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**)
- Tuesday we finished **lexing** those strings into tokens, and started the rest of **parsing**
- Today we will continue **parsing**

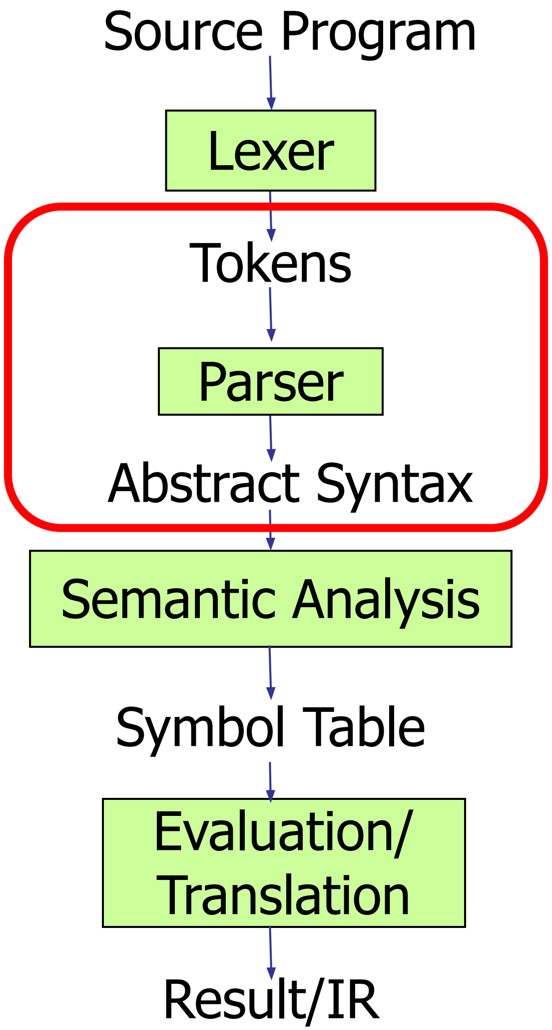
Objectives for Today

- **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**)
- Tuesday we finished **lexing** those strings into tokens, and started the rest of **parsing**
- Today we will continue **parsing**

Questions from Tuesday?

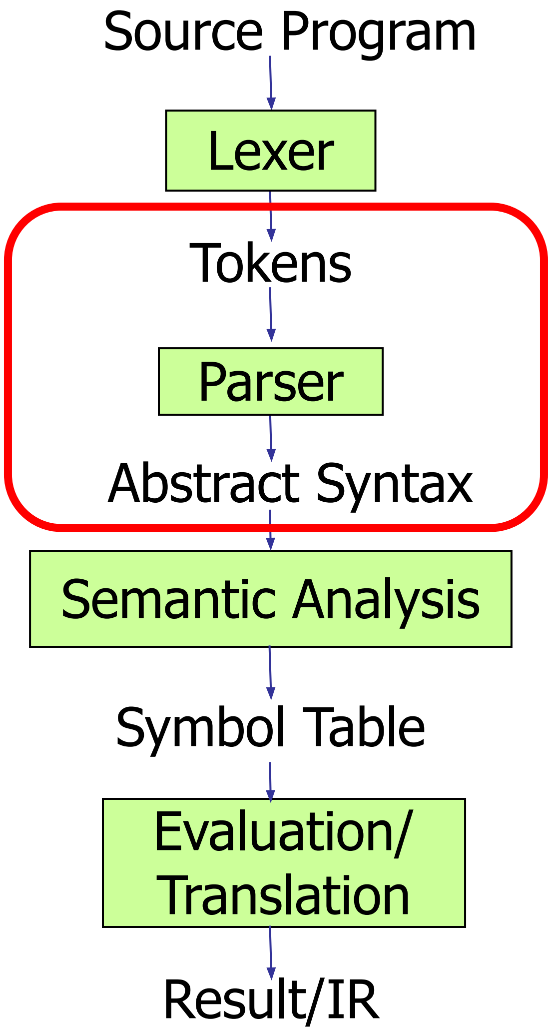
Parsing, Continued

Lexing and **Parsing**



To **parse** our source program and get **abstract syntax**, we need a **grammar** defined in terms of the kinds of **tokens** we get out of our lexer.

Lexing and **Parsing**



To **parse** our source program and get **abstract syntax**, we need a **grammar** defined in terms of the kinds of **tokens** we get out of our lexer.

The output, an **abstract syntax tree**, will track not just categories, but also **structure**.

Parse Trees

- **Abstract syntax tree** with **more detail**
- **Graphical representation** of derivation
- Each **node** labeled with either nonterminal or terminal
  - If node is labeled with a **terminal**, then it is a **leaf** (no sub-trees)
  - If node is labeled with a **nonterminal**, then it has **one branch for each element** in the **right-hand side** of rule used to substitute for it

Parse Trees

- **Abstract syntax tree** with **more detail**
- **Graphical representation** of derivation
- Each **node** labeled with either nonterminal or terminal
  - If node is labeled with a **terminal**, then it is a **leaf** (no sub-trees)
  - If node is labeled with a **nonterminal**, then it has **one branch for each element** 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

Consider grammar:

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

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

We could derive this **more than one way**, but for now we fix one

Example

Consider grammar:

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

1 * 1 + 0 : <exp>

Example

Consider grammar:

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

1 * 1 + 0 : <exp>

<exp> is the start symbol

Example

Consider grammar:

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

1 * 1 + 0 : <exp>

<factor>

use <exp> ::= <factor>

Example

Consider grammar:

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

1 * 1 + 0 : <exp>

<factor>

now derive <factor>

Example

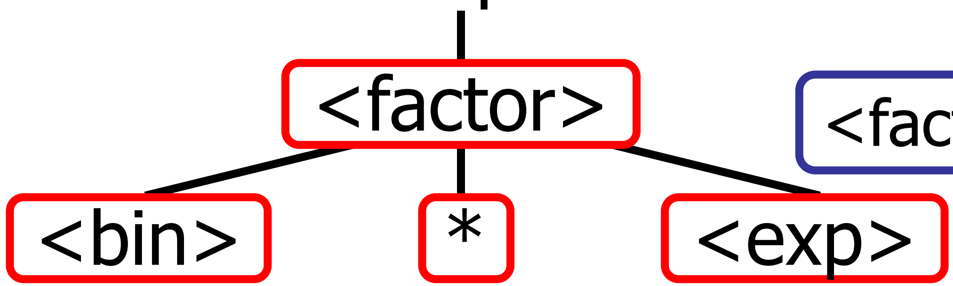
Consider grammar:

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

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

<bin> ::= 0 | 1

1 * 1 + 0 : <exp>



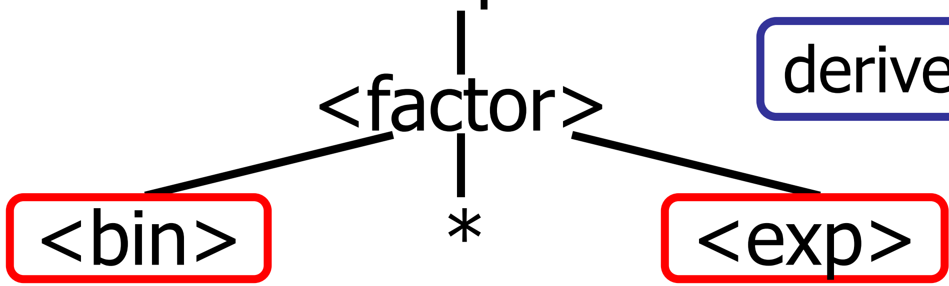
<factor> ::= <bin> * <exp>

Example

Consider grammar:

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

1 * 1 + 0 : <exp>



derive <bin> and <exp>

Example

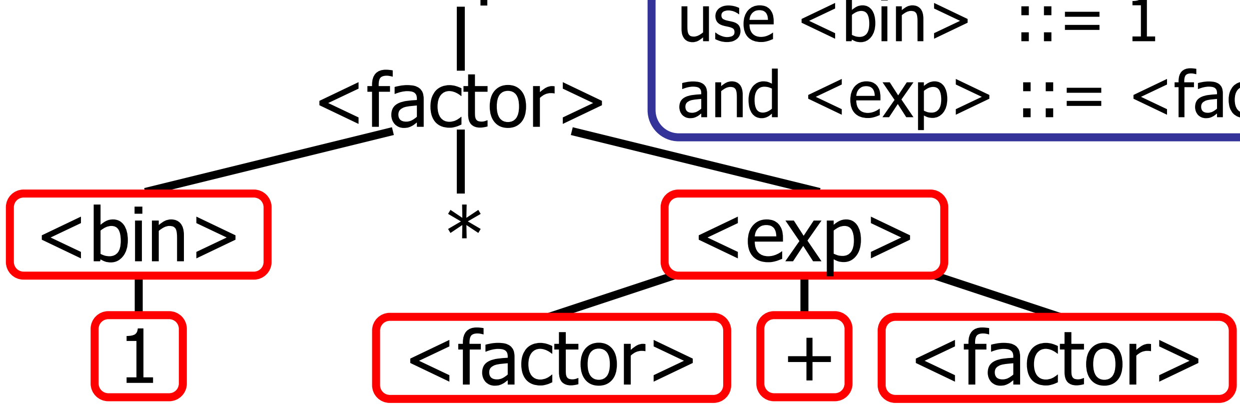
Consider grammar:

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

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

<bin> ::= 0 | 1

1 * 1 + 0 : <exp>



use <bin> ::= 1
and <exp> ::= <factor> + <factor>

Example

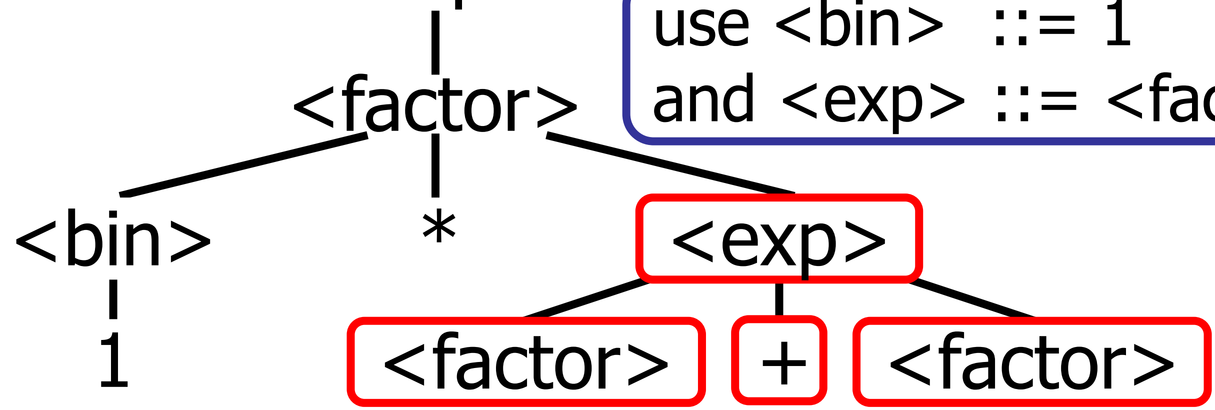
Consider grammar:

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

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

<bin> ::= 0 | 1

1 * 1 + 0 : <exp>



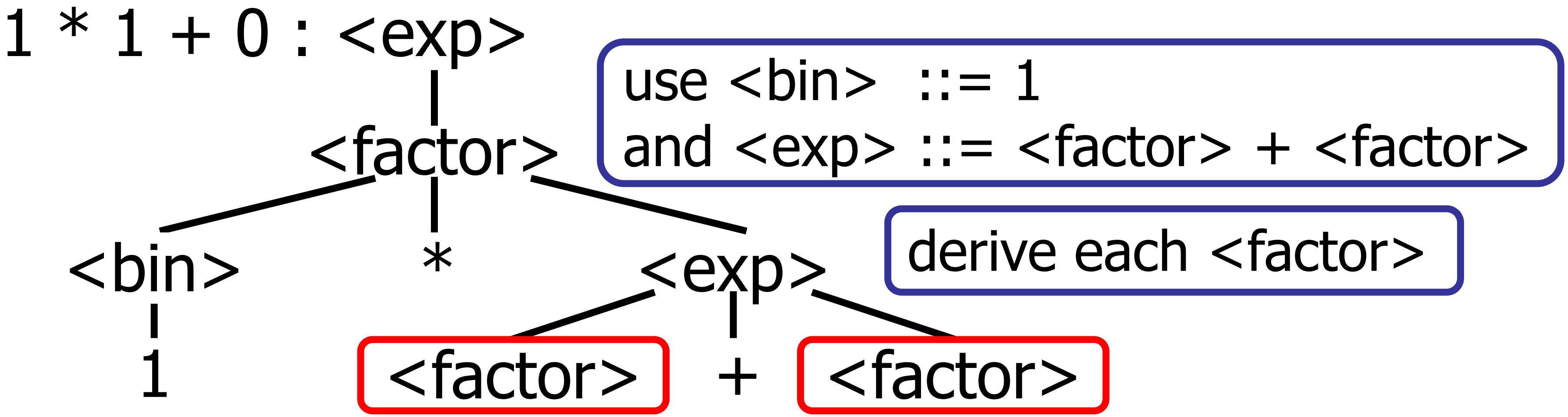
use <bin> ::= 1
and <exp> ::= <factor> + <factor>

1 is terminal

Example

Consider grammar:

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

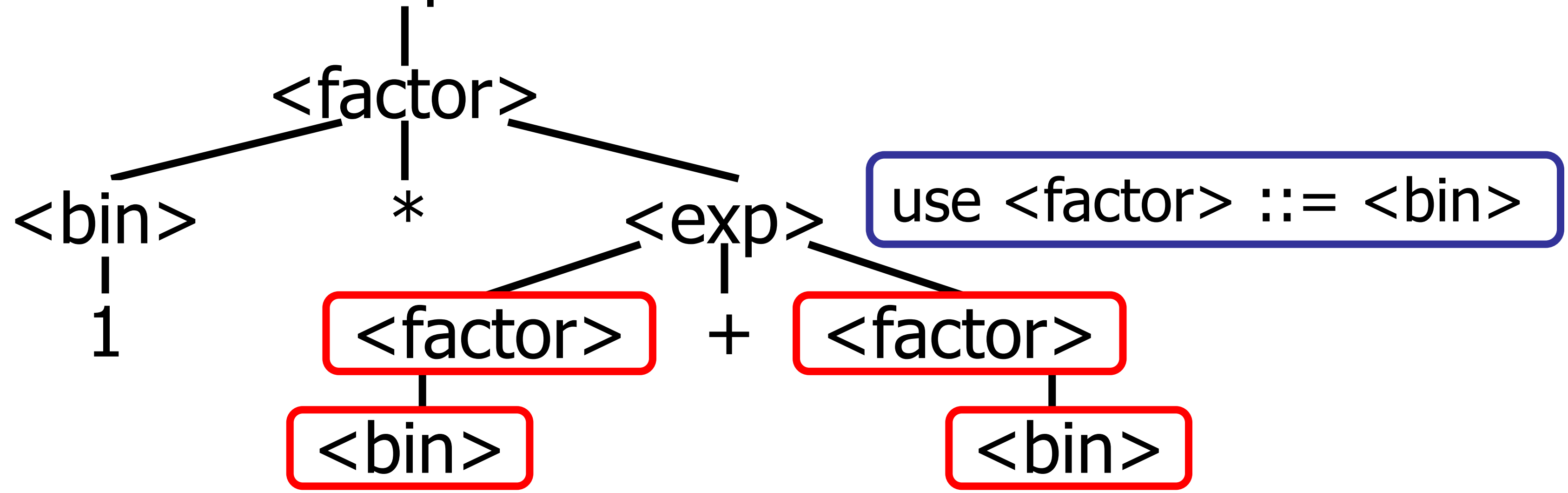


Example

Consider grammar:

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

1 * 1 + 0 : <exp>

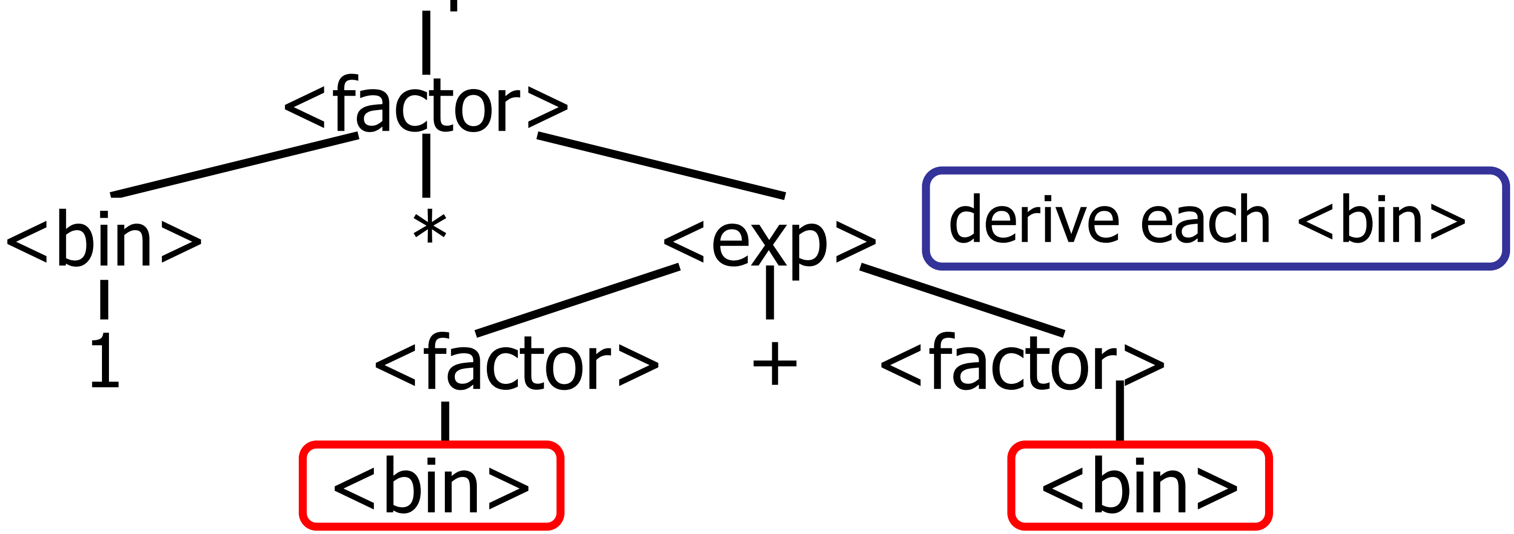


Example

Consider grammar:

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

1 * 1 + 0 : <exp>

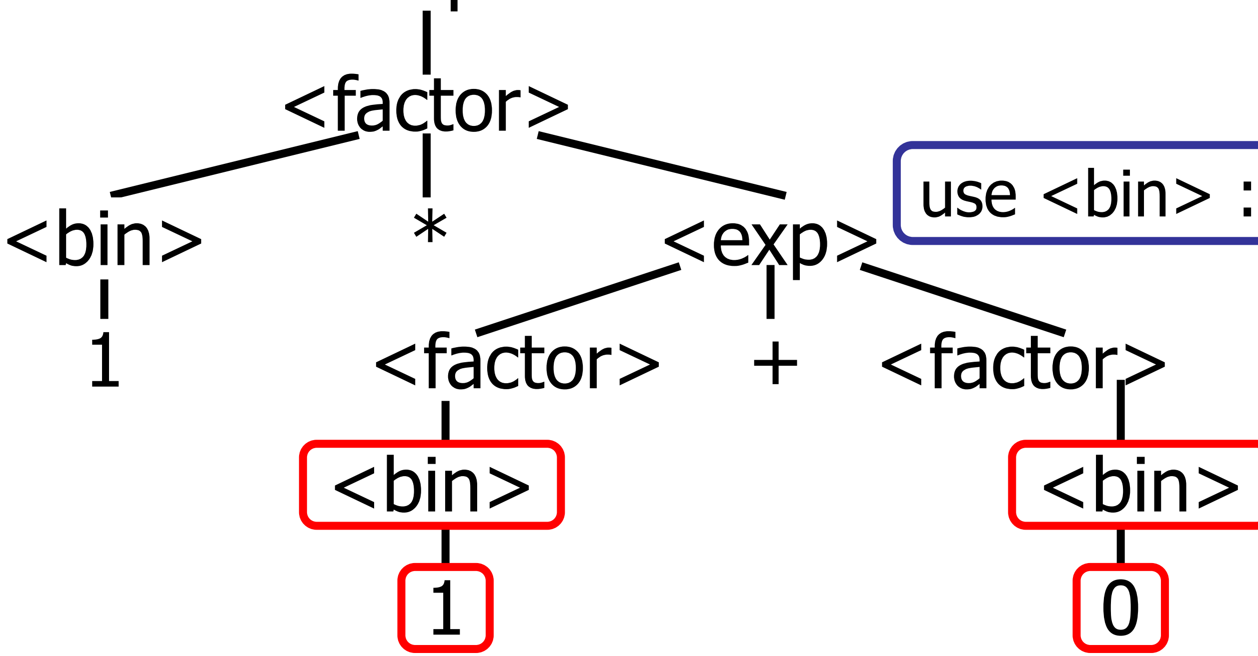


Example

Consider grammar:

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

1 * 1 + 0 : <exp>



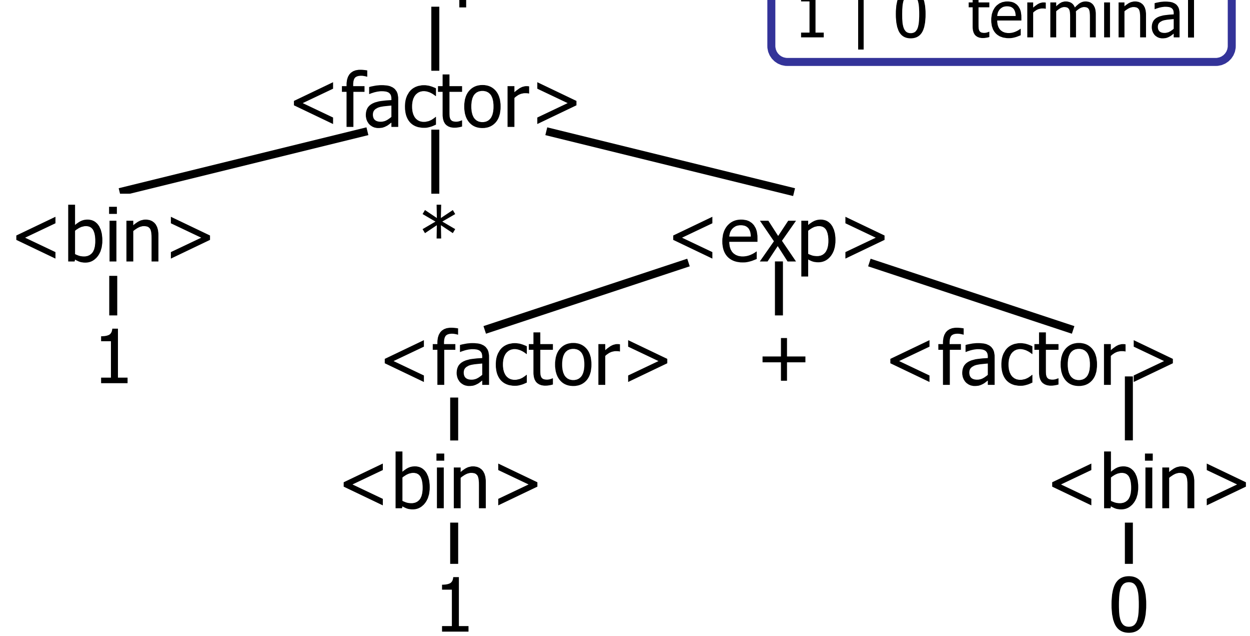
use <bin> ::= 1 | 0

Example

Consider grammar:

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

1 * 1 + 0 : <exp>



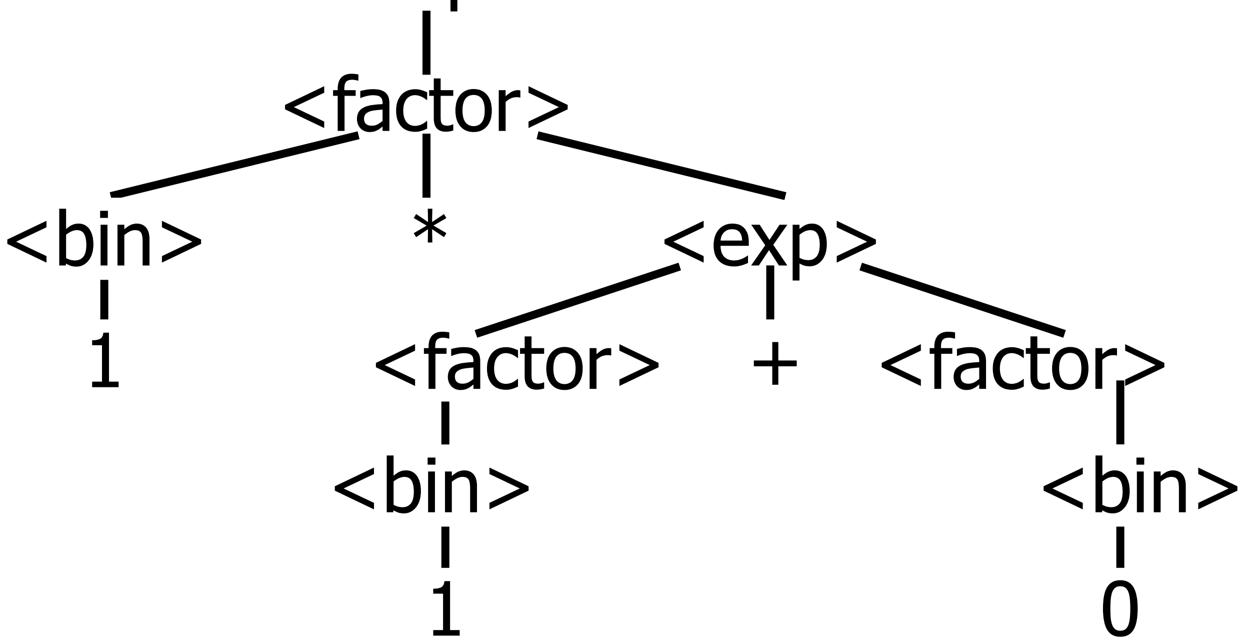
1 | 0 terminal

Example

Consider grammar:

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

1 * 1 + 0 : <exp>

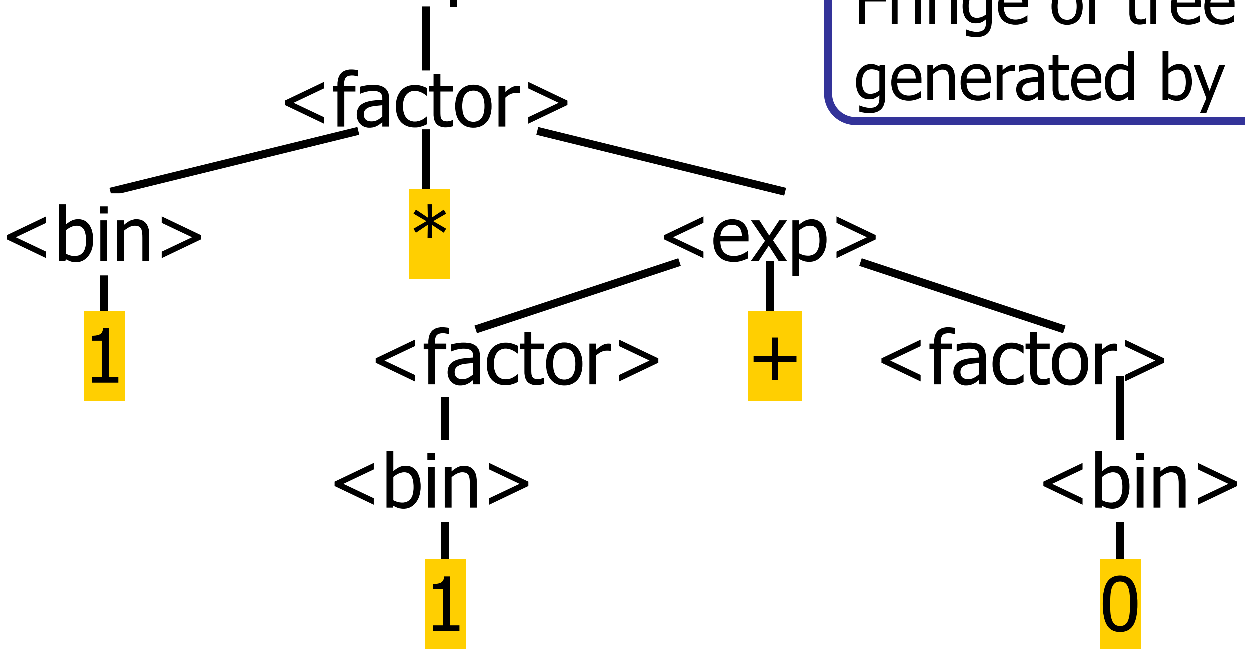


Example

Consider grammar:

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

1 * 1 + 0 : <exp>



Fringe of tree is string generated by grammar

Example

Consider grammar:

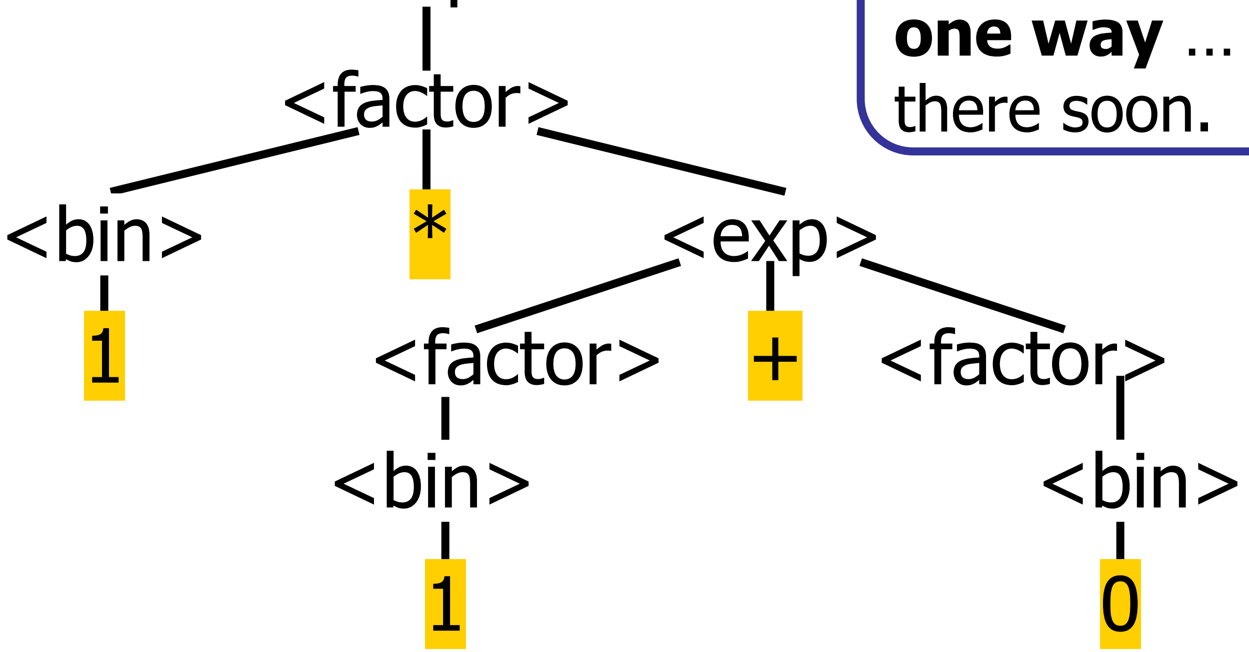
<exp> ::= <factor> | <factor> + <factor>

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

<bin> ::= 0 | 1

Note that we could derive this **more than one way** … we'll get there soon.

1 * 1 + 0 : <exp>



Questions so far?

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

- 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

Recall grammar:

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

Example

```
type exp    = Factor of factor | Plus of factor * factor
and factor = Bin of bin | Mult of bin * exp
and bin     = Zero | One
```

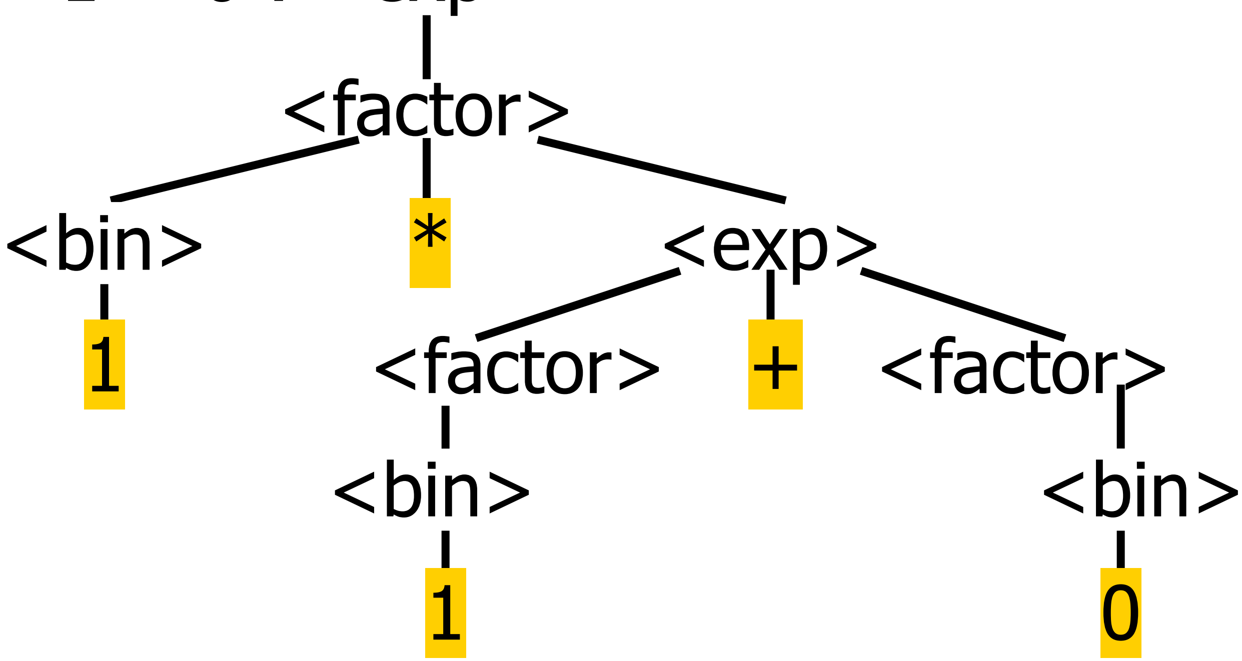
Recall grammar:

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

Example

```
type exp    = Factor of factor | Plus of factor * factor
and factor = Bin of bin | Mult of bin * exp
and bin      = Zero | One
```

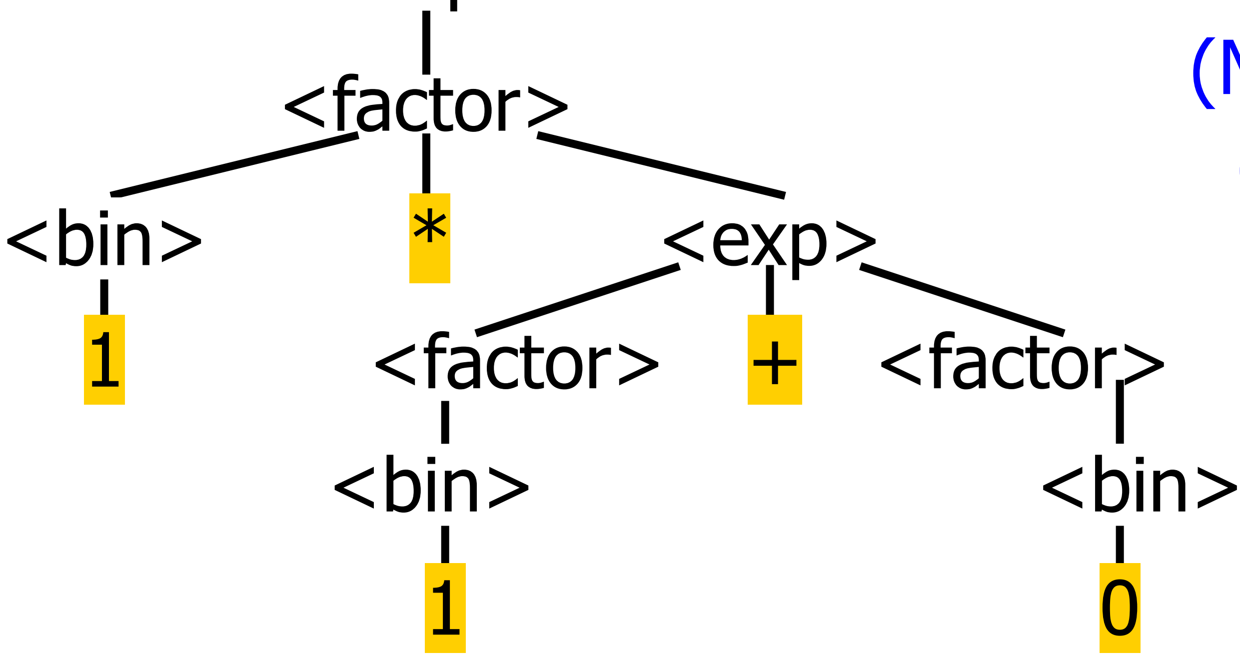
1 * 1 + 0 : <exp>



Example

```
type exp    = Factor of factor | Plus of factor * factor
and factor = Bin of bin | Mult of bin * exp
and bin       = Zero | One
```

1 * 1 + 0 : <exp> can be written as Factor (Mult (One, Plus (Bin One, Bin Zero)))



Questions so far?

