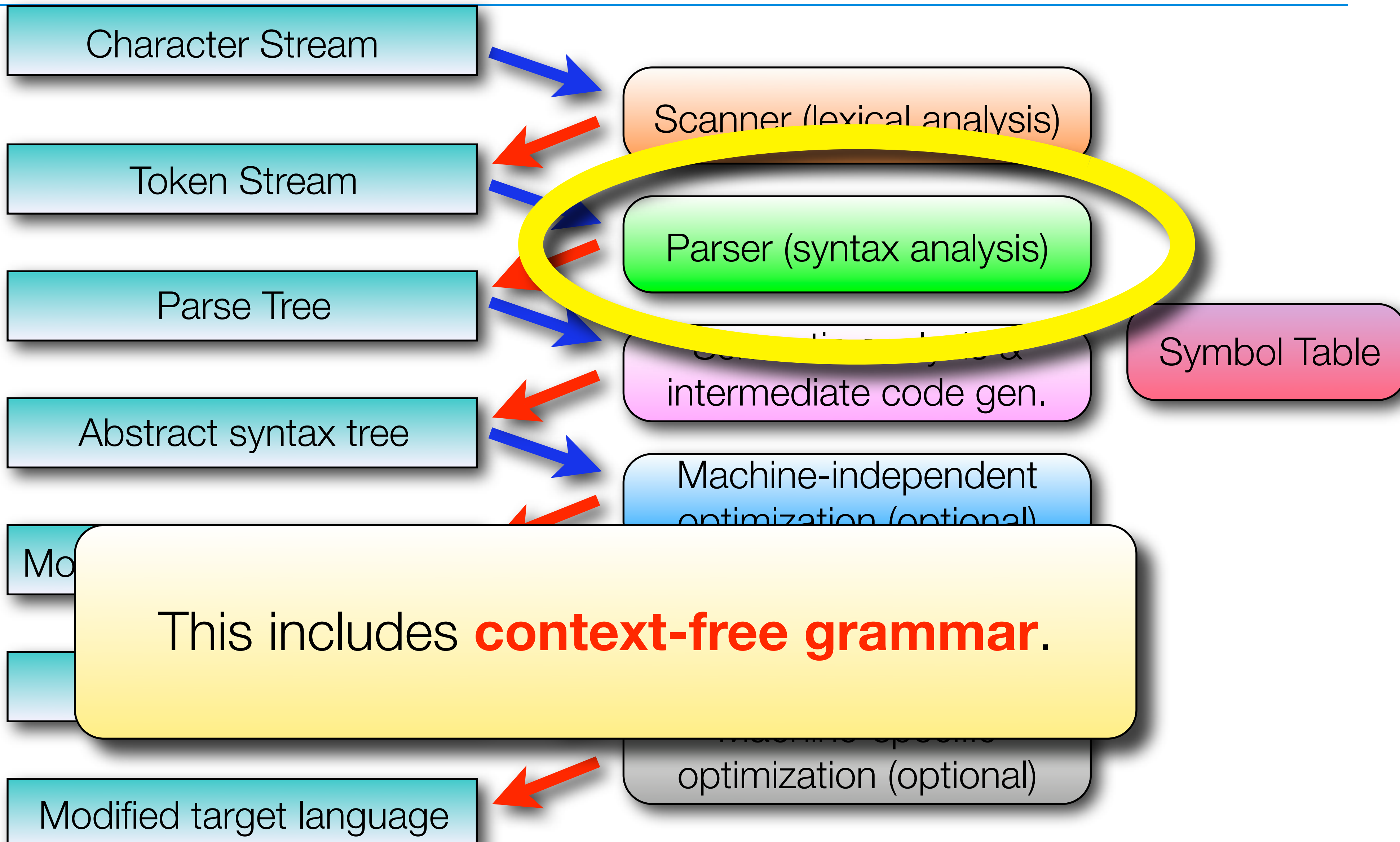
Lecture 4: Syntactic Analysis

COMP 524 Programming Language Concepts
Stephen Olivier
January 27, 2009

Based on notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

Goal of Lecture



Parsing

- The main task of parsing is to **identify the syntax**.

Review: Regular Expression Rules

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The empty string (i.e., “**ε**”)
  - Two REs next to each other (e.g., “*non_negative_digit digit*”) to denote concatenation.
  - Two REs separated by “|” next to each other (e.g., “*non_negative_digit* | *digit*”) to denote one RE or the other.
  - An RE followed by “*” (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

Review: Regular Expression Rules

- A RE consist of:
  - A character (e.g., "0", "1", ...)
  - The
  - Two *igit*") to deno

A RE is **NEVER** defined in terms of itself!
Thus, REs cannot **define recursive statements**.

  - Two REs separated by "|" next to each other (e.g., "*non_negative_digit* | *digit*") to denote one RE or the other.
  - An RE followed by "*" (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

Review: Regular Expression Rules

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The
  - Two *igit*”) to deno

For example, REs cannot define arithmetic expressions with parentheses.

  - Two REs separated by “|” next to each other (e.g., “*non_negative_digit* | *digit*”) to denote one RE or the other.
  - An RE followed by “*” (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

Context-Free Grammars

- **Context-Free Grammars** (**CFGs**) are similar to REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Context-Free Grammars

Each rule is called a **production**.

- **Context-** … o REs except that they can hand… e **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Contex

One of the nonterminals, usually the first one, is called the **start symbol**, and it defines the construct defined by the grammar.

- **Conte** ... o REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Non-terminals are symbols that are defined by the CFG and can appear on both the **left** and **right** side of "→".

Co

- **Context-Free Grammars** (**CFGs**) are similar to REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Co

In the book, non-terminals are written in ***italics***, but in other literature they are written in **<brackets>**.

- **Context-Free Grammars** (**CFGs**) are similar to REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Co

Terminals are the strings that define the grammar and can only appear on the right side of "→".

- **Context-Free Grammars** (**CFGs**) are similar to REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

In the book non-terminals are written in **`typewriter`** font, others write it in “normal” font.

Co

- **Context-Free Grammars** (**CFGs**) are similar to REs except that they can handle **recursion**.

Arithmetic expression with parentheses

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

BNF

- Technically, the Kleene star (*) and parentheses are not allowed under the CFG rules, called **Backus-Naur Form** (**BNF**).
- However, for convenience, we will use Extended BNF that includes the Kleene star, parentheses, and the **Kleene Plus** ($^+$), which stands for “one or more iterations.”

EBNF Example.

- The Kleene star and parentheses can be we written as follows

id_list → `id` (, `id`)*

id_list → `id`

id_list → *id_list*, `id`

Derivation

- A **derivation** is “a series of replacement operations that derive a string of terminals from the start symbol.”

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* id

⇒ *expr* + id

⇒ *expr op expr* + id

⇒ *expr op* id + id

⇒ *expr* * id + id

⇒ id * id + id

(slope) * (x) + (intercept)

Derivation

- A **derivation** is "a series of replacement operations that deriv… start symbol."

⇒ denotes "derived from"

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* id

⇒ *expr* + id

⇒ *expr op expr* + id

⇒ *expr op* id + id

⇒ *expr* * id + id

⇒ id * id + id

(slope) * (x) + (intercept)

Deri

This derivation replaces the **rightmost** non-terminal. Derivations with this behavior are called (surprisingly) **rightmost** or **canonical derivation**.

- A d… …erations that deri… …mbol."

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* id

⇒ *expr* + id

⇒ *expr op expr* + id

⇒ *expr op* id + id

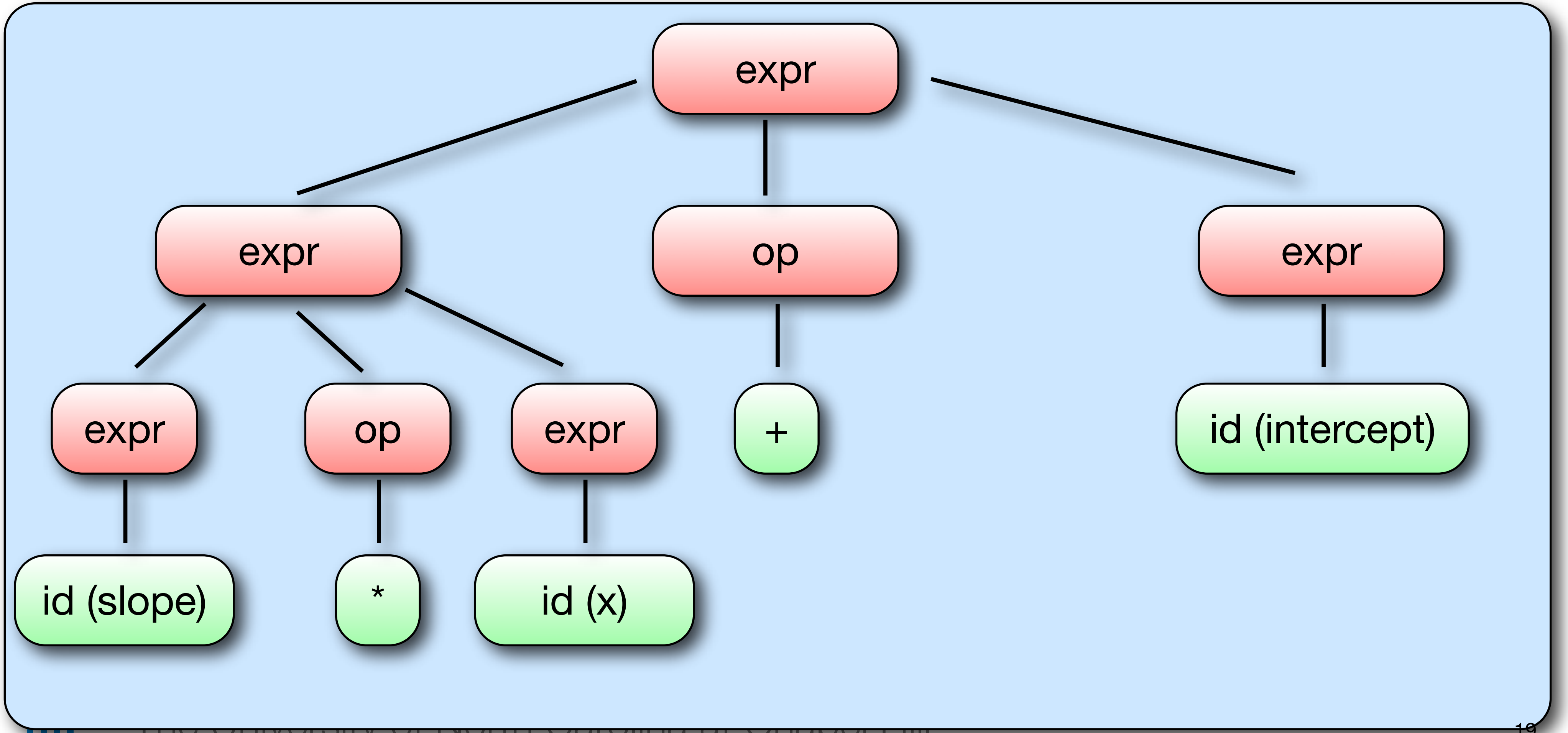
⇒ *expr* * id + id

⇒ id * id + id

(slope) * (x) + (intercept)

