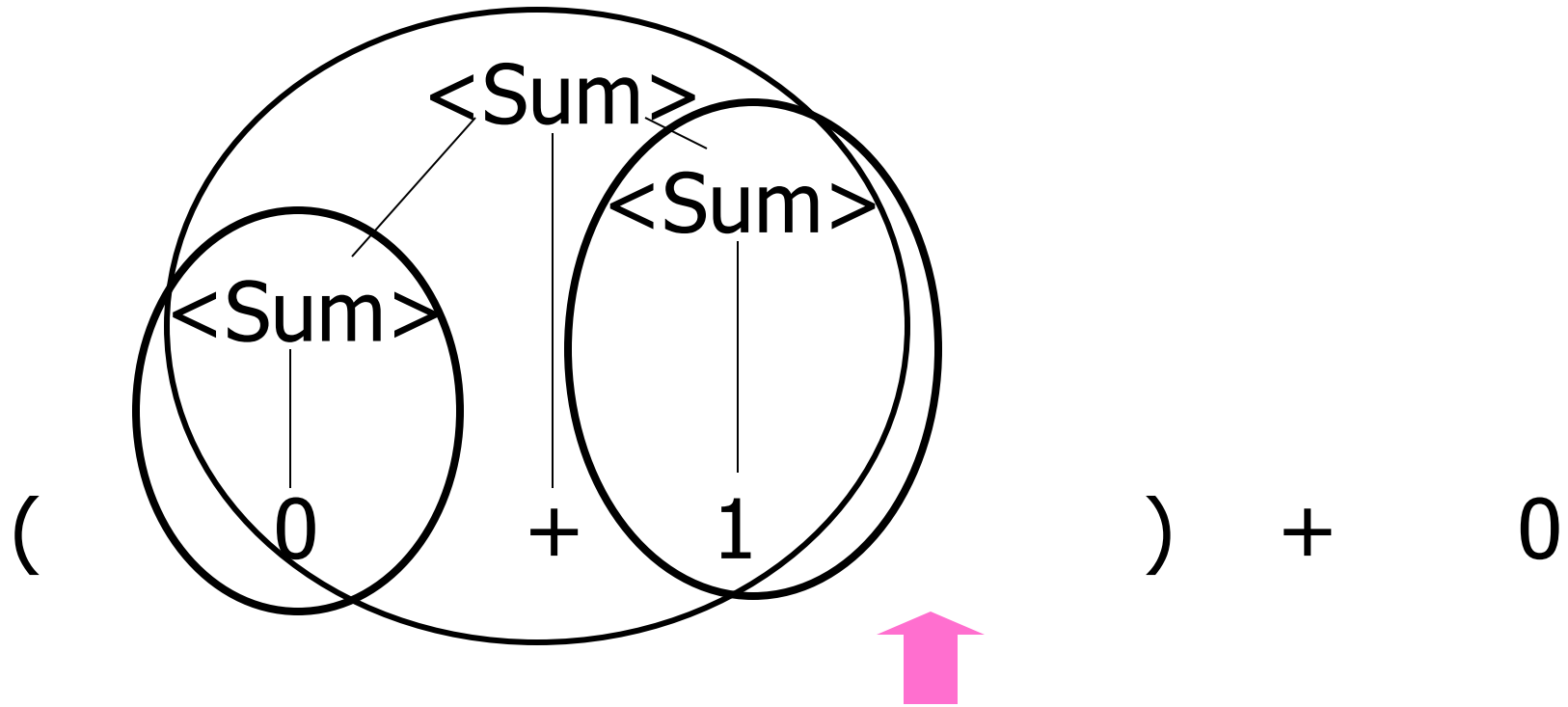
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>



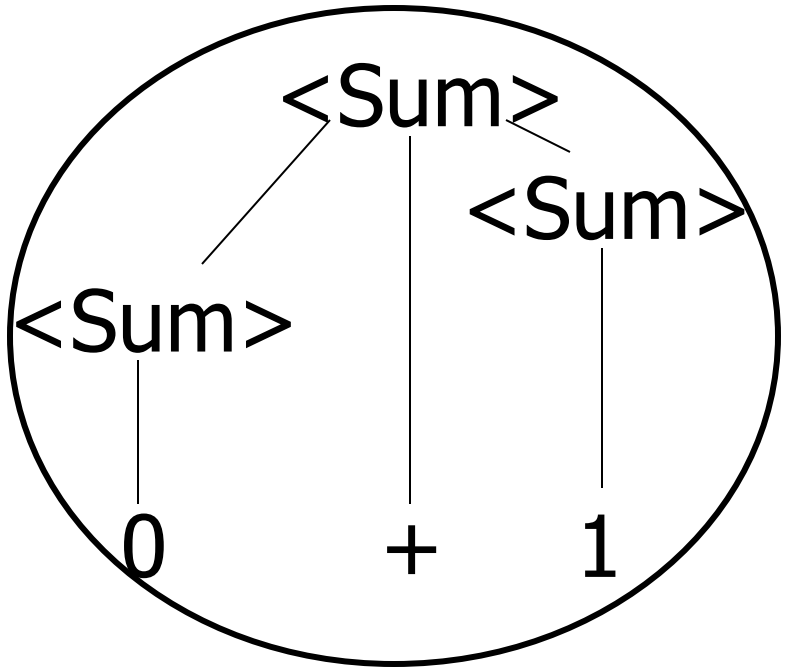
Example: <Sum> ::= 0 | 1 | (<Sum>)
| <Sum> + <Sum>