Ambiguity

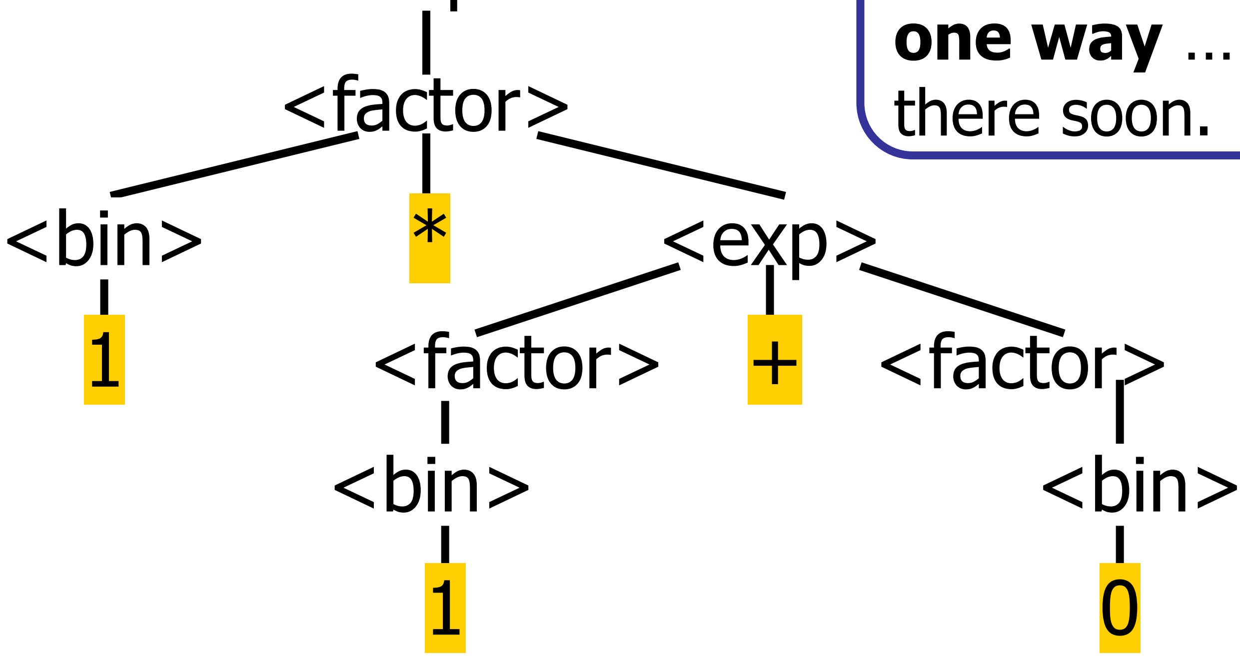
Example

Consider grammar:

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

Note that we could derive this **more than one way** … we'll get there soon.

1 * 1 + 0 : <exp>



Ambiguous Grammars and Languages

- A BNF grammar is **ambiguous** if its language **contains strings** for which there is **more than one parse tree**
  - Common sources of ambiguity:
    - Lack of determination of operator **precedence**
    - Lack of determination of operator **associativity**
  - Not the only sources of ambiguity
- If ***all*** **BNFs** for a language are ambiguous, then the language is **inherently ambiguous**
- Otherwise, we will try to **disambiguate**

Ambiguity

Ambiguous Grammars and Languages

- A BNF grammar is **ambiguous** if its language **contains strings** for which there is **more than one parse tree**
  - Common sources of ambiguity:
    - Lack of determination of operator **precedence**
    - Lack of determination of operator **associativity**
  - Not the only sources of ambiguity
- If ***all* BNFs** for a language are ambiguous, then the language is **inherently ambiguous**
- Otherwise, we will try to **disambiguate**

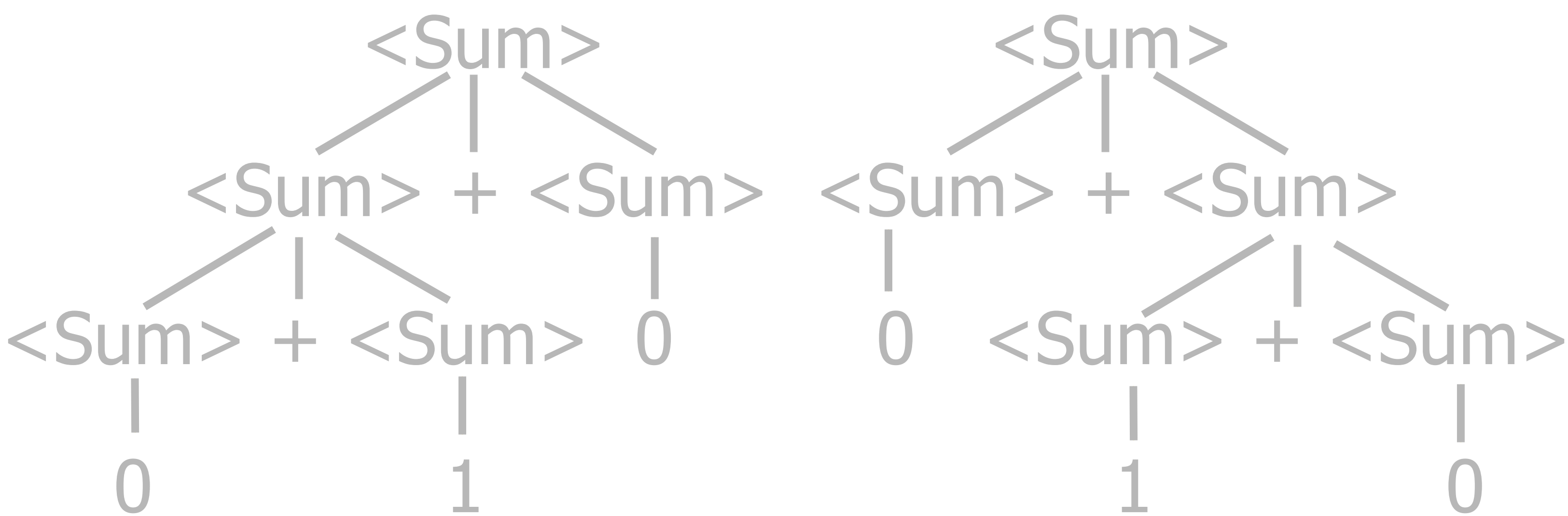
Ambiguous Grammars and Languages

- A BNF grammar is **ambiguous** if its language **contains strings** for which there is **more than one parse tree**
  - Common sources of ambiguity:
    - Lack of determination of operator **precedence**
    - Lack of determination of operator **associativity**
  - Not the only sources of ambiguity
- If ***all*** **BNFs** for a language are ambiguous, then the language is **inherently ambiguous**
- Otherwise, we will try to **disambiguate**