Parse Tree

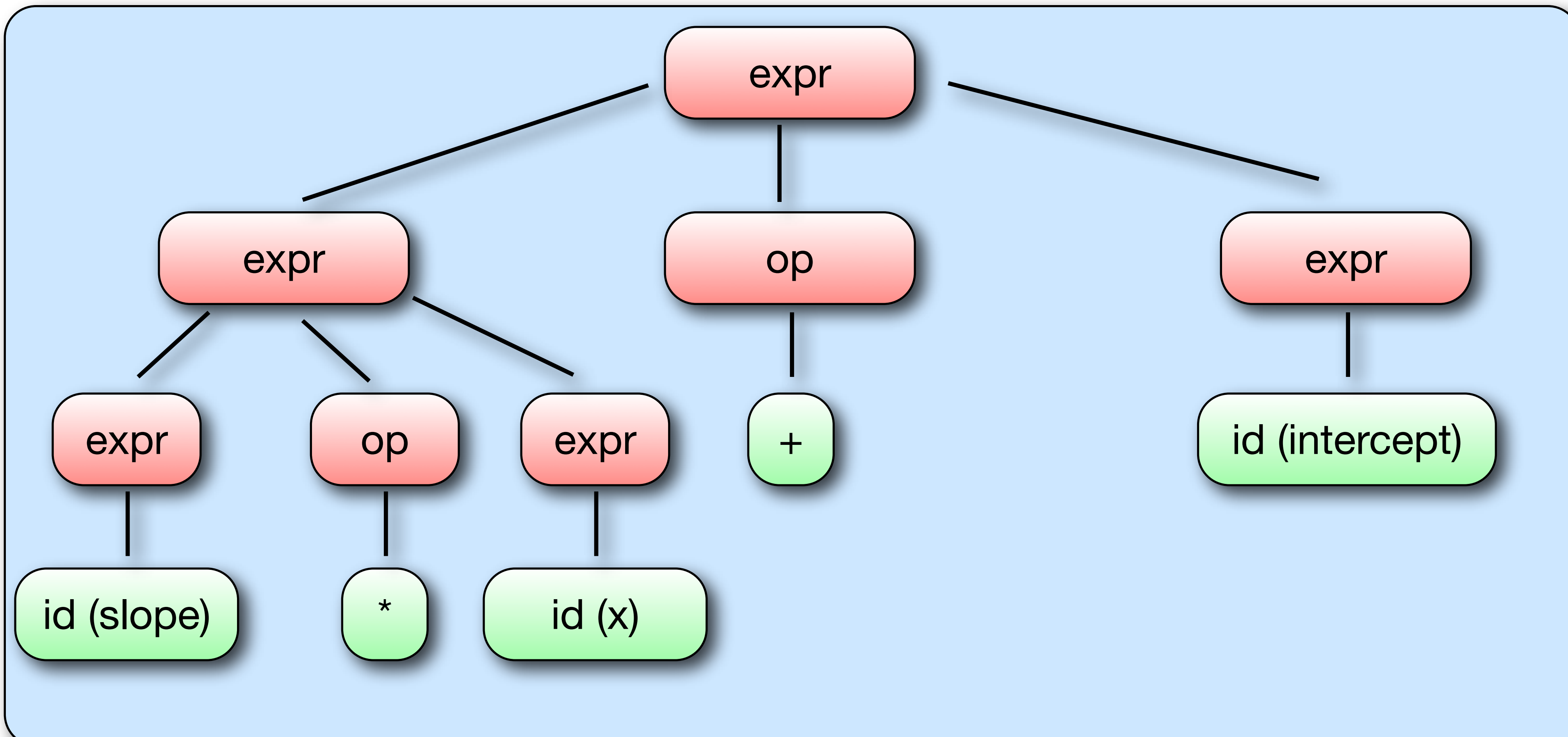
- A parse tree is the graphical representation of the derivation.



Pa

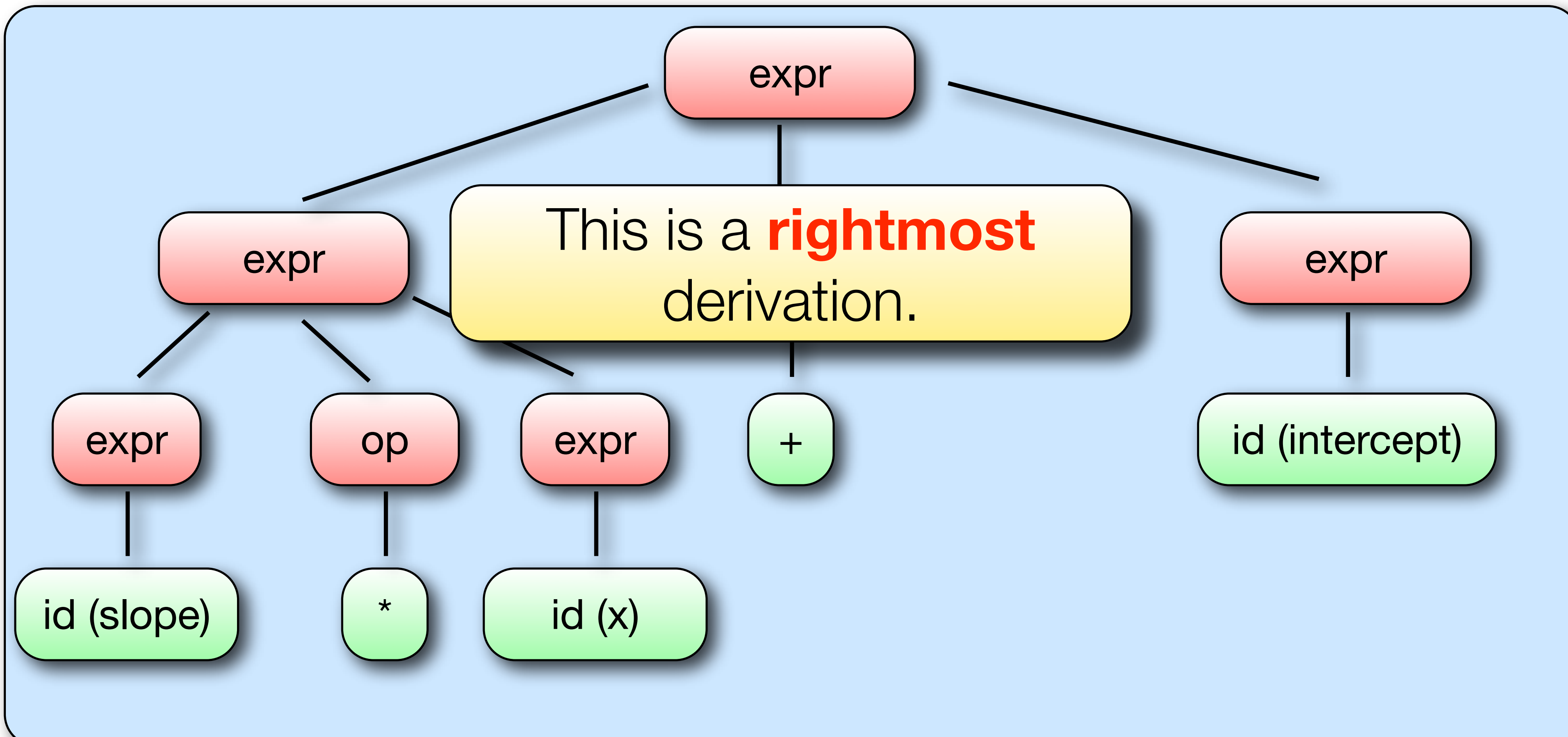
This parse tree constructs the formula
(slope*x) + intercept

- A parse tree is the graphical representation of the derivation.



Parse Tree

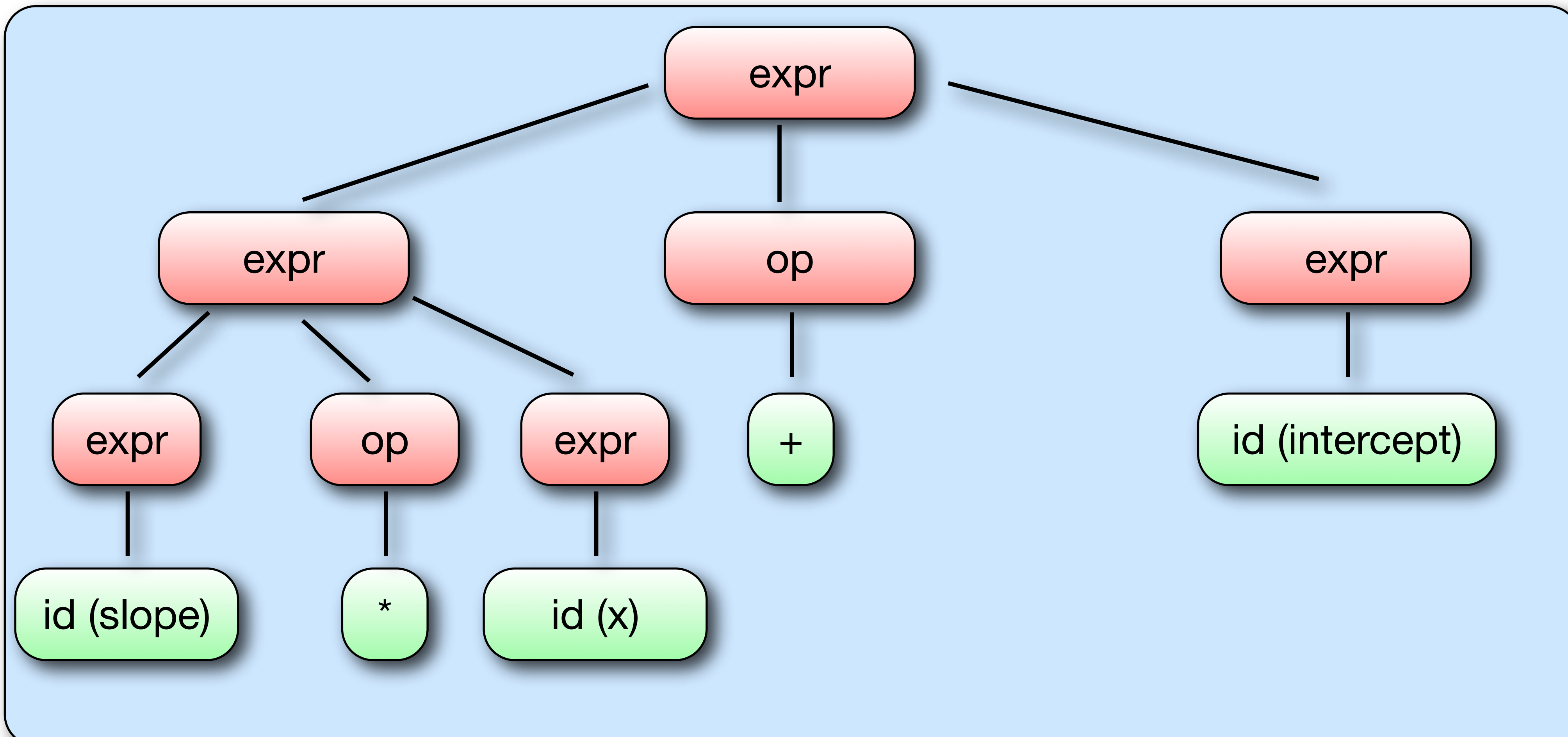
- A parse tree is the graphical representation of the derivation.