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

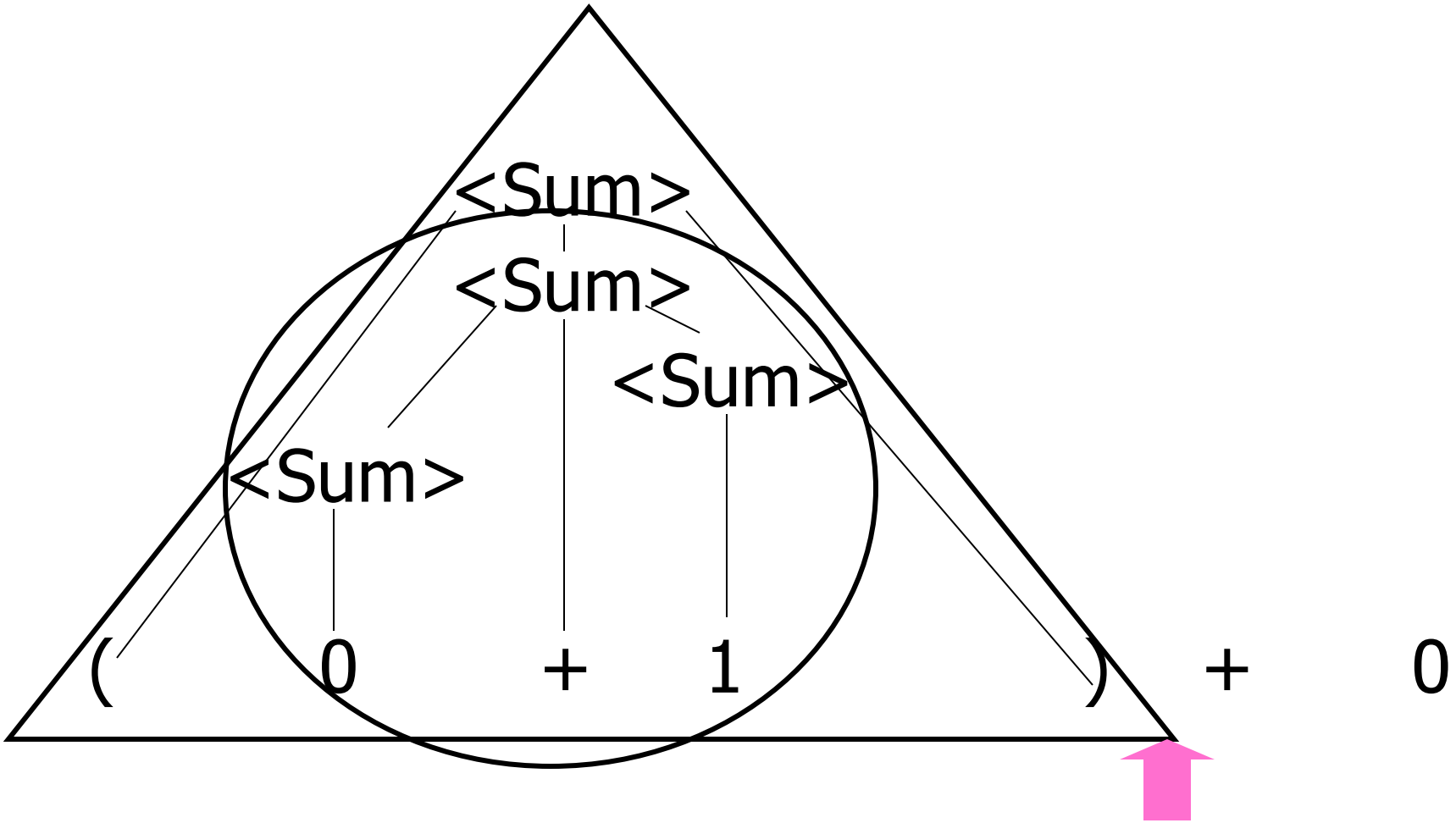


Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