Example: Ambiguous Grammar

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

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

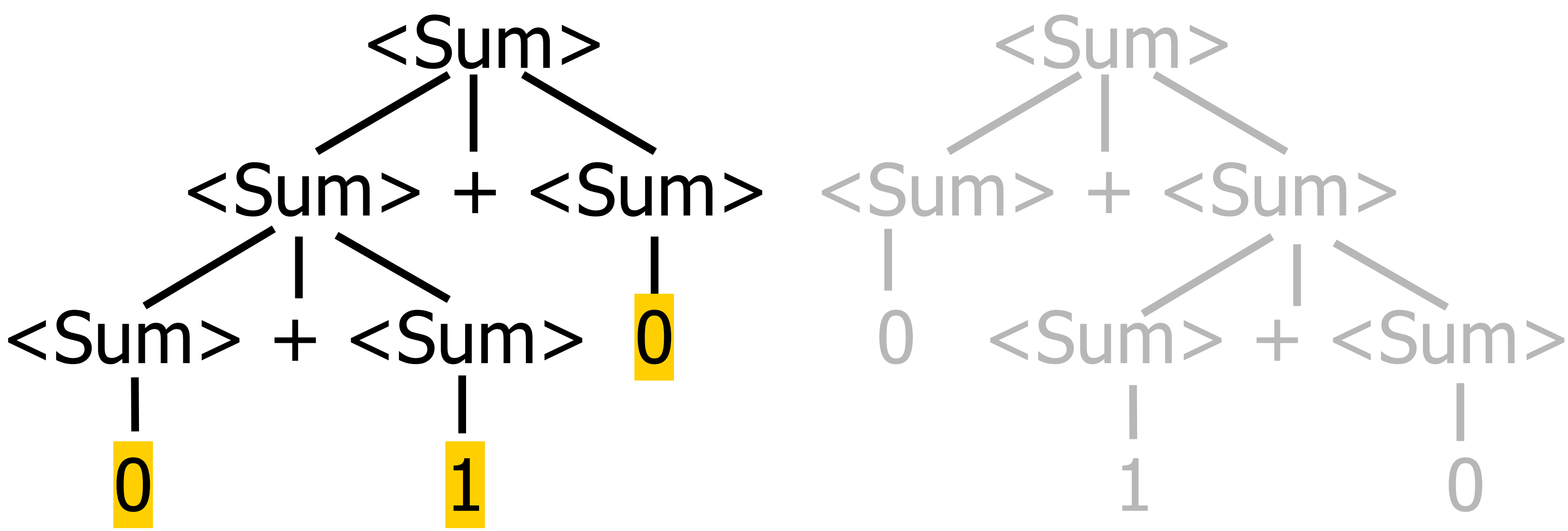


Ambiguity

Example: Ambiguous Grammar

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

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

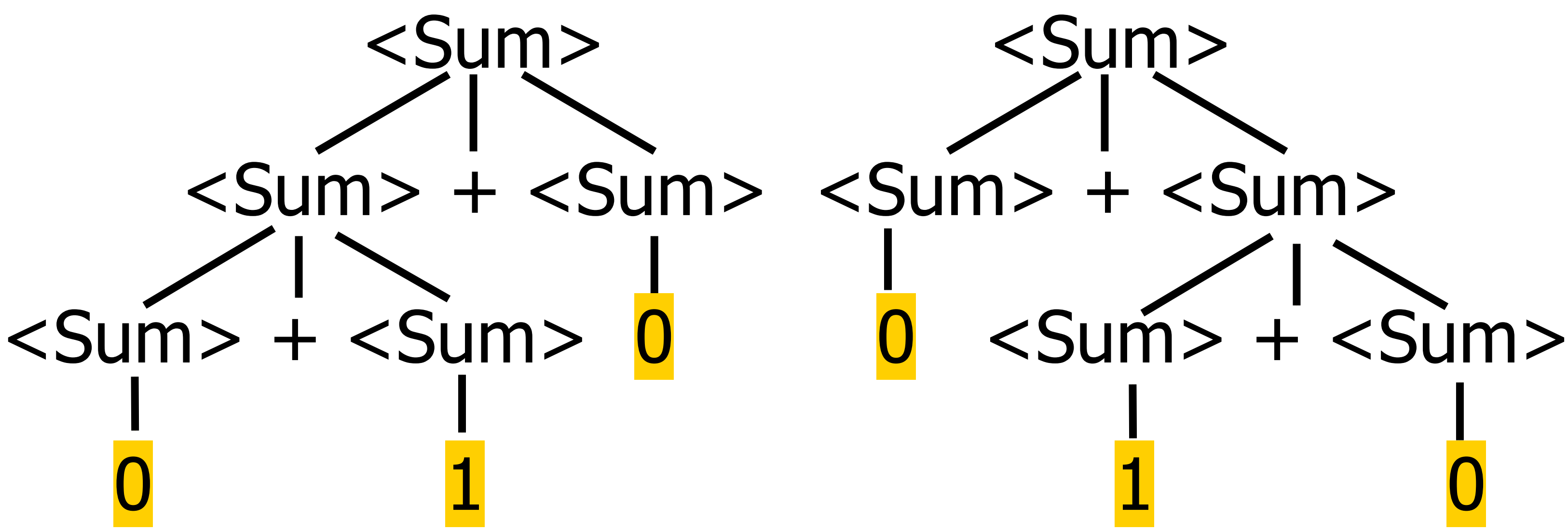


Ambiguity

Example: Ambiguous Grammar

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

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

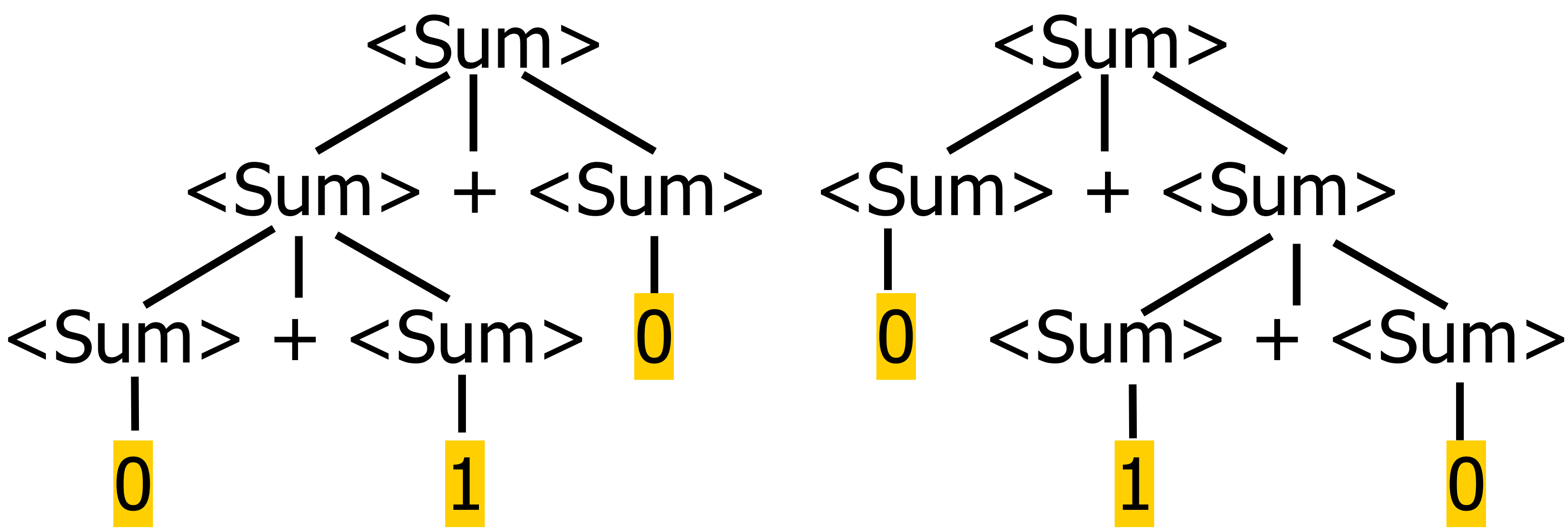


Questions so far?

Example: Ambiguous Grammar

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

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



How do we disambiguate?

Disambiguating a Grammar

- Given an ambiguous grammar G with start symbol S, find a grammar G′ with the same start symbol S, such that language of G = language of G′
- **Not always possible**
- **No algorithm in general**
- But often in programming languages, when faced with an undecidable problem, we can either
  - solve some useful **decidable subproblems**, or
  - **approximately solve** the whole problem.

How do we disambiguate?

Disambiguating a Grammar

- Given an ambiguous grammar G with start symbol S, find a grammar G′ with the same start symbol S, such that language of G = language of G′
- **Not always possible**
- **No algorithm in general**
- But often in programming languages, when faced with an undecidable problem, we can either
  - solve some useful **decidable subproblems**, or
  - **approximately solve** the whole problem.

How do we disambiguate?

Disambiguating a Grammar

- **Idea:** Each nonterminal represents **all strings having some property**
- **Identify these properties** (often in terms of things that cannot happen)
- **Use these properties** to **inductively guarantee** every string in language has a unique parse
- We'll handle this in more detail later, but let's start with some examples

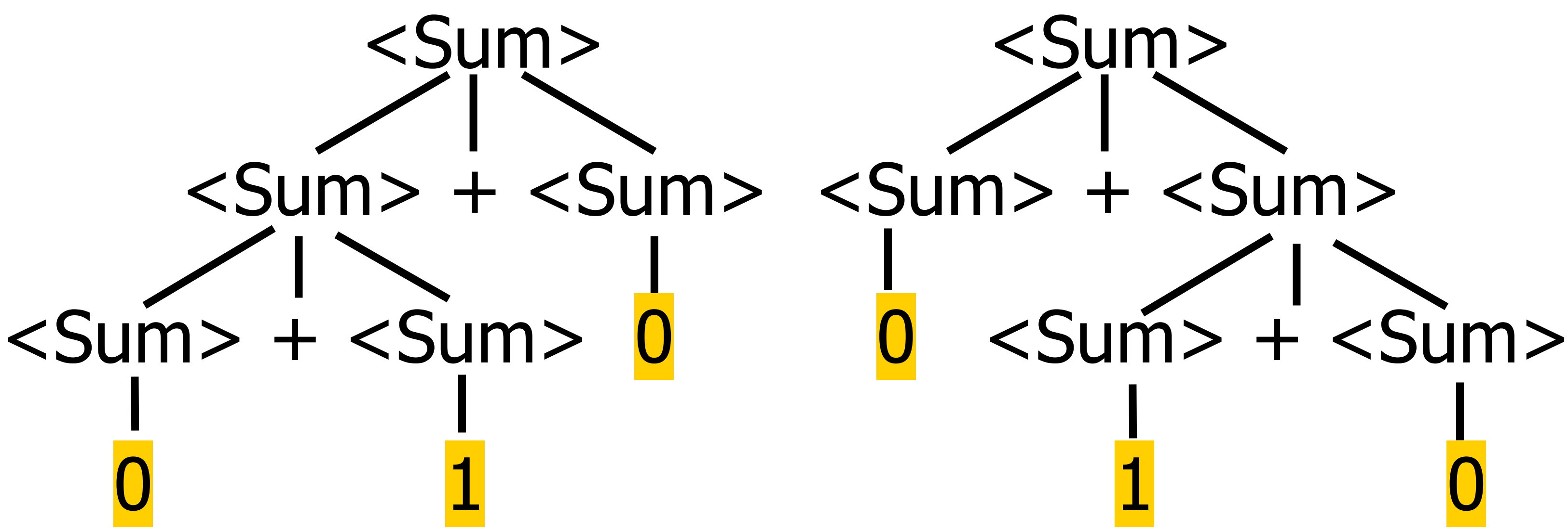
Disambiguating a Grammar

- **Idea:** Each nonterminal represents **all strings having some property**
- **Identify these properties** (often in terms of things that cannot happen)
- **Use these properties** to **inductively guarantee** every string in language has a unique parse
- We'll handle this in more detail later, but let's start with some examples

Example: Ambiguous Grammar

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

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



Source of ambiguity: **associativity**

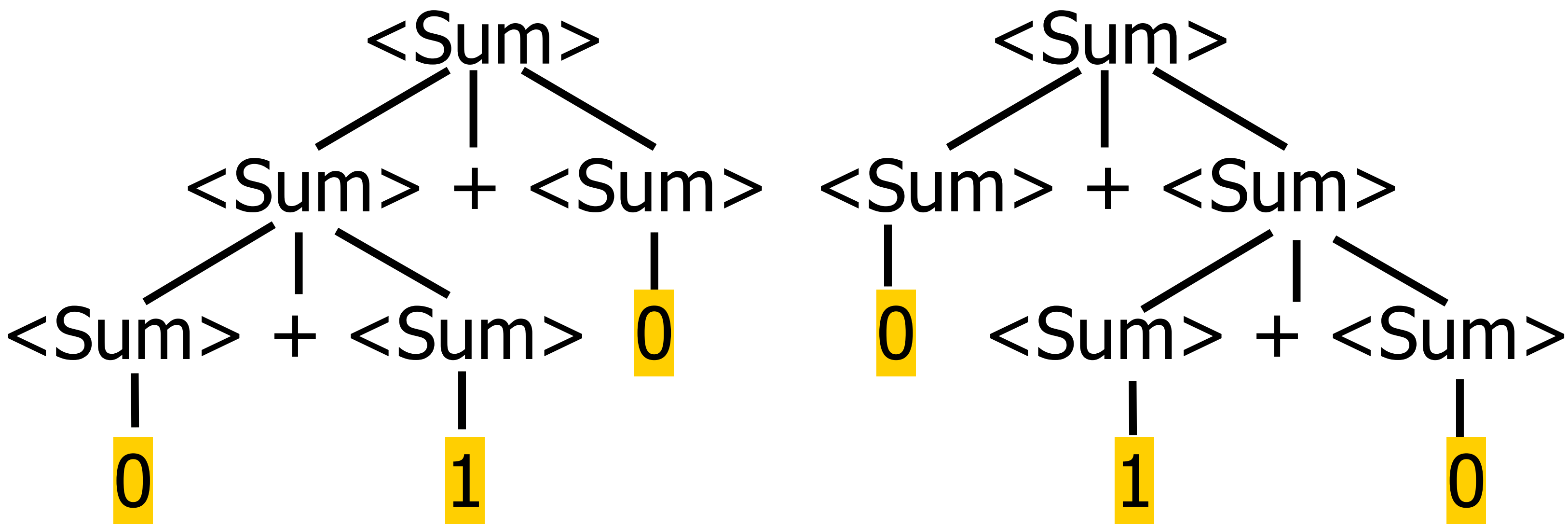
How to Enforce Associativity

- Have at most **one recursive call** per production
- When **two or more recursive calls** would be natural, to refactor:
  - Leave **rightmost call** for **right associativity**
  - Leave **leftmost call** for **left associativity**