Parse Tree

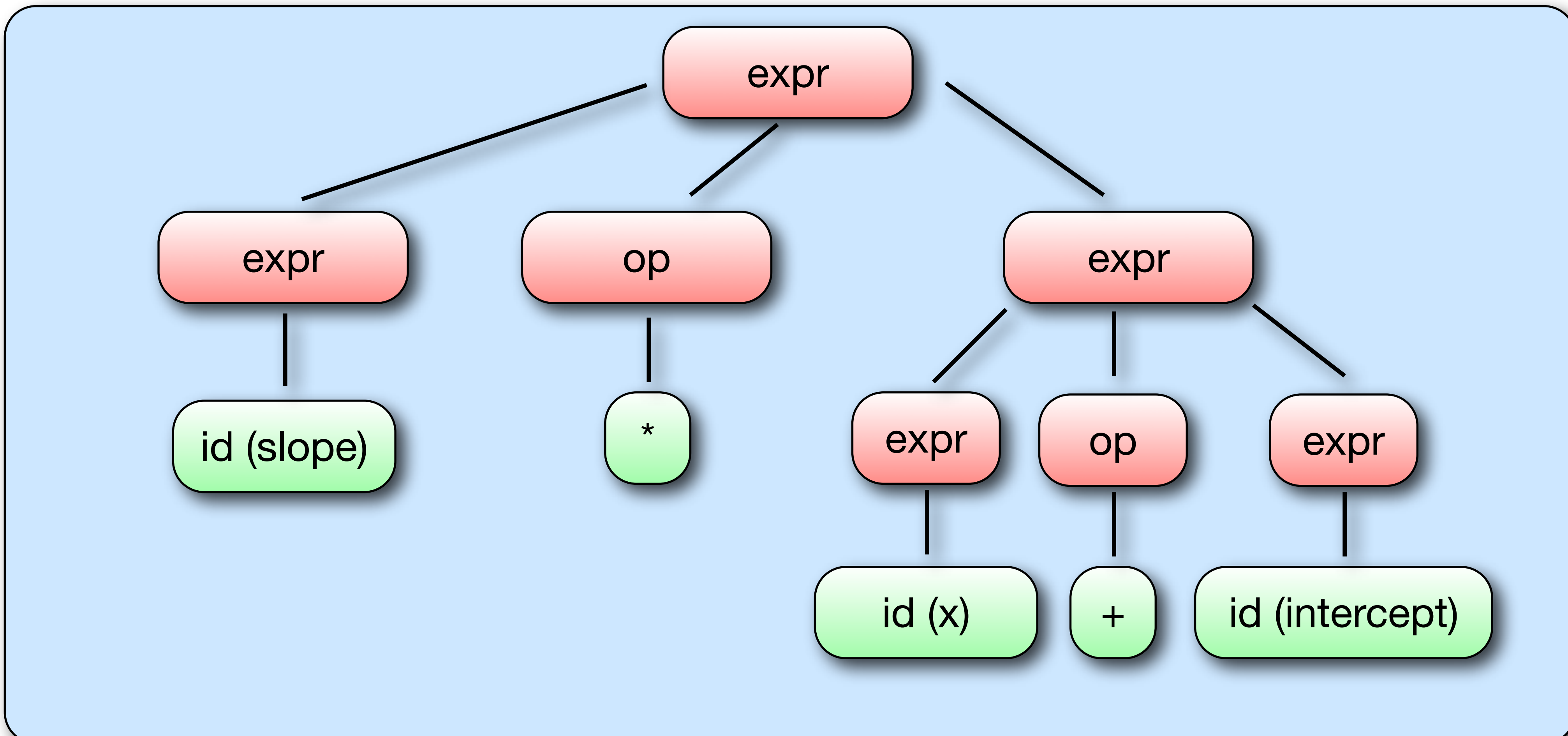
Lets try deriving “2*a*b+c”

- A parse tree is the graphical representation of the derivation.



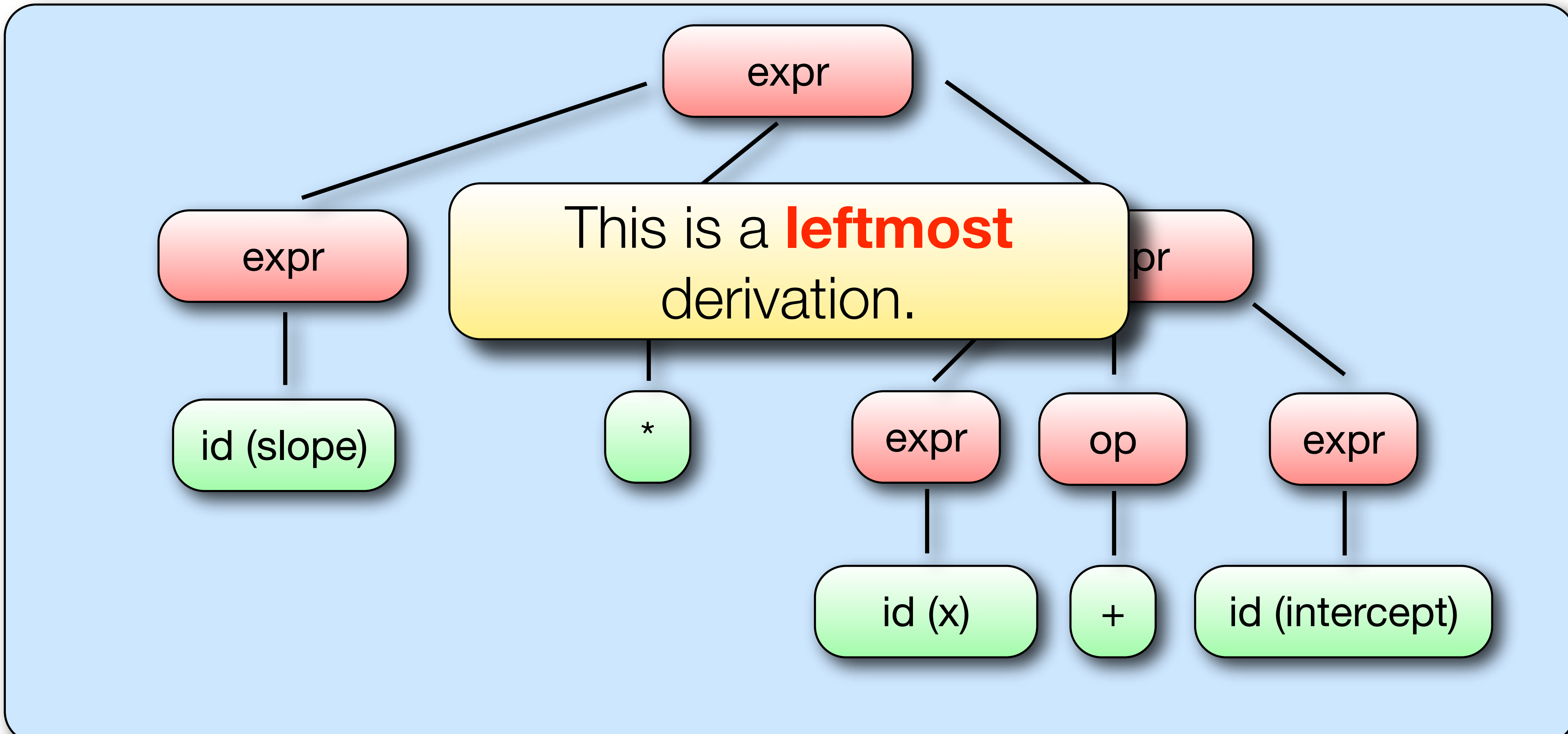
Parse Tree (Ambiguous)

- This grammar is ambiguous and can construct the following parse tree.



Parse Tree (Ambiguous)

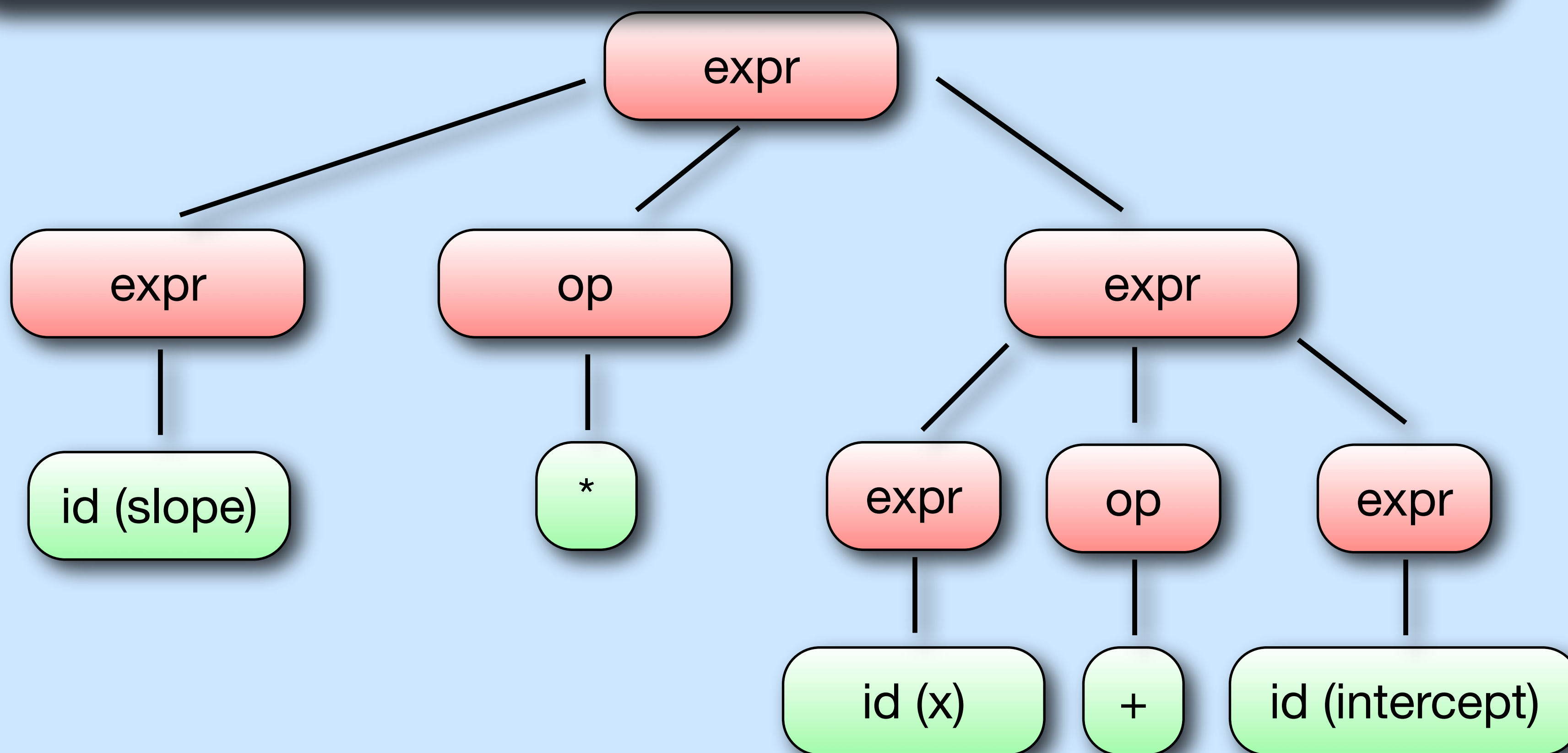
- This grammar is ambiguous and can construct the following parse tree.