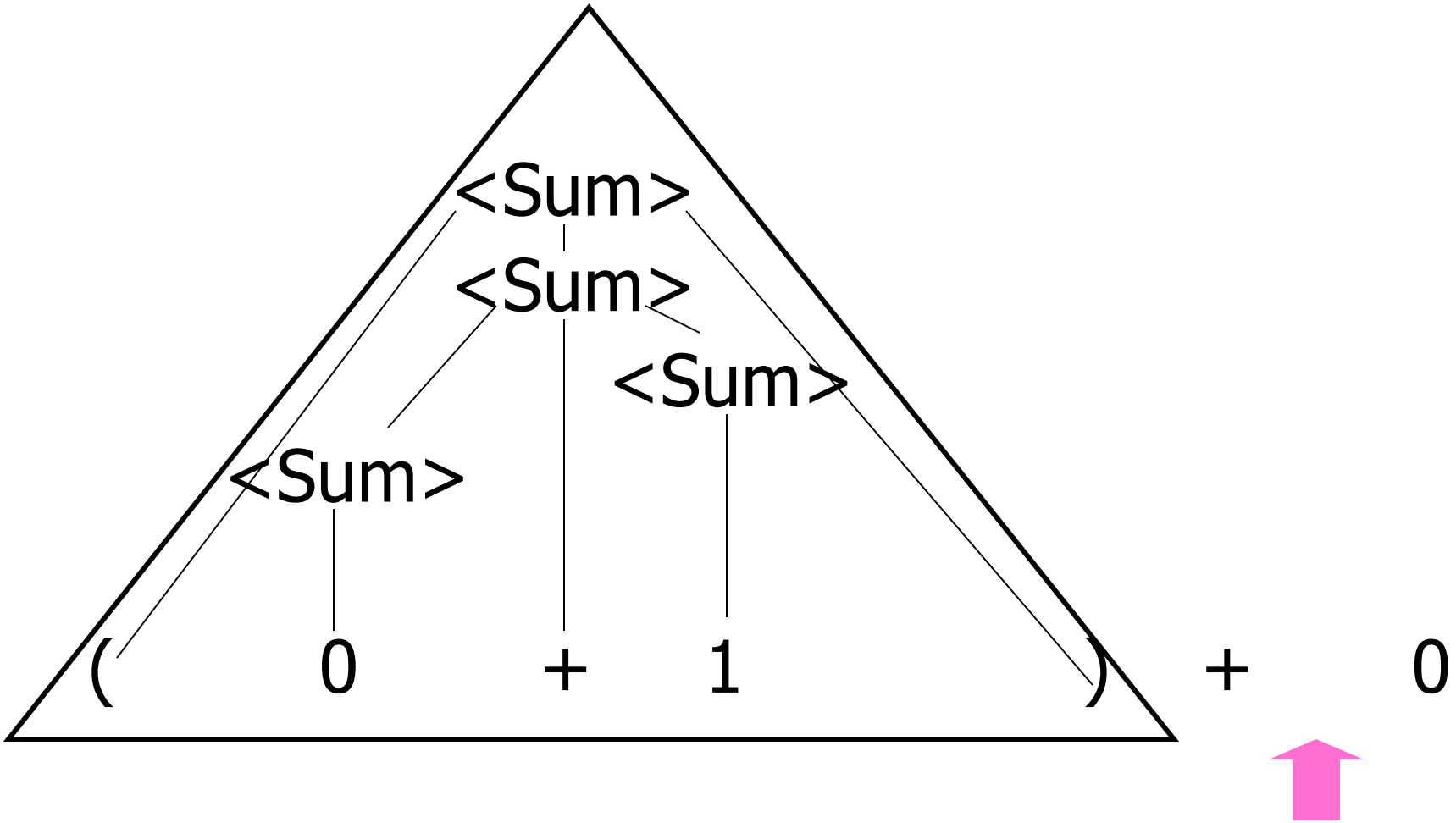


(0 + 1) + 0

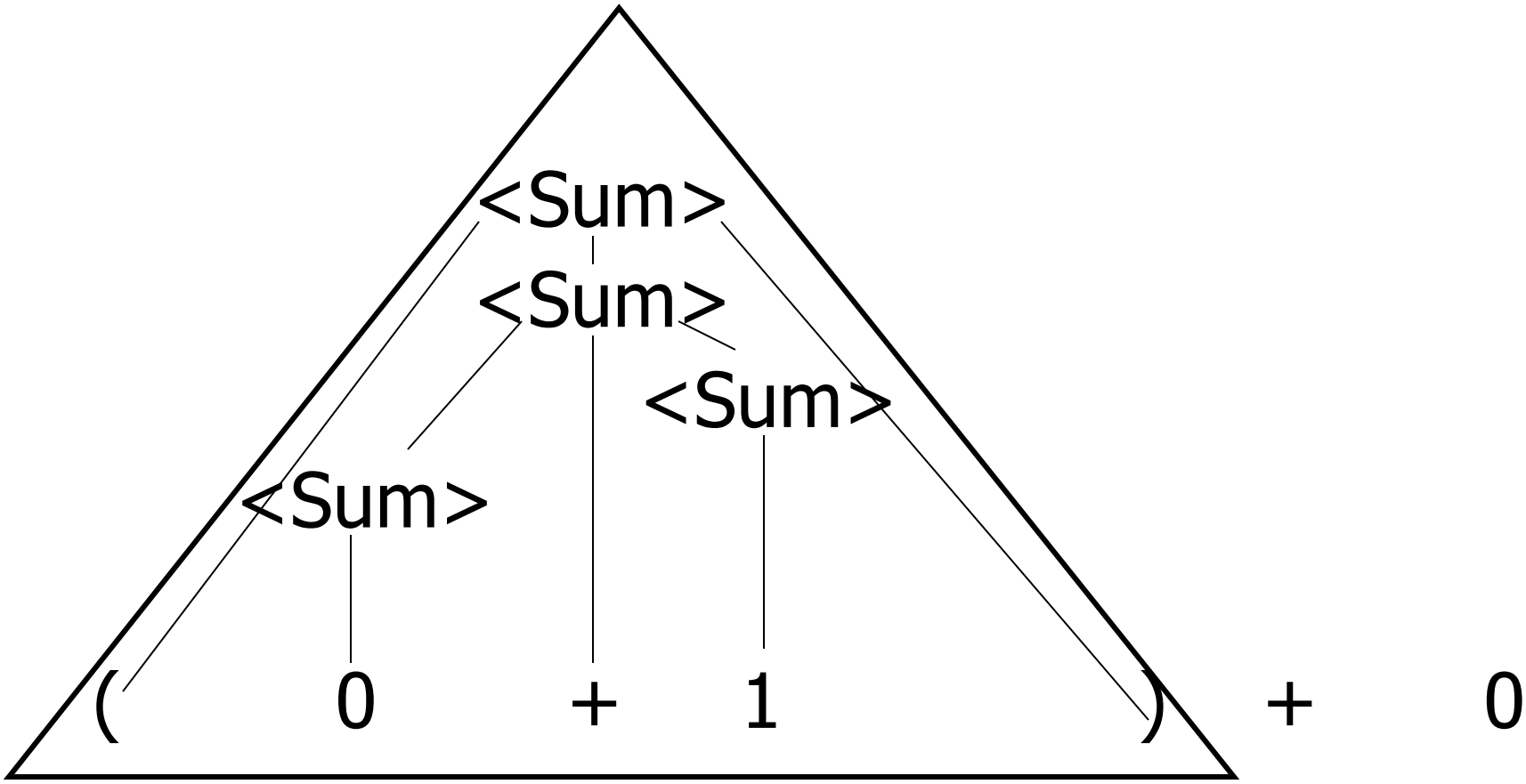
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