Example: Disambiguation

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

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

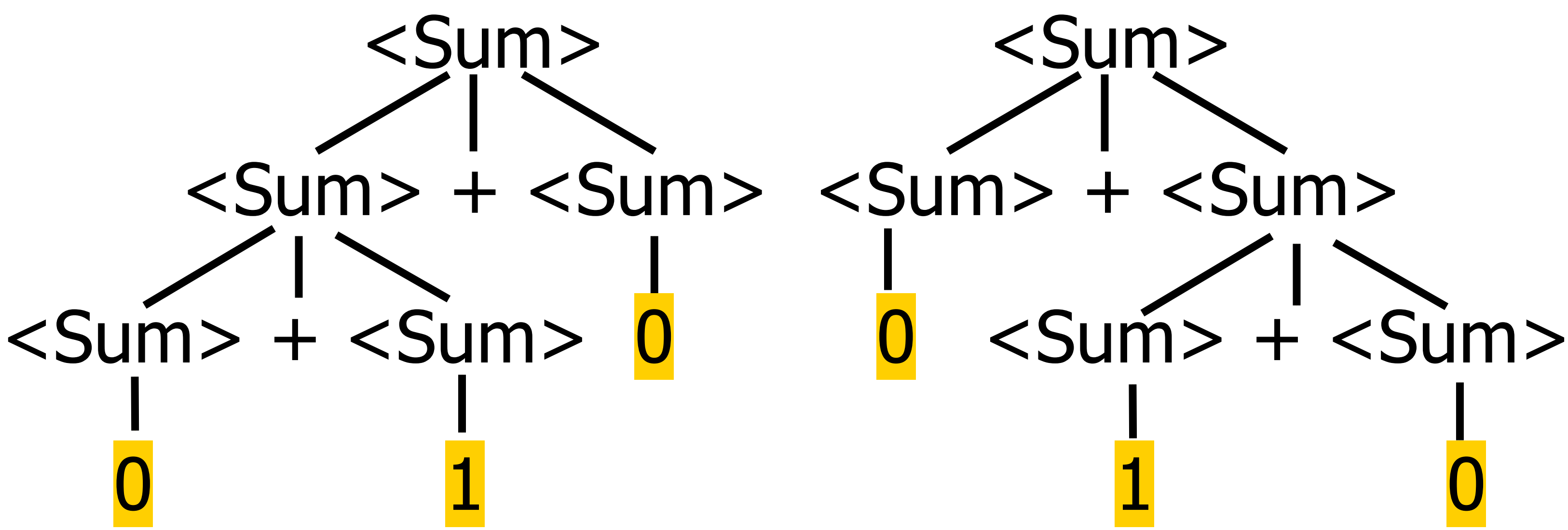


Source of ambiguity: **associativity**

Example: Disambiguation

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

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



Source of ambiguity: **associativity**

Example: Disambiguation

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

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

Source of ambiguity: **associativity**

Example: Disambiguation

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

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

Source of ambiguity: **associativity**

Example: Disambiguation

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

<Sum> ::= <Num> | **<Num> + <Sum>**

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

Source of ambiguity: **associativity**

Example: Disambiguation

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

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

Source of ambiguity: **associativity**

Example: Disambiguation

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

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

Source of ambiguity: **associativity**

Example: Disambiguation

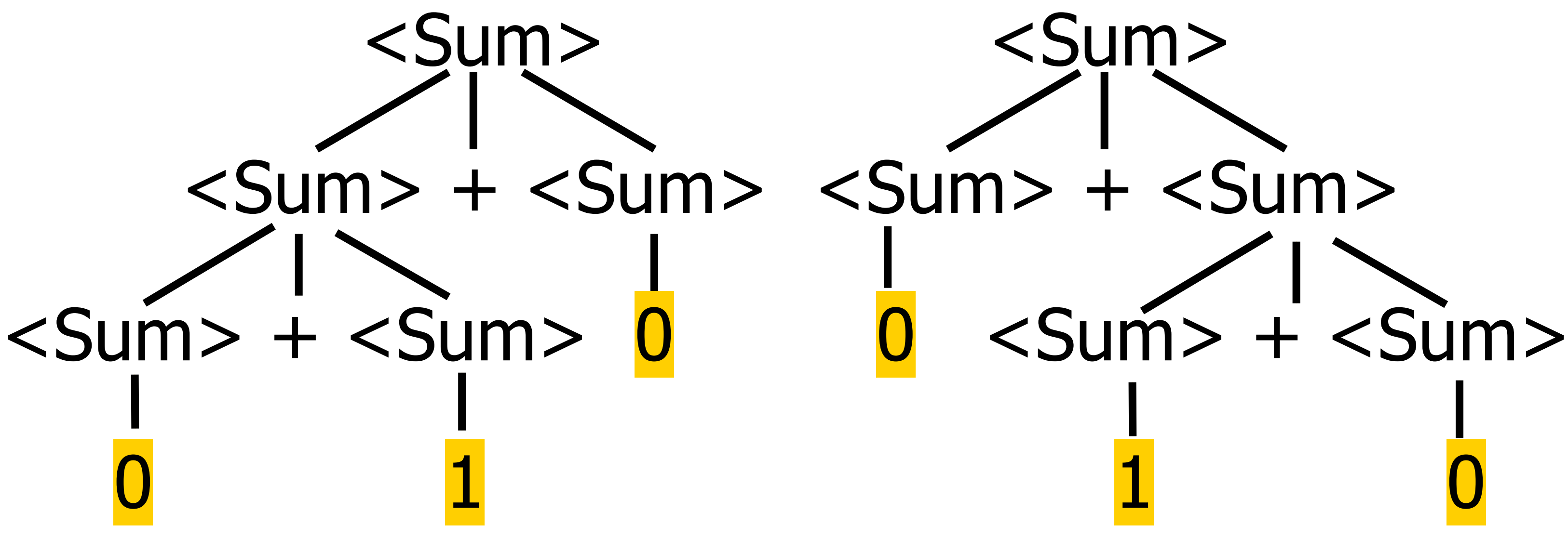
<Sum> ::= 0 | 1 | **<Sum> + <Sum>** | (<Sum>)