Pa

- T
  fo

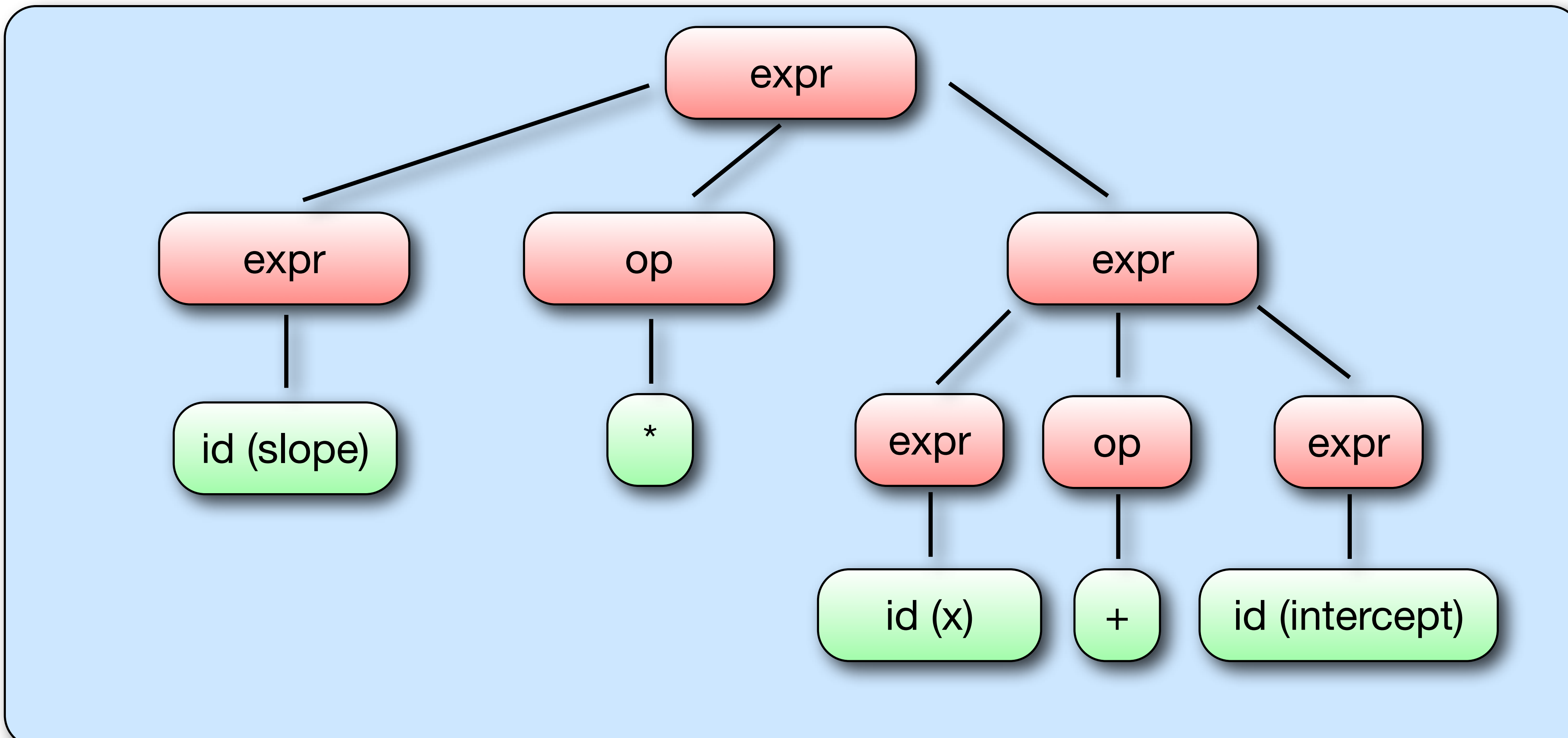
This parse tree constructs the formula **slope*(x+intercept)** which is **not equal** to **slope*x + intercept**



Parse Tree (Ar

Lets try deriving “2*a*b+c”

- This grammar, is ambiguous and can construct the following parse tree.



Disambiguating grammar

- The problem with our original grammar was that we did **not fully express the grammatical structure** (i.e., associativity and precedence).
- To create an unambiguous grammar, we need to fully specify the grammar.

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

mult_op → * | /

Disambiguating grammar

- The problem with our original grammar was that we did **not fully express the grammatical structure** (i.e., associativity and precedence
- To create an unambiguous grammar, we need to fully specify the grammar.

Gives precedence to multiply

expr → *term* | *expr add_op term*

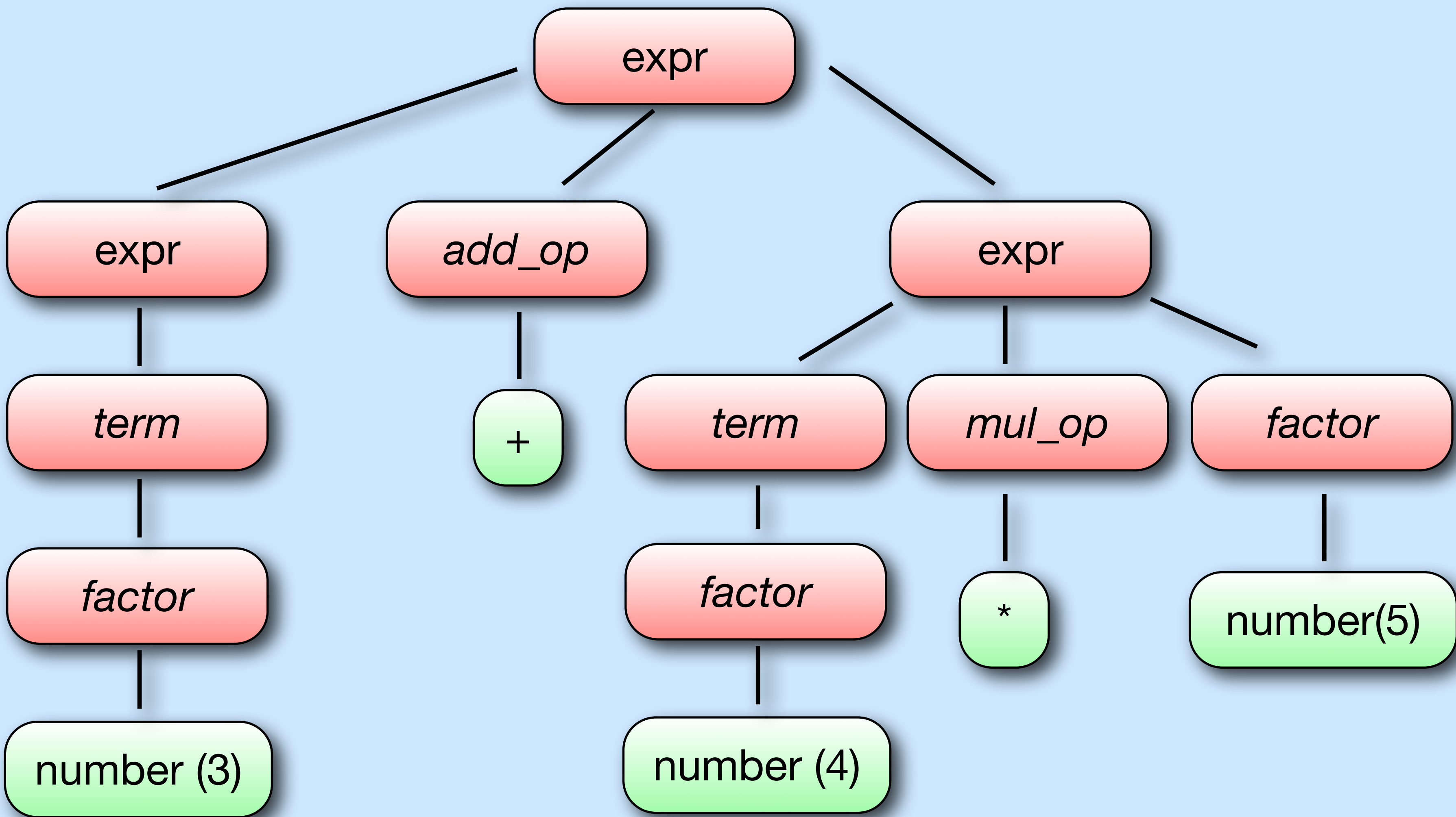
term → *factor* | *term mult_op factor*

factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

mult_op → * | /

3+4*5


expr
expr
add_op
expr
term
+
term
mul_op
factor
factor
factor
*
number(5)
number (3)
number (4)

Disambiguating grammar

- The problem … did **not fully express the** … ty and precedence).

Lets try deriving “3*4+5*6+7”

- To create an unambiguous grammar, we need to fully specify the grammar.