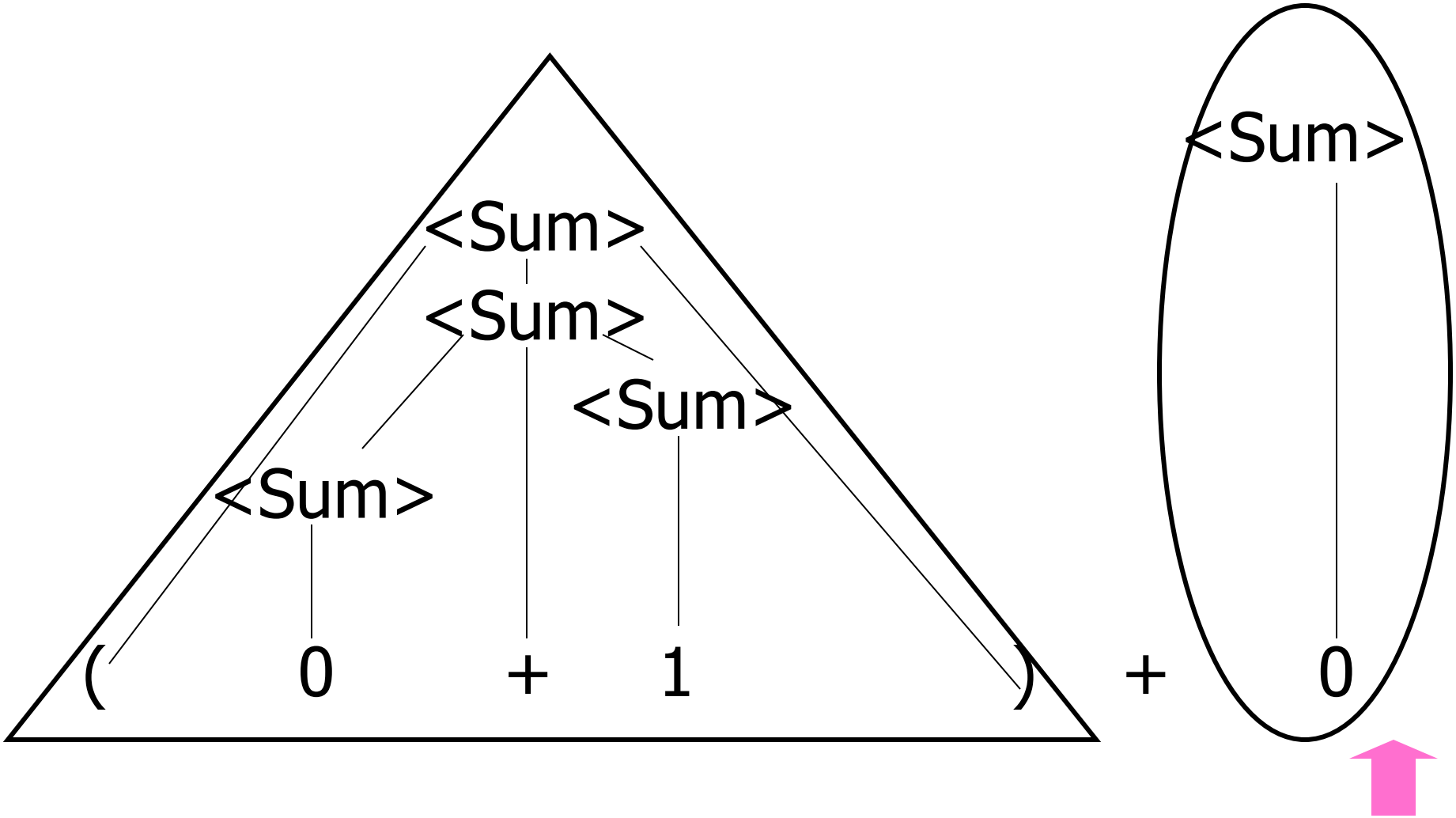
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



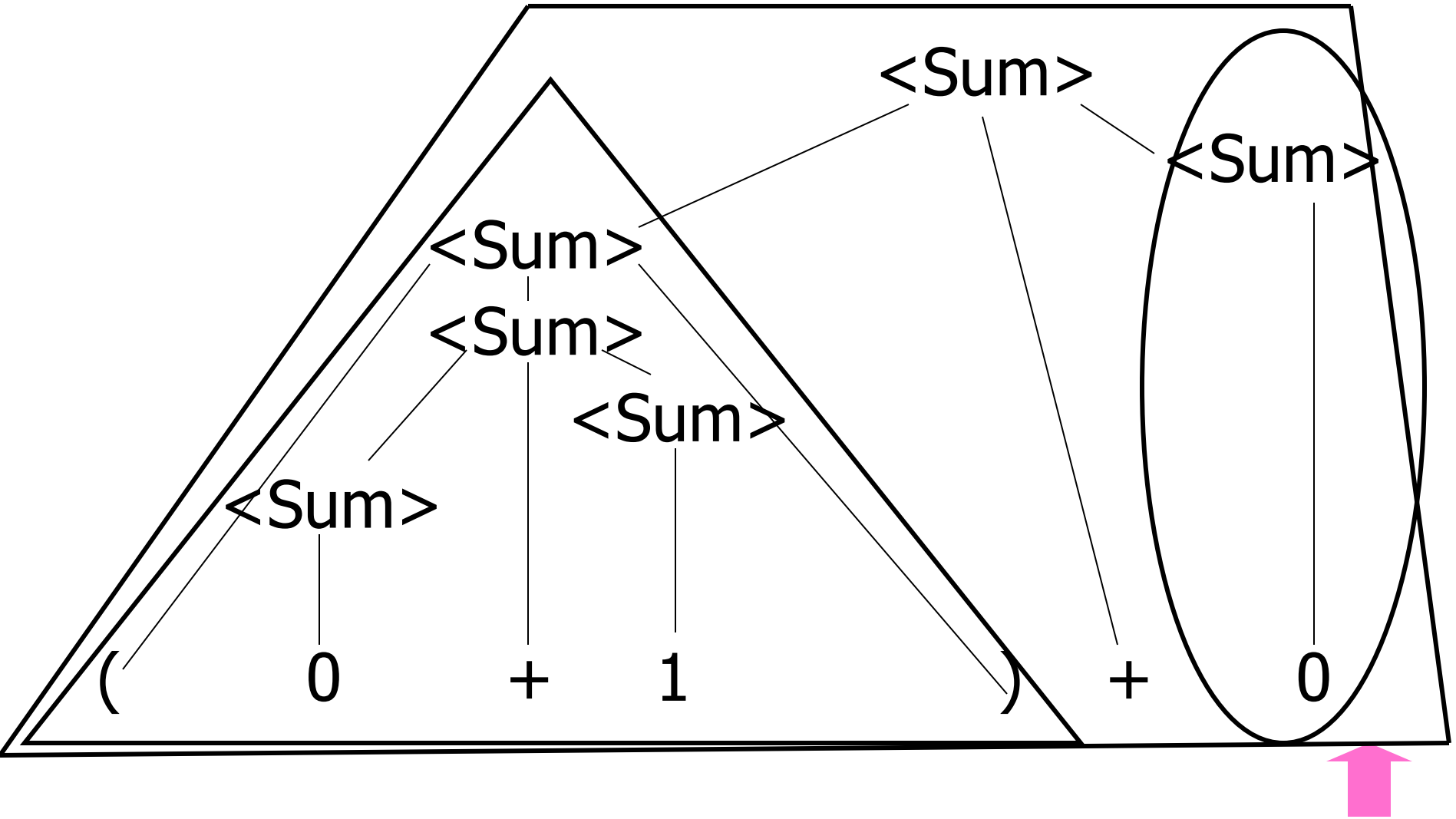
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