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

Example: Disambiguation

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

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

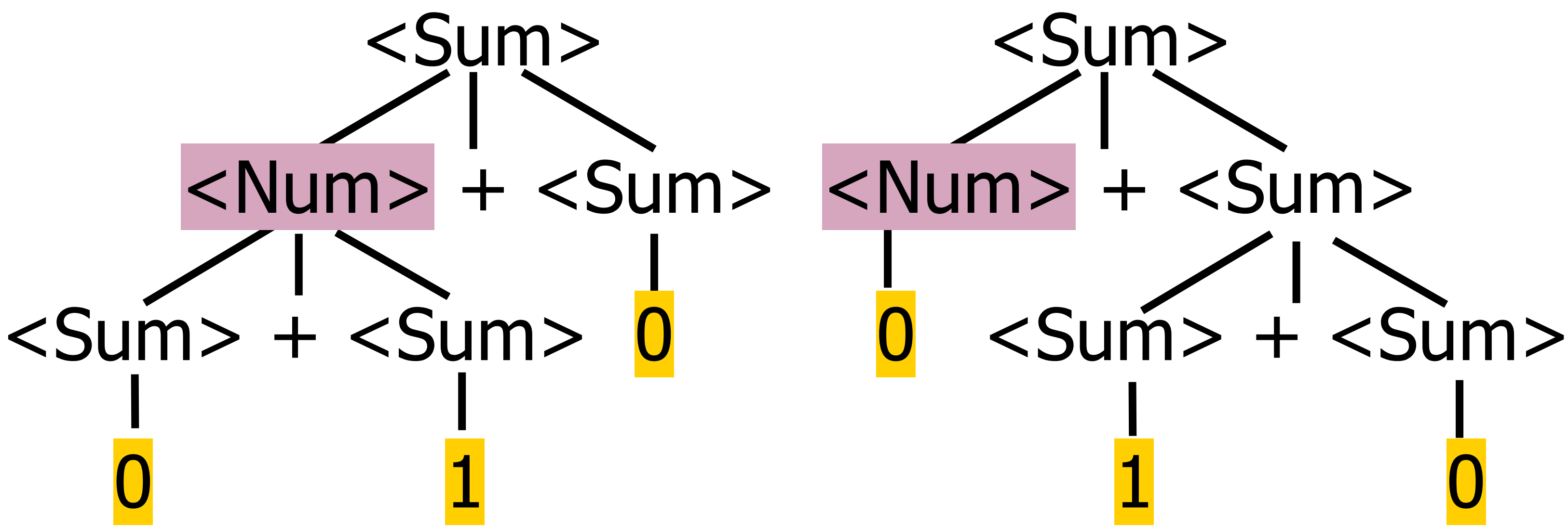


Ambiguity

Example: Disambiguation

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

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

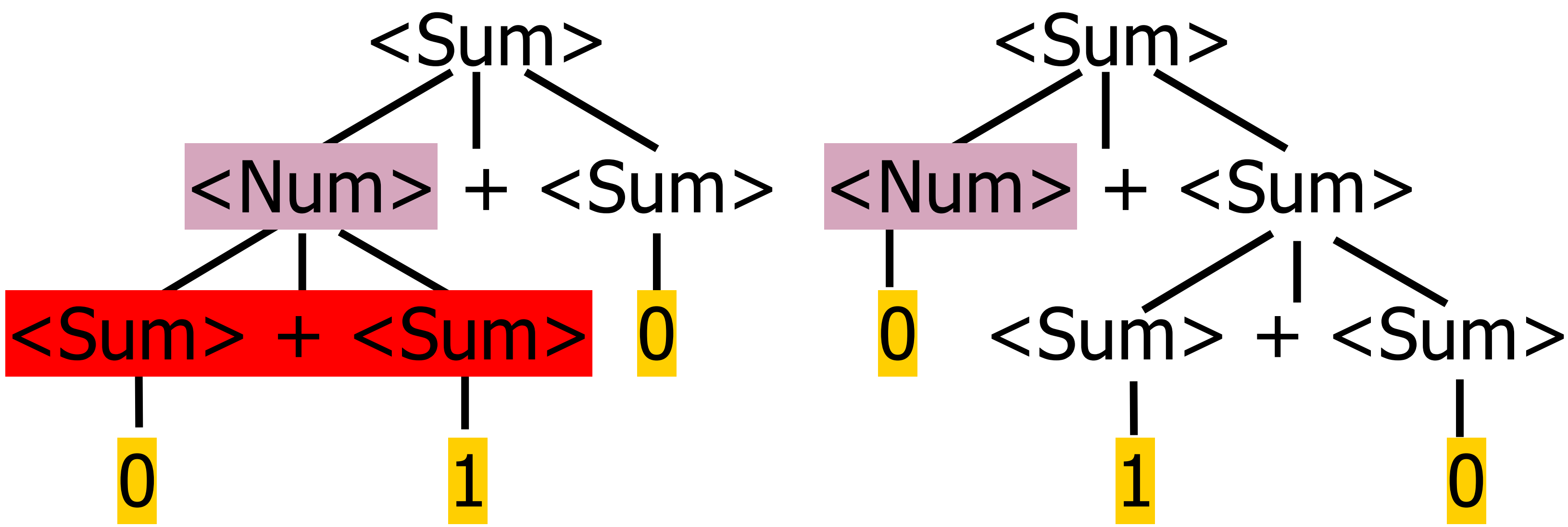


Ambiguity

Example: Disambiguation

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

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

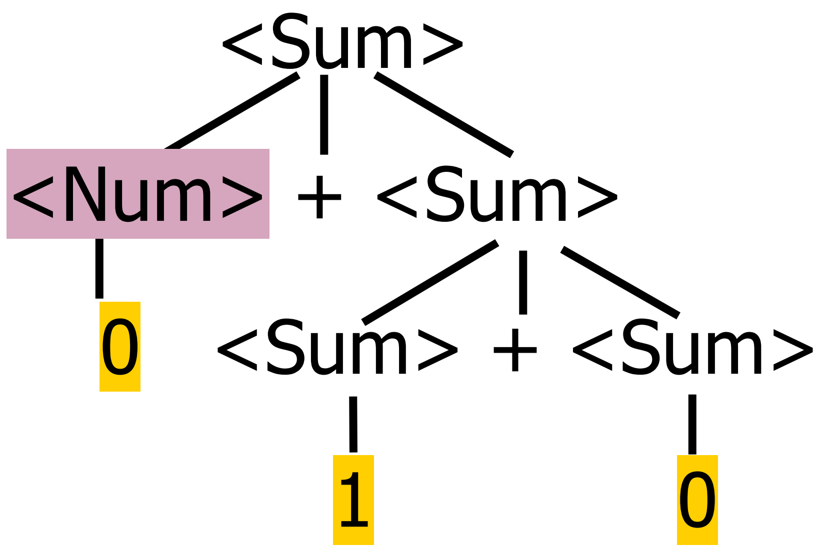


Ambiguity

Example: Disambiguation

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

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

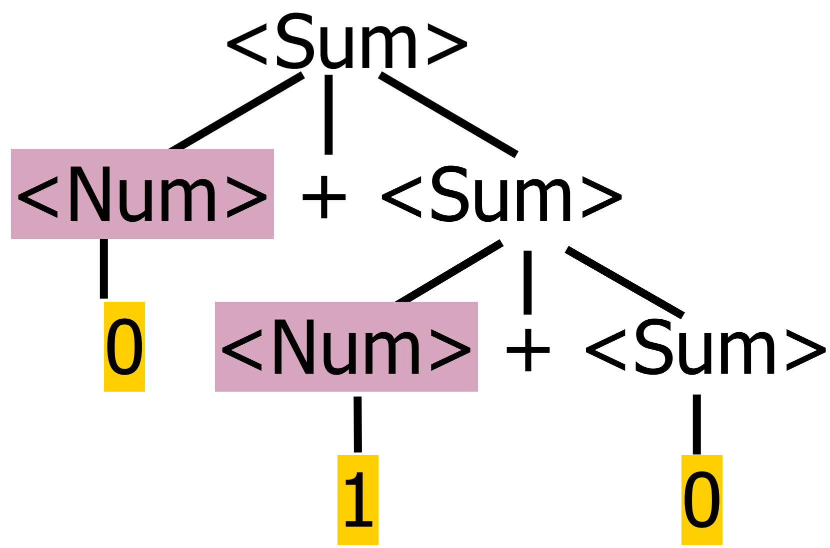


Ambiguity

Example: Disambiguation

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

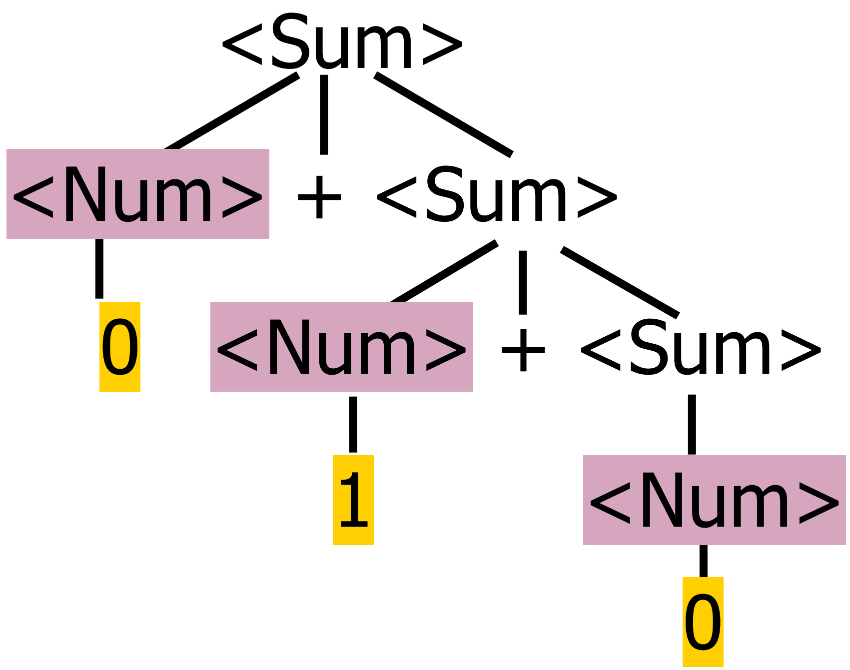
<Sum> ::= <Num> | **<Num> + <Sum>**
<Num> ::= 0 | 1 | (<Sum>)



Example: Disambiguation

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

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



Ambiguity

Example: Disambiguation

Ambiguous grammar:

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

Strings with more then one parse:

0 + 1 + 0

1 * 1 + 1

Sources of ambiguity:

associativity and precedence

Example: Disambiguation

Ambiguous grammar:

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

Strings with more then one parse:

0 + 1 + 0

1 * 1 + 1

Sources of ambiguity:

associativity and precedence

Example: Disambiguation

Ambiguous grammar:

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

Strings with more then one parse:

0 + 1 + 0

1 * 1 + 1

Sources of ambiguity:

associativity and precedence

Operator Precedence

- Operators of **highest precedence** evaluated **first** (that is, they bind more tightly)
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar

Precedence Table - Dated Sample

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

Precedence 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

Precedence 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

Precedence 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

Questions so far?

Implementing Parsers

Parser Code

- <*grammar*>.mly defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point

Ocamlyacc Input

File format:

%{

<*header*>

%}

<*declarations*>

%%

<*rules*>

%%

<*trailer*>

Ocamlyacc <*header*>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <*footer*> similar. Possibly used to call parser

Ocamlyacc <declarations>

- %token *symbol* … *symbol*
  Declare given symbols as tokens
- %token <*type*> *symbol* … *symbol*
  Declare given symbols as token constructors, taking an argument of type <*type*>
- %start *symbol* … *symbol*
  Declare given symbols as entry points; functions of same names in <*grammar*>.ml

Ocamlyacc <*declarations*>

- %type <*type*> *symbol … symbol*

  Specify type of attributes for given symbols.
  Mandatory for start symbols
- %left *symbol … symbol*
- %right *symbol … symbol*
- %nonassoc *symbol … symbol*

  Associate precedence and associativity to given symbols. Same line,same precedence; earlier line, lower precedence (broadest scope)

Ocamlyacc <*rules*>

- *nonterminal* :
  symbol ... *symbol* { *semantic_action* }
  | ...
  | *symbol* ... *symbol* { *semantic_action* }
  ;
- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access semantic attributes (values) of symbols by position: $1 for first symbol, $2 to second …

Example - Base types

```
(* File: expr.ml *)
type expr =
   | Term_as_Expr of term
   | Plus_Expr of (term * expr)
   | Minus_Expr of (term * expr)
and term =
   | Factor_as_Term of factor
   | Mult_Term of (factor * term)
   | Div_Term of (factor * term)
and factor =
   | Id_as_Factor of string
   | Parenthesized_Expr_as_Factor of expr
```

Example - Lexer (exprlex.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")))
```

Questions?

Next Class: Underlying Algorithm

Next Class

- **MP8** due next **Tuesday**
- **WA8** due next **Thursday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help