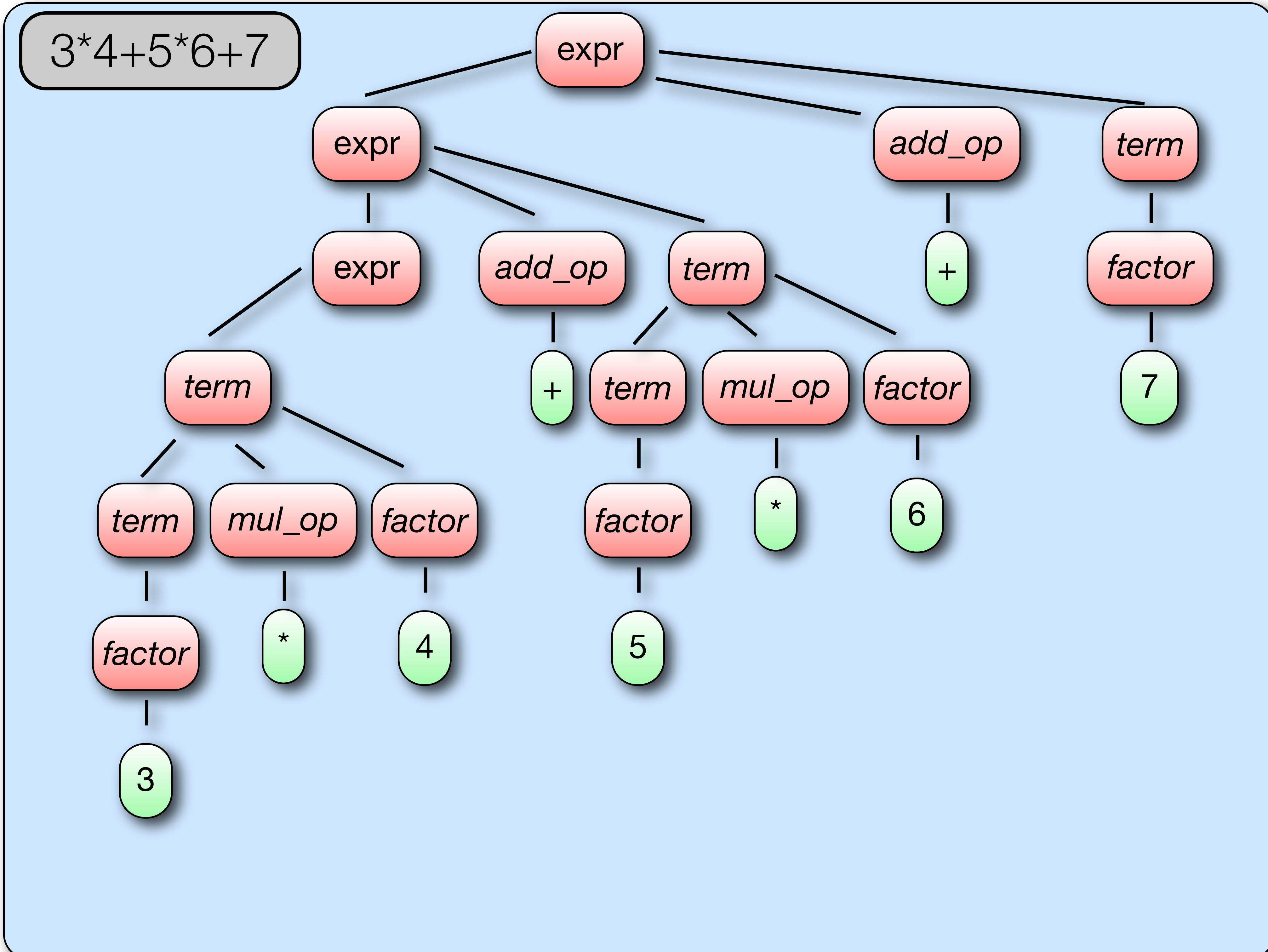
expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

mult_op → * | /


3*4+5*6+7
expr
expr
add_op
term
expr
add_op
term
+
factor
term
+
term
mul_op
factor
7
term
mul_op
factor
factor
*
6
factor
*
4
5
3

Disambiguating grammar

- The problem ... did **not fully express the** ... ty and precedence).

Lets try deriving “3*4” & “3+4”

- To create an unambiguous grammar, we need to fully specify the grammar.

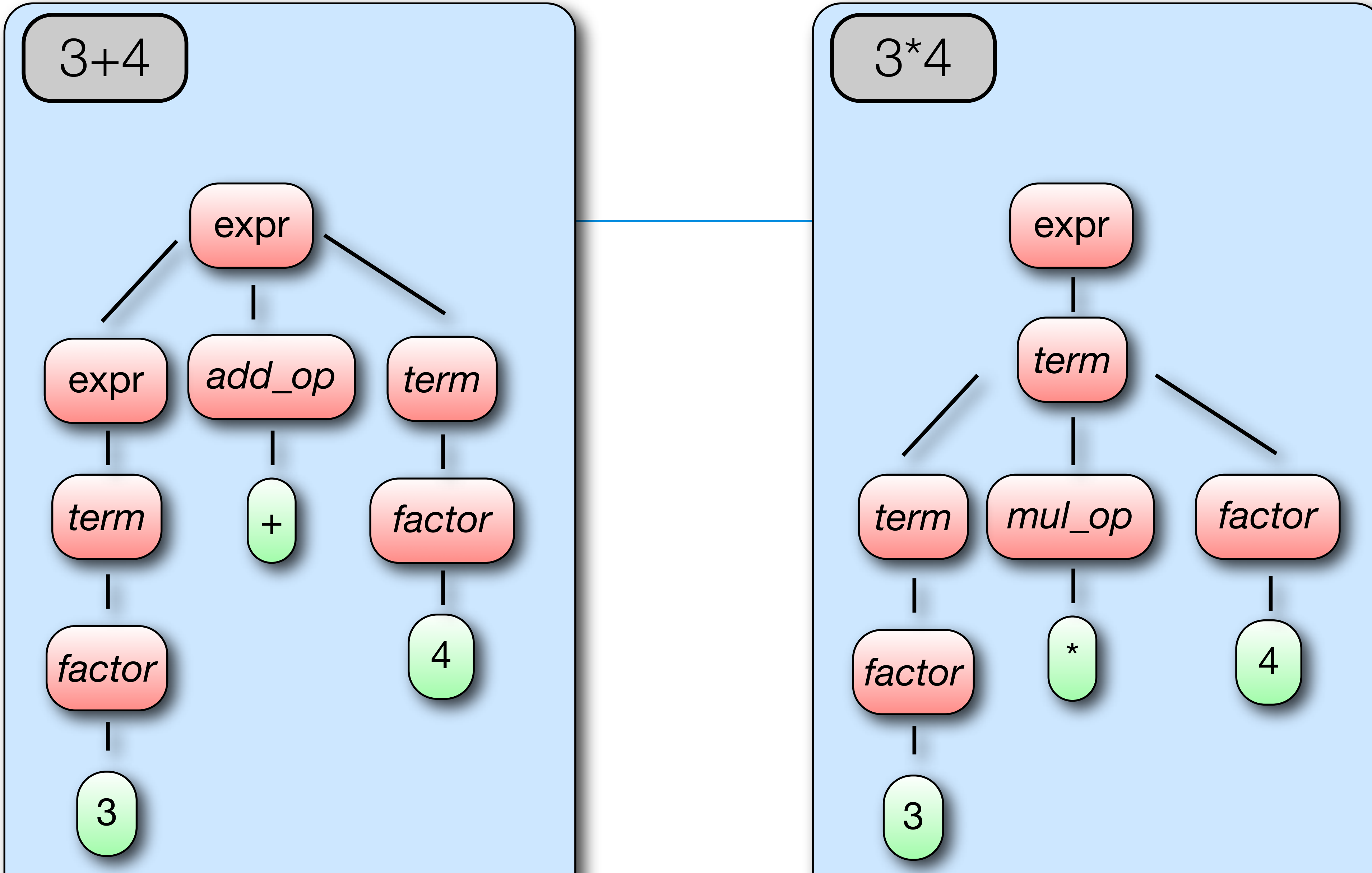
expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

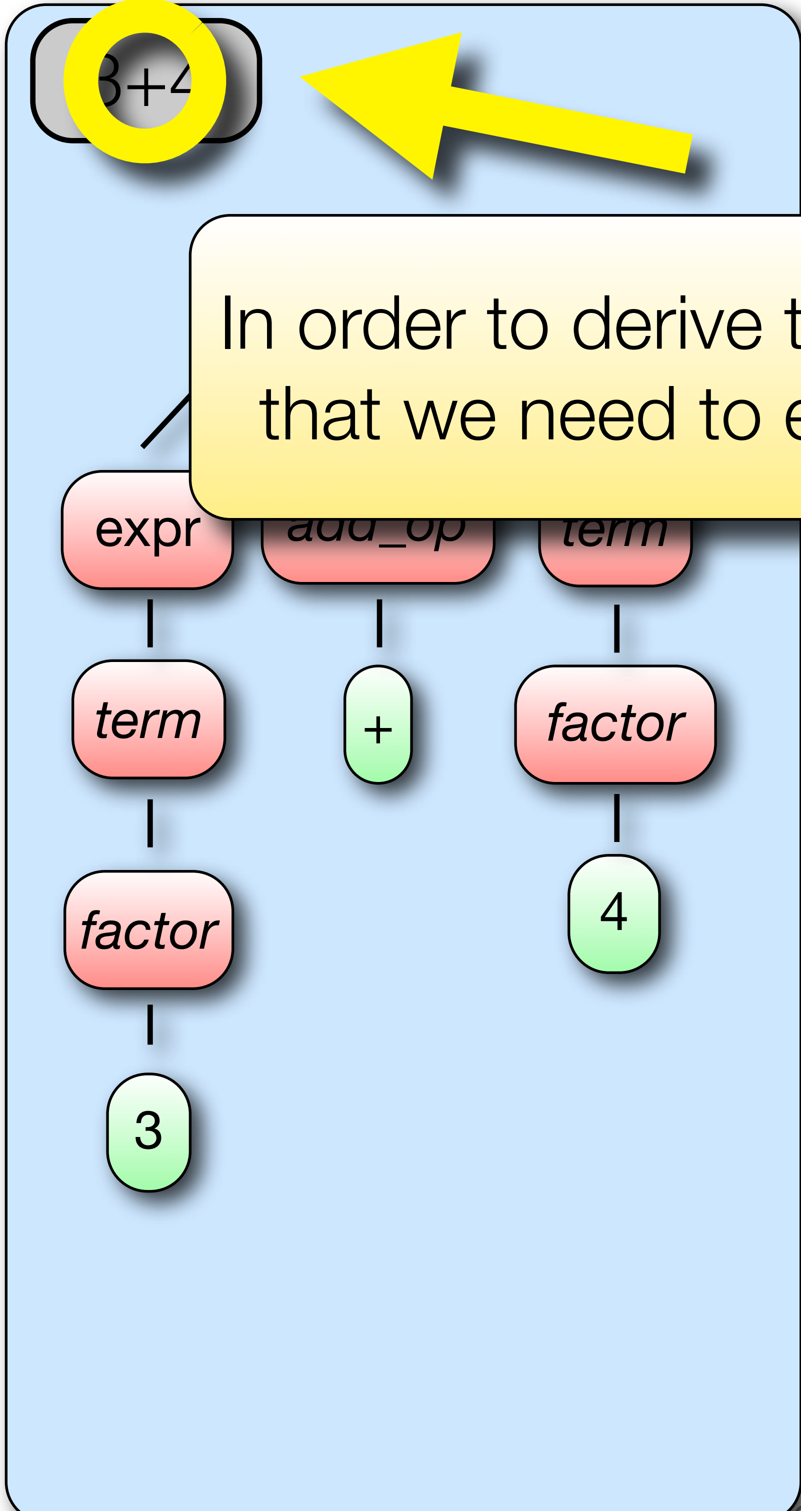
factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