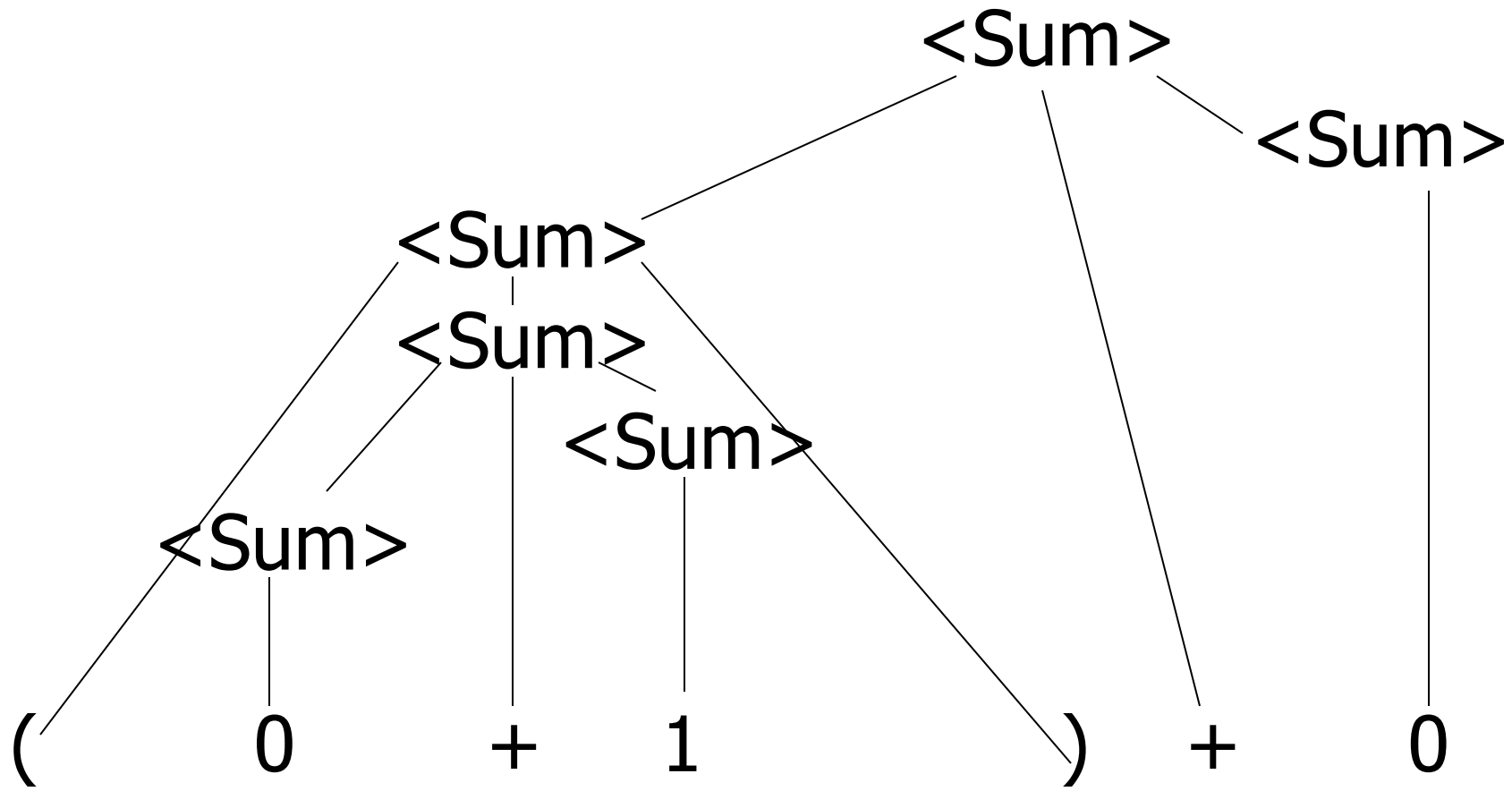
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



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



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



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. 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 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

Recall: 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*)

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

<Sum> =>

= (0 + 1) + 0 ?

stack input stream

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

<Sum> =>

= □ (0 + 1) + 0 shift

stack input stream

Recall: 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

From: □ (0 + 1) + 0

To: (□ 0 + 1) + 0

stack input stream

We don't visualize putting state(m) here…

Recall: 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

If we wanted to visualize that state(m) on the stack it would look like this…

From: □ (0 + 1) + 0

To: (<state(m)> □ 0 + 1) + 0

stack

input stream

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

<Sum> =>

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

stack input stream

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

<Sum> =>

= (□ 0 + 1) + 0 shift

= □ (0 + 1) + 0 shift

stack input stream

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

<Sum> =>

=> (0 □ + 1) + 0 ?
= (□ 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

Recall: 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	?
= (<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 ?
=> (<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

Recall: 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

- Back to Ambiguous Grammars…

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!

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>

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

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

- **Problem:** shift or reduce?

  -> <Sum> + <Sum> □ + 0 ??

- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

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

- **Problem:** shift or reduce?

  -> <Sum> + <Sum> □ + 0 ??

- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

-> <Sum> + <Sum> □ reduce

-> <Sum> + <Sum> + <Sum> □ reduce

-> <Sum> + <Sum> + □ 0 shift

-> <Sum> + <Sum> □ + 0 shift

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

- **Problem:** shift or reduce?

  -> <Sum> + <Sum> □ + 0 ??

- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

-> <Sum> + <Sum> □ reduce

-> <Sum> + □ 0 shift

-> <Sum> □ + 0 shift

-> <Sum> + <Sum> □ + 0 reduce

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

- **Problem:** shift or reduce?

  -> <Sum> + <Sum> □ + 0 ??

- 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

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

- Back to Grammar Disambiguation…

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

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?

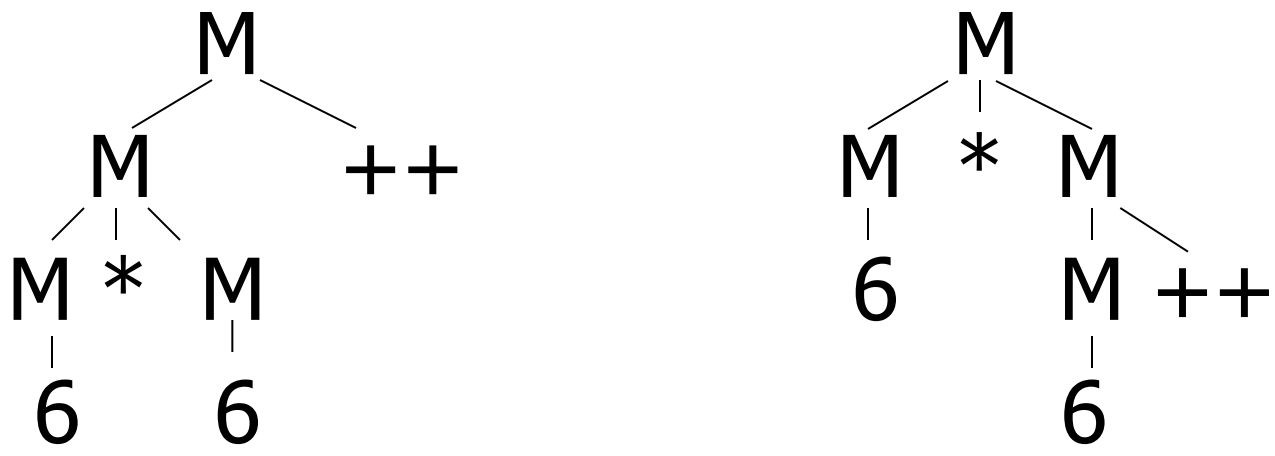
```
<exp>     ::= <factor>
            |  <factor> +  <exp> | <factor> - <exp>

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

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

More Disambiguating Grammars

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



M ::= M * M | (M) | M ++ | 6

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

M ::= M * M | (M) | M ++ | 6

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

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6

- M :: = M++ | StarExp | (M) | 6
- What is StarExp
- It is everything that parses as a * and can't parse as a ++
- But what is the associativity of *?
- Class chose left

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | (M) | 6
- StarExp ::= PossStar * NoStarNoPlusPlus
- What is PossStar? It could it be a *, but it also doesn't have to be.
- Can it be ++? YES! It can be anything
- It is M !