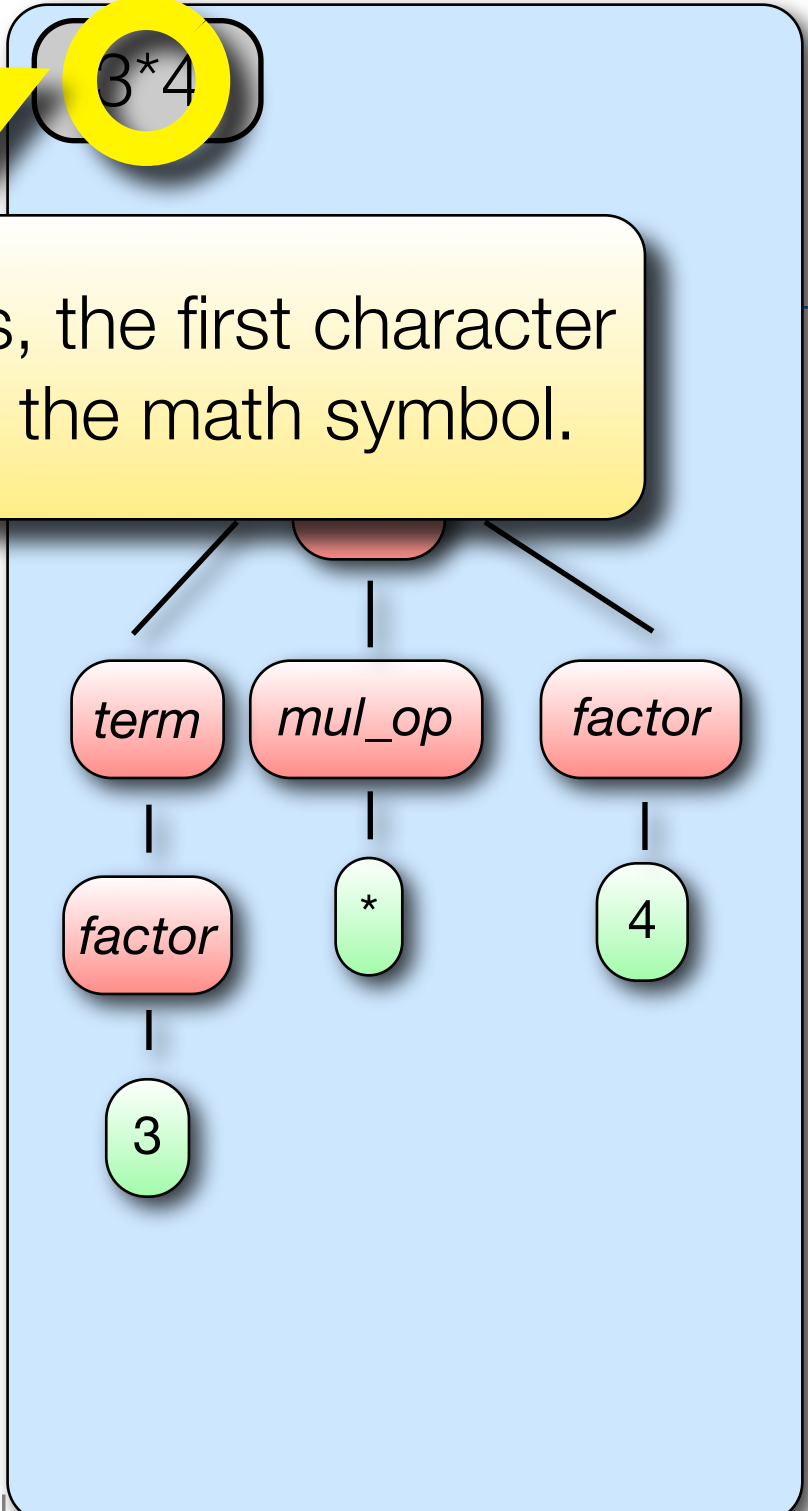
mult_op → * | /

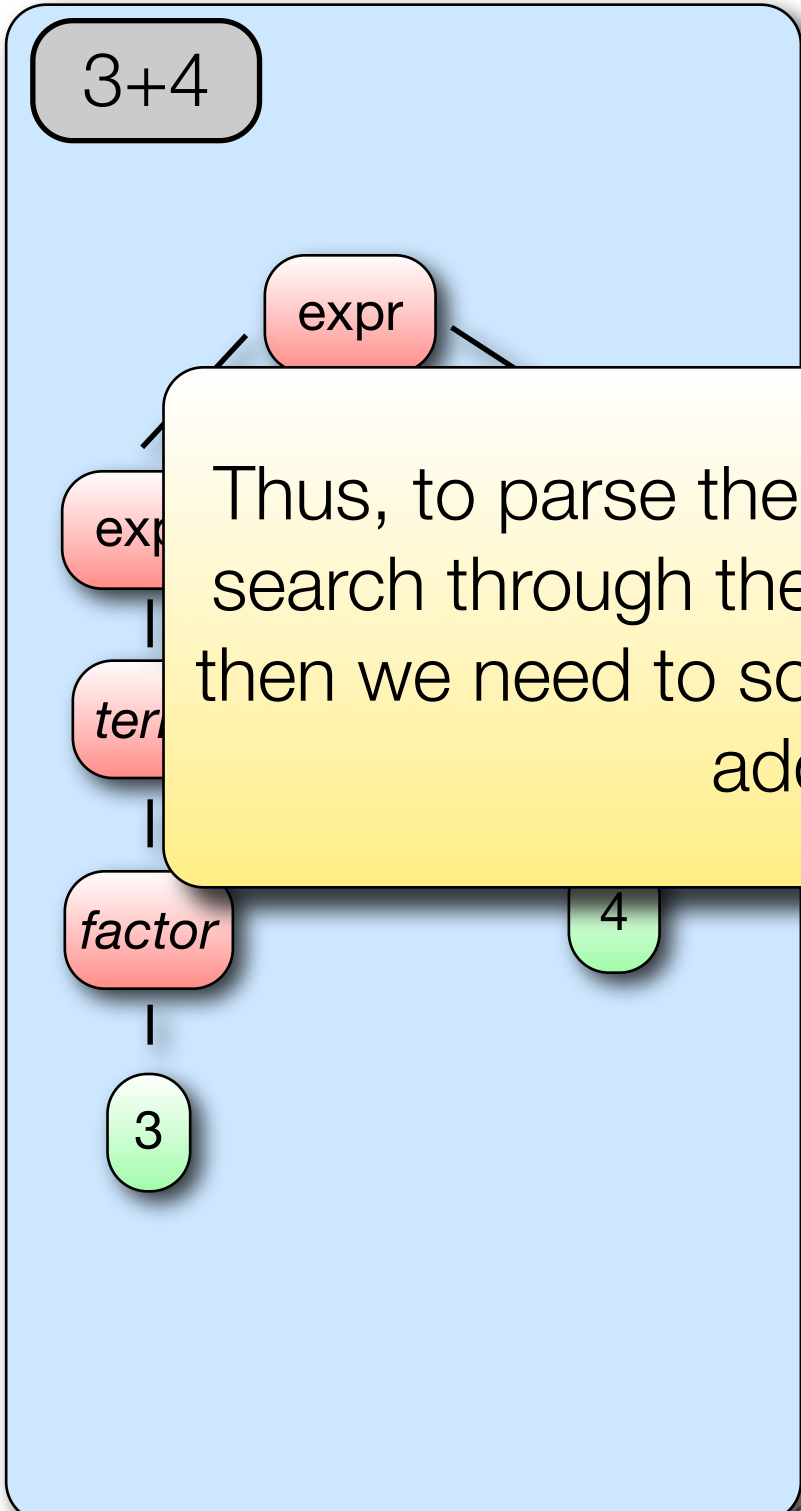


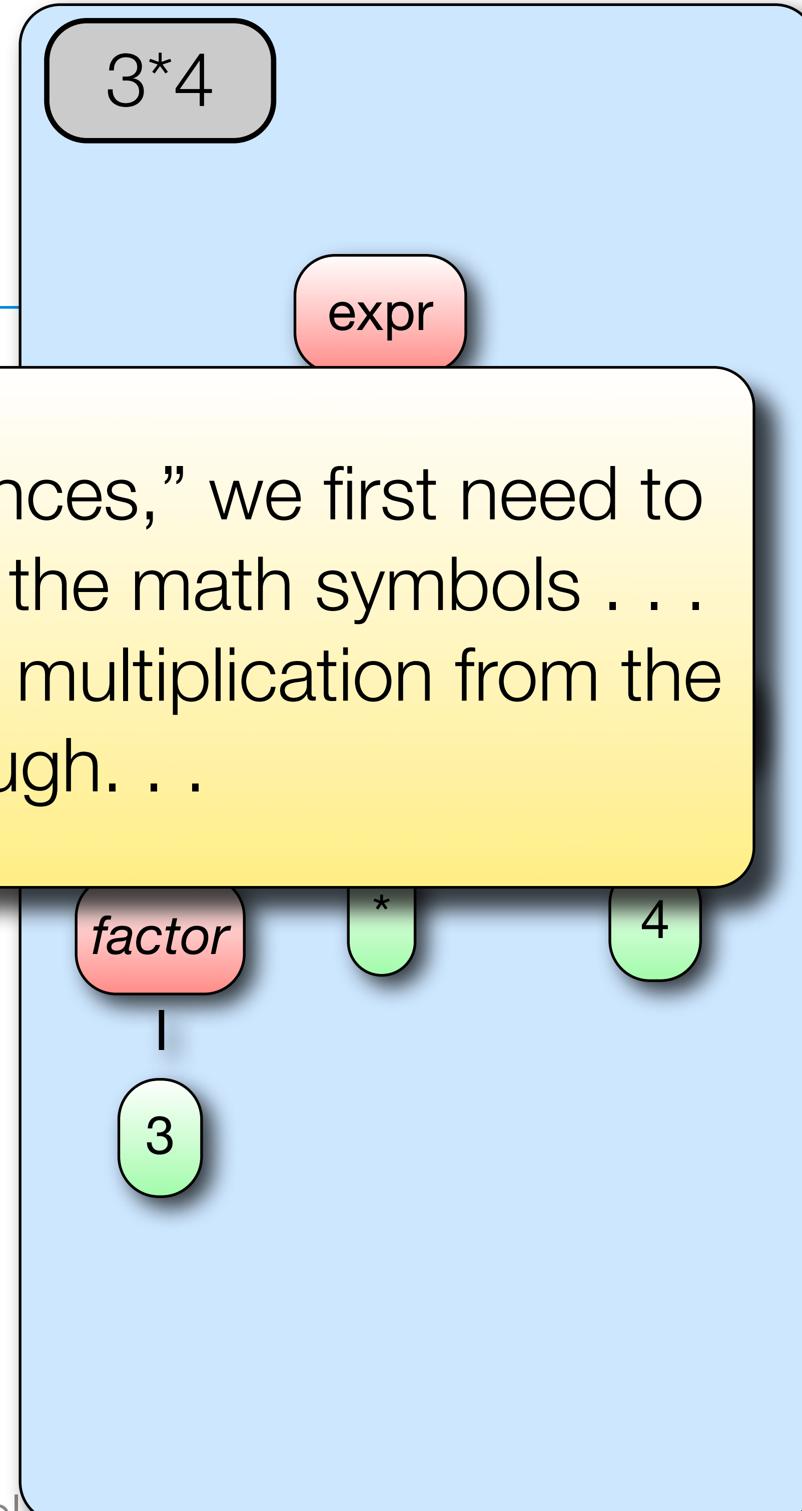
How can you derive these trees by examining one character at a time?