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | (M) | 6
- StarExp ::= M * NoStarNoPlusPlus

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | (M) | 6
- StarExp ::= M * NoStarNoPlusPlus
- But what is NoStarNoPlusPlus?
- Well, the other two original rules: (M) | 6

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | (M) | 6
- StarExp ::= M * NoStarNoPlusPlus
- NoStarNoPlusPlus ::= (M) | 6
- But we have (M) | 6 twice, and it's the same language each time. Let's have one

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | NoStarNoPlusPlus
- StarExp ::= M * NoStarNoPlusPlus
- NoStarNoPlusPlus ::= (M) | 6

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ 6 ++ * 6
- * Left assoc
- M :: = M++ | StarExp | NoStarNoPlusPlus
- StarExp ::= M * NoStarNoPlusPlus
- NoStarNoPlusPlus ::= (M) | 6

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.

- Think of a,b,m as operators
  e.g., a is "++", b is "!" , m is "*"

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 higher precedence than ***m***, which in turn has higher precedence than ***a***, and such that ***m*** associates to the right.

Disambiguating a Grammar – Take 2

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

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

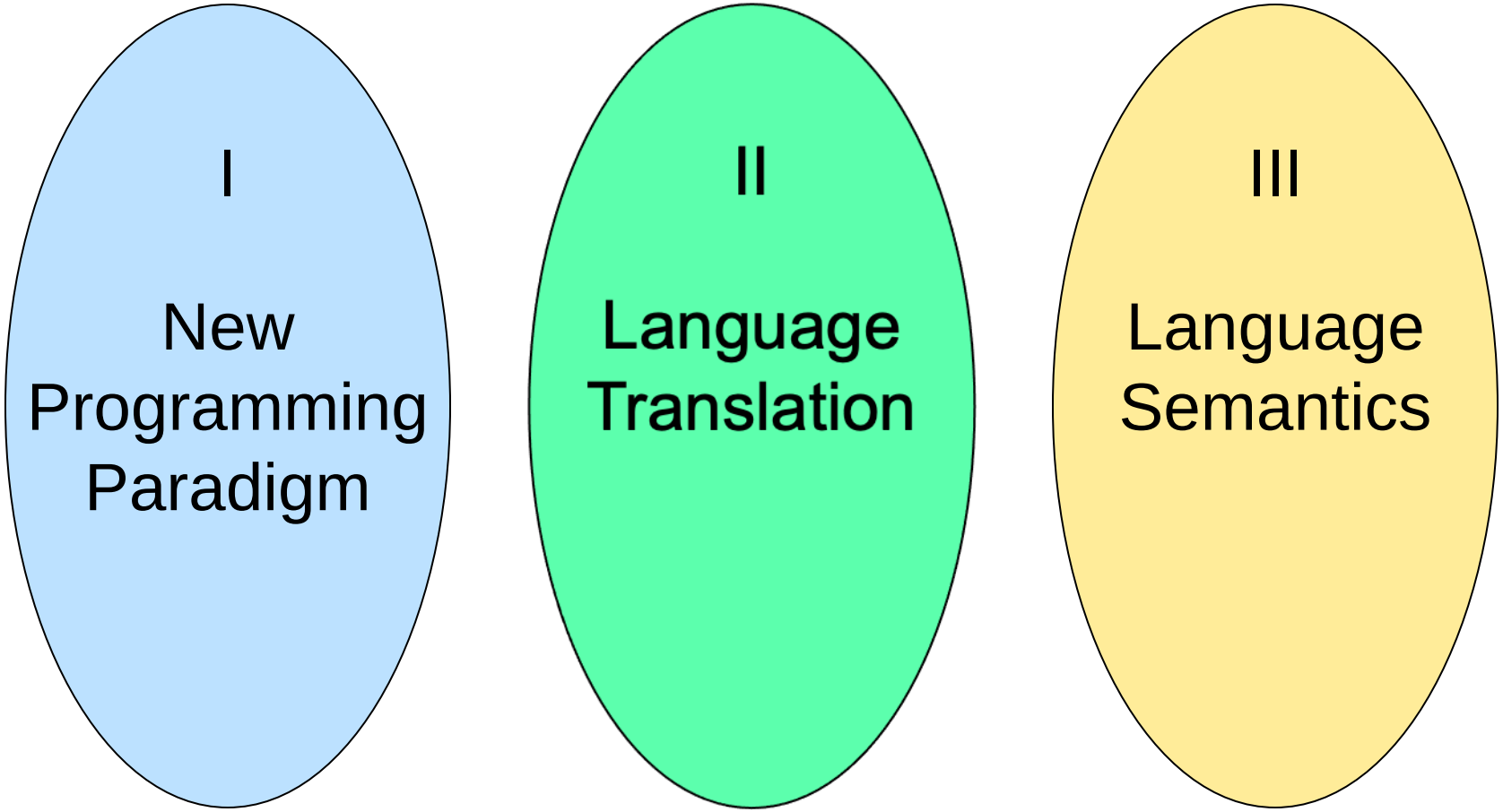
- seq = ε | may_word | word seq
- may_word = ε | “word”

- How do you parse “word”? And ε?

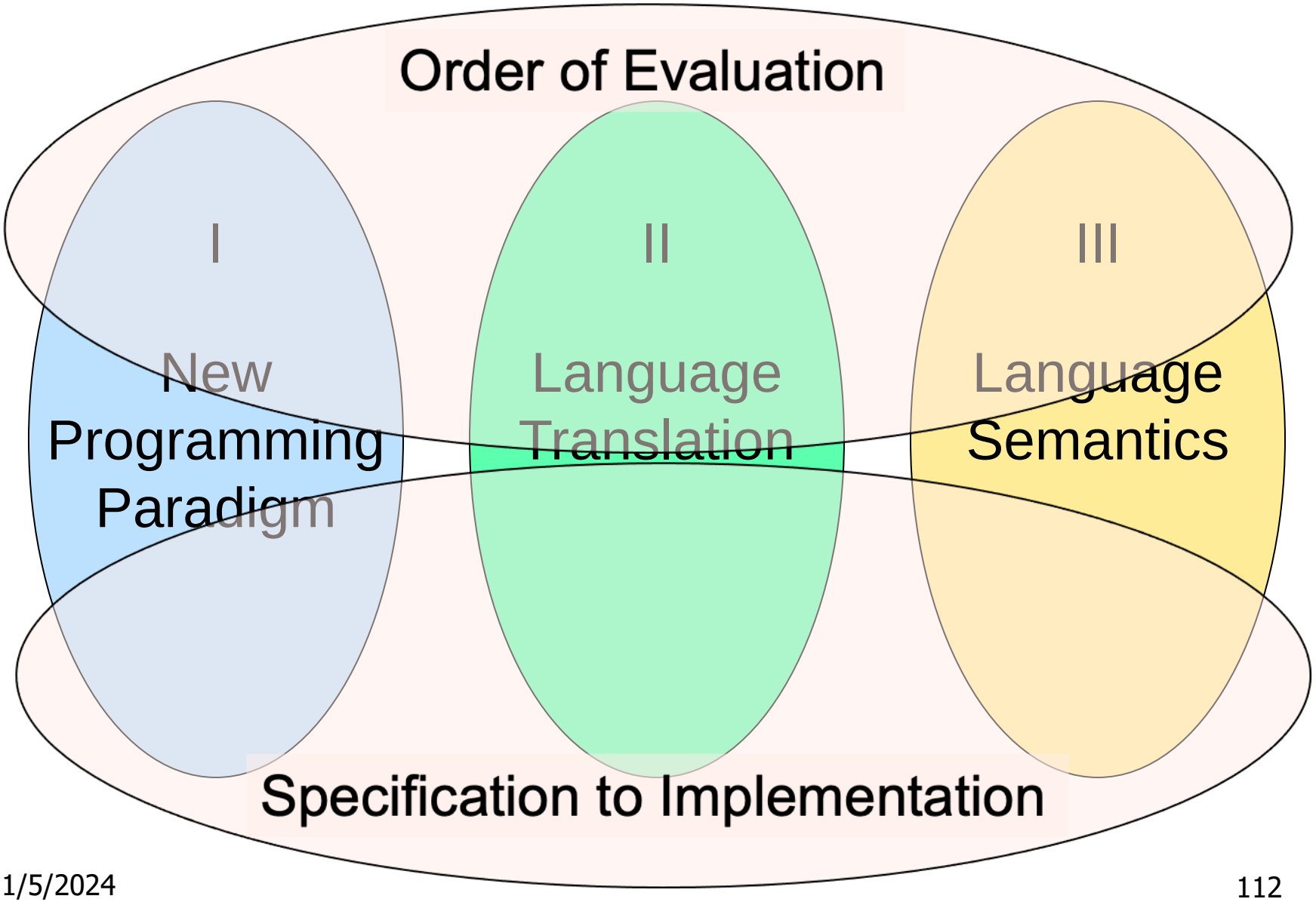
- How do you fix it?

Programming Languages & Compilers

Three Main Topics of the Course

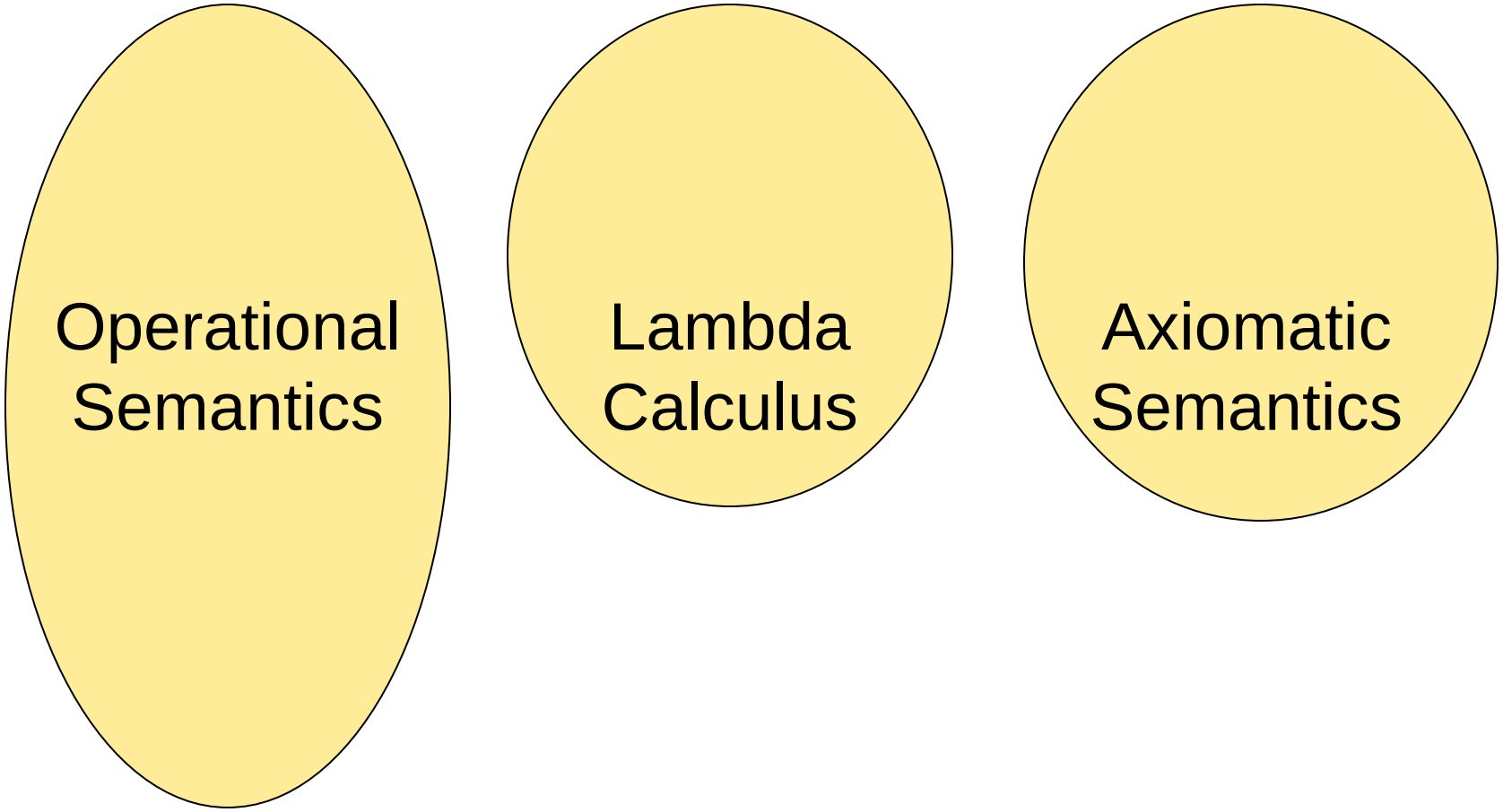


Programming Languages & Compilers

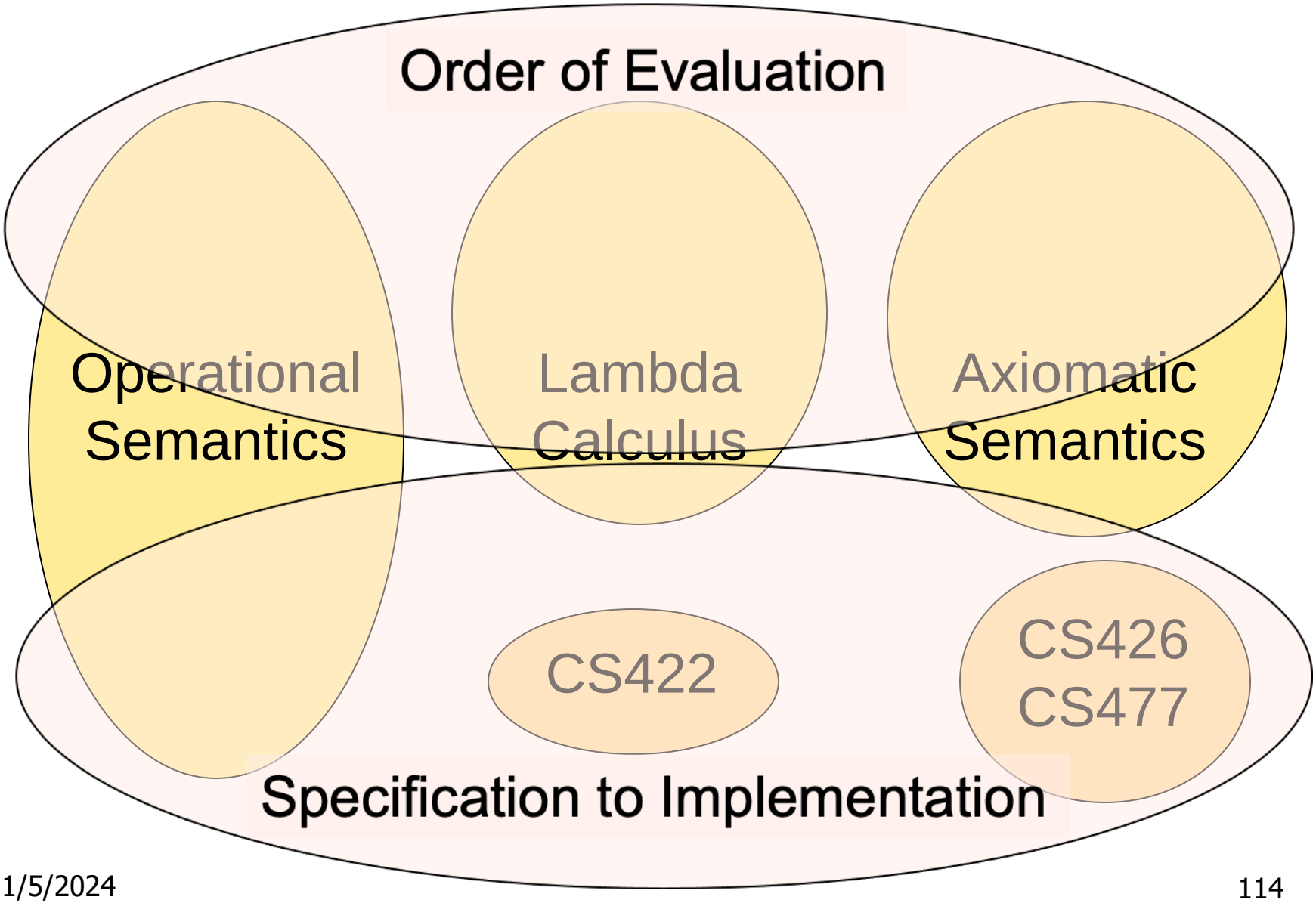


Programming Languages & Compilers

III : Language Semantics



Programming Languages & Compilers



Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

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

Dynamic Semantics

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