In order to derive these trees, the first character that we need to examine is the math symbol.







Thus, to parse these “sentences,” we first need to search through them to find the math symbols . . . then we need to sort out the multiplication from the addition. . . ugh. . .

Java Spec

- Available on-line
  - http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
- Examples
  - Comments: http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html#48125
  - Multiplicative Operators: http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#239829
  - Unary Operators: http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#4990

LL and LR Derivation

- CFGs can be parsed in **O(n^3)** time, where n is length of the program.
- This is too long for most code; however, there are two types of grammars that can be parsed in **linear time**, i.e., **O(n)**.
  - **LL**: “Left-to-right, Left-most derivation”
  - **LR** “Left-to-right, Right-most derivation”

Top-down

- **LL parsers** are top-down, i.e., they identify the non-terminals first and terminals second.
  - **LL grammars** are grammars that can be parsed by an LL parser.

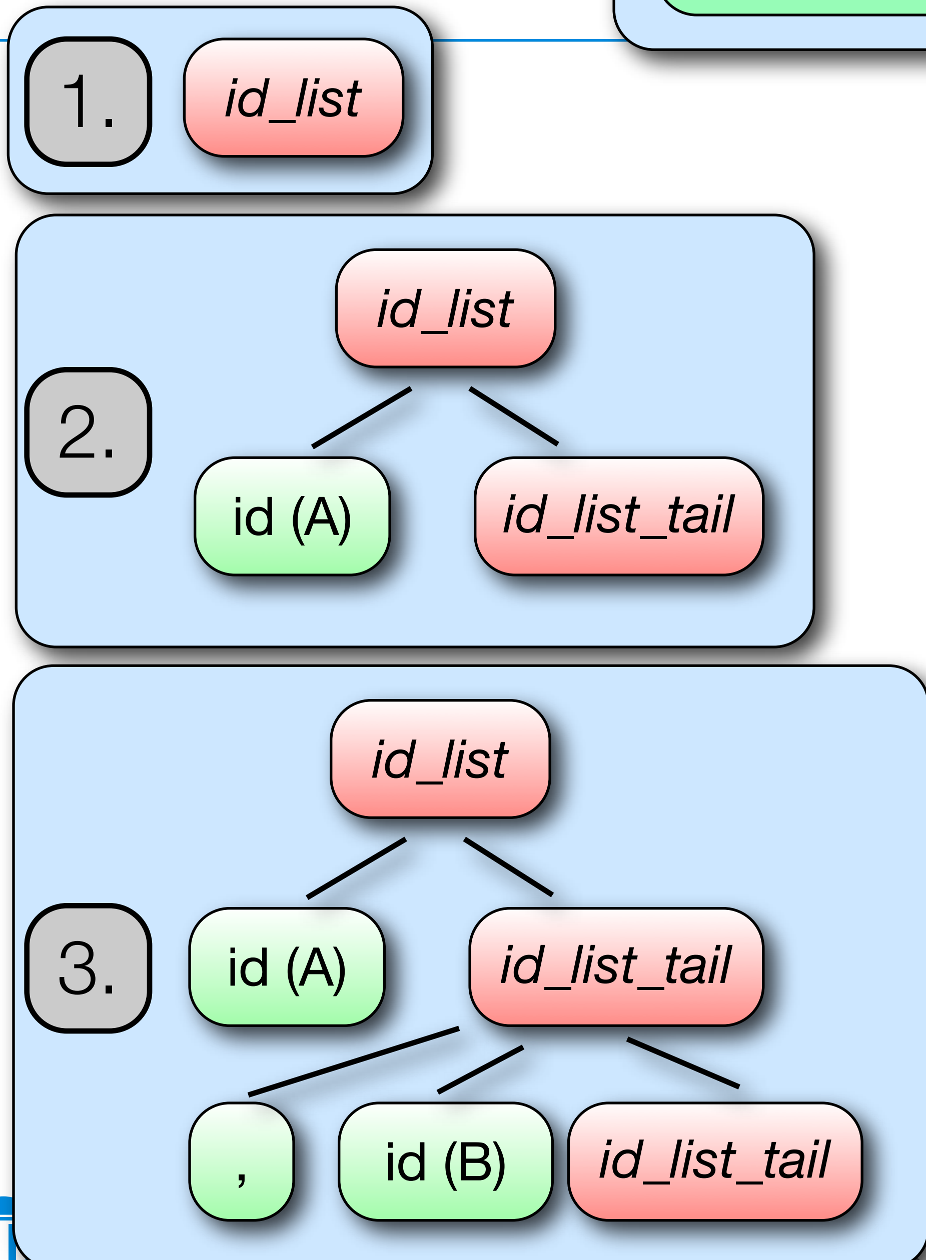
Top-Down

id_list → id *id_list_tail*

id_list_tail → ;

id_list_tail → , id *id_list_tail*

A, B, C;



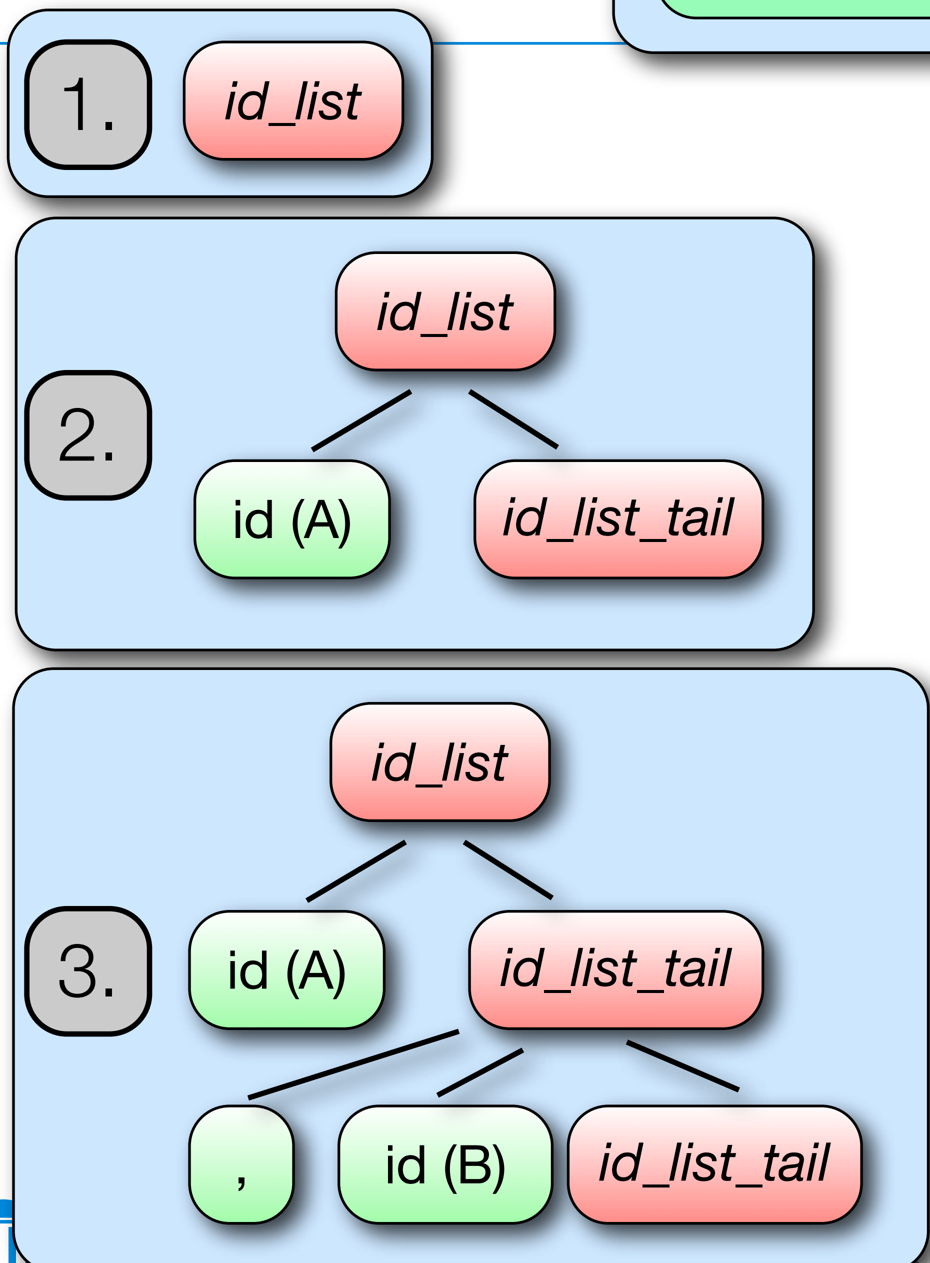
Top-Down

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → `;`

A, B, C;



LL parsers are some time called **predictive** because they “predict” the next state.

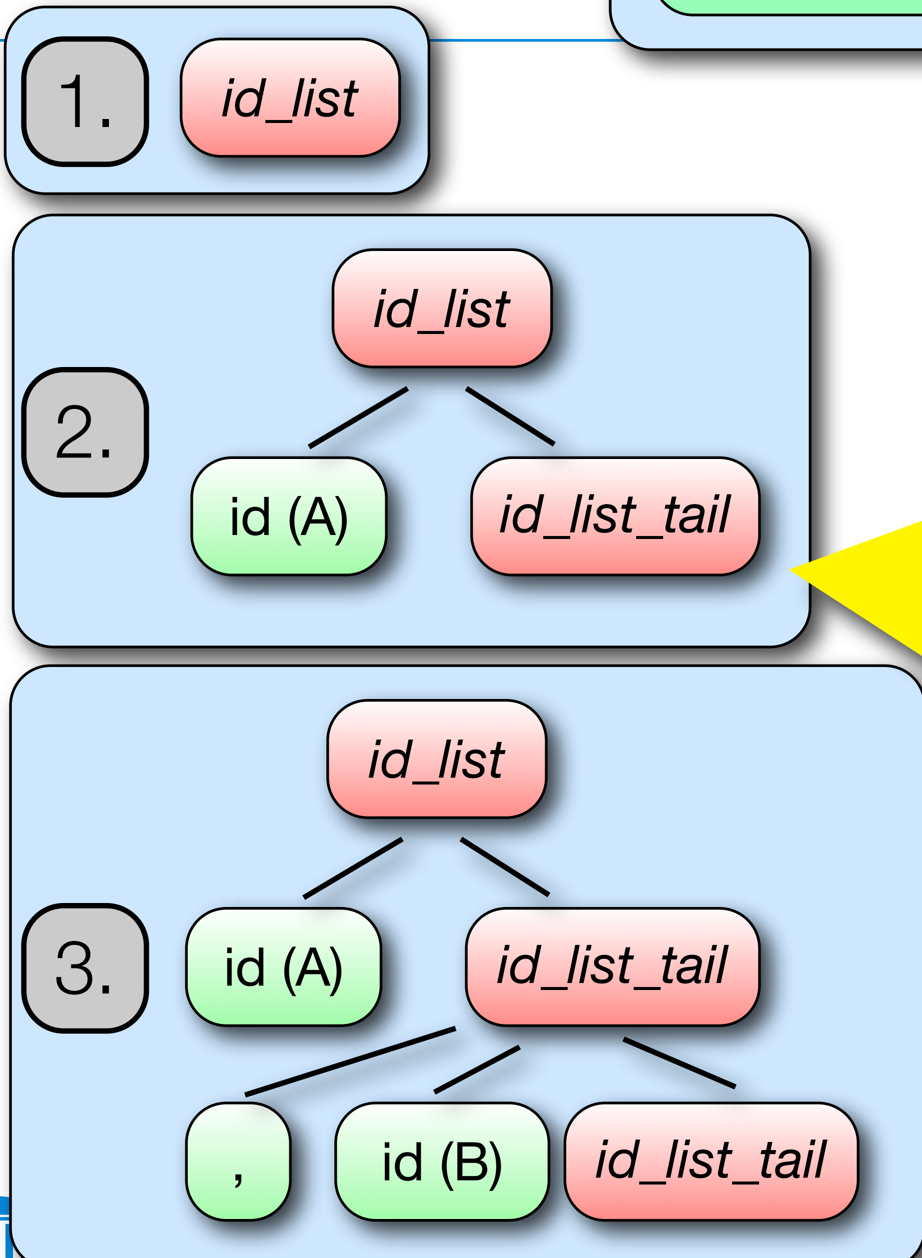
Top-Down

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → ;

A, B, C;



For example, after *id(A)* is “discovered”, the next state is “predicted as *id_list_tail*.

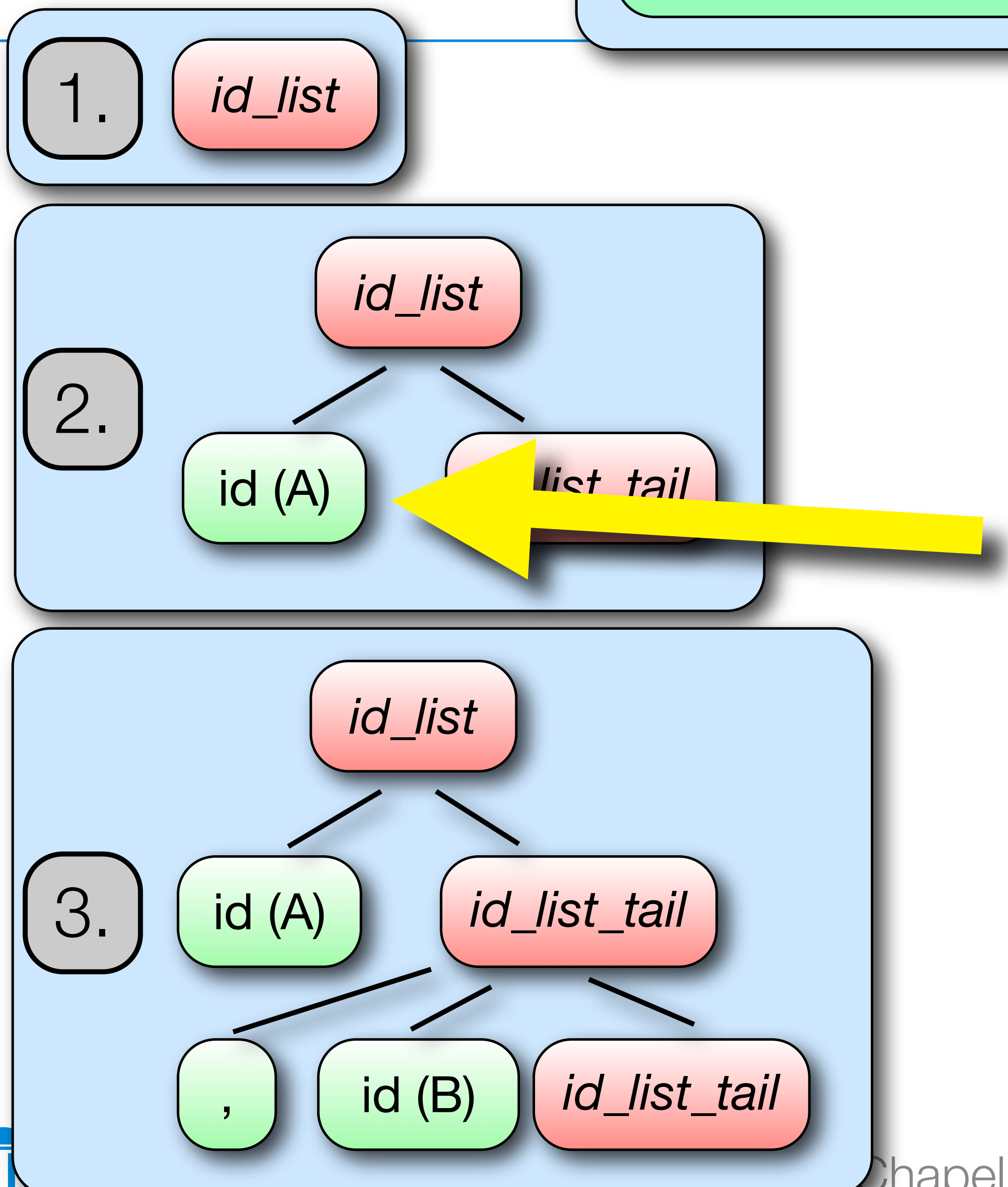
Top-Down

id_list → `id` *id_list_tail*

id_list_tail → `;`

id_list_tail → `, id` *id_list_tail*

A, B, C;



Notice that tokens are placed in the tree from the **left-most** to **right-most**.

Bottom-up

- **LR parsers** are bottom-up, i.e., they discover the terminals first and non-terminals second.
  - **LR grammars** are grammars that can be parsed by an LR parser.
  - All **LL grammars are LR grammars** but not vice versa.

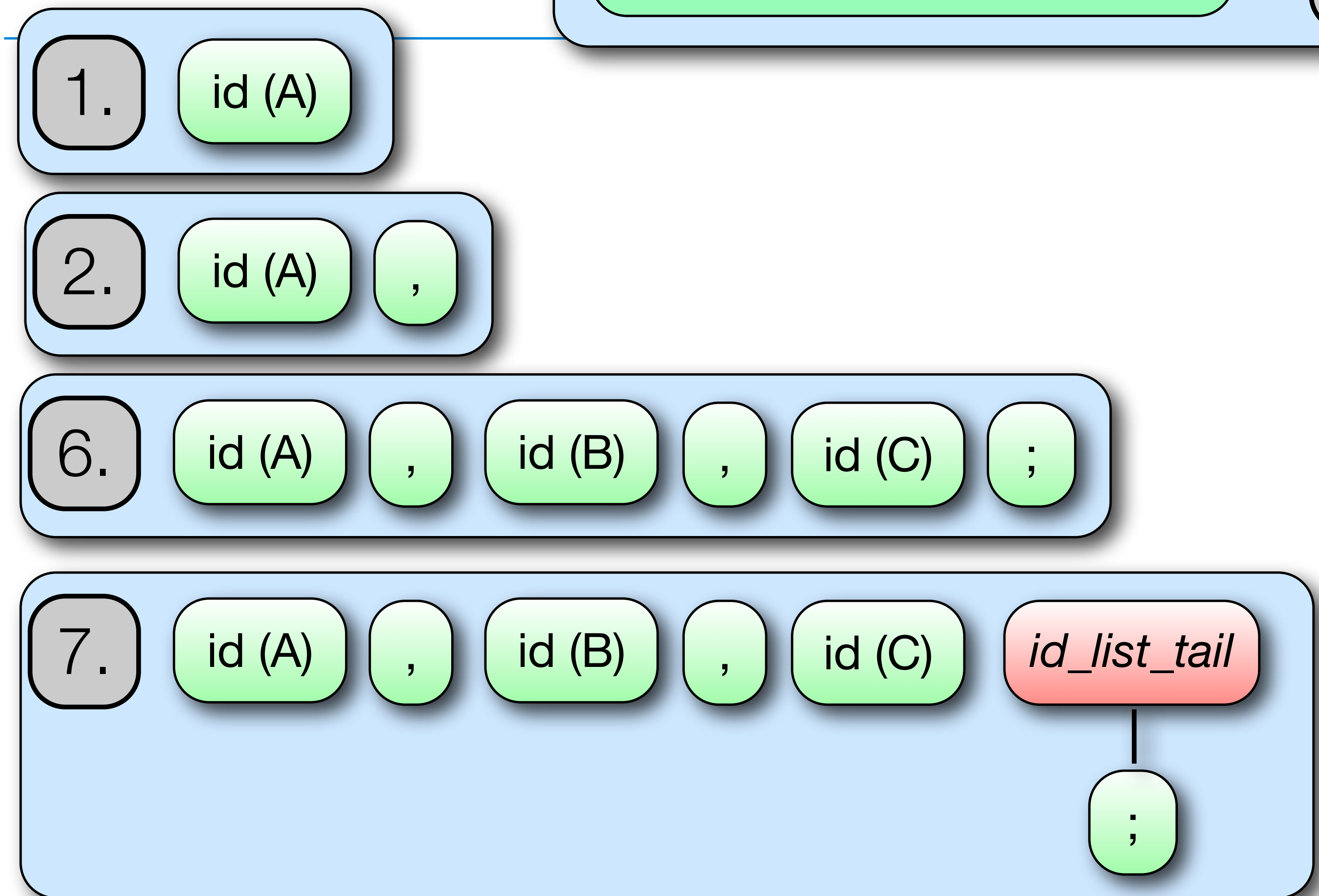
Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → ;

id_list_tail → , `id` *id_list_tail*

A, B, C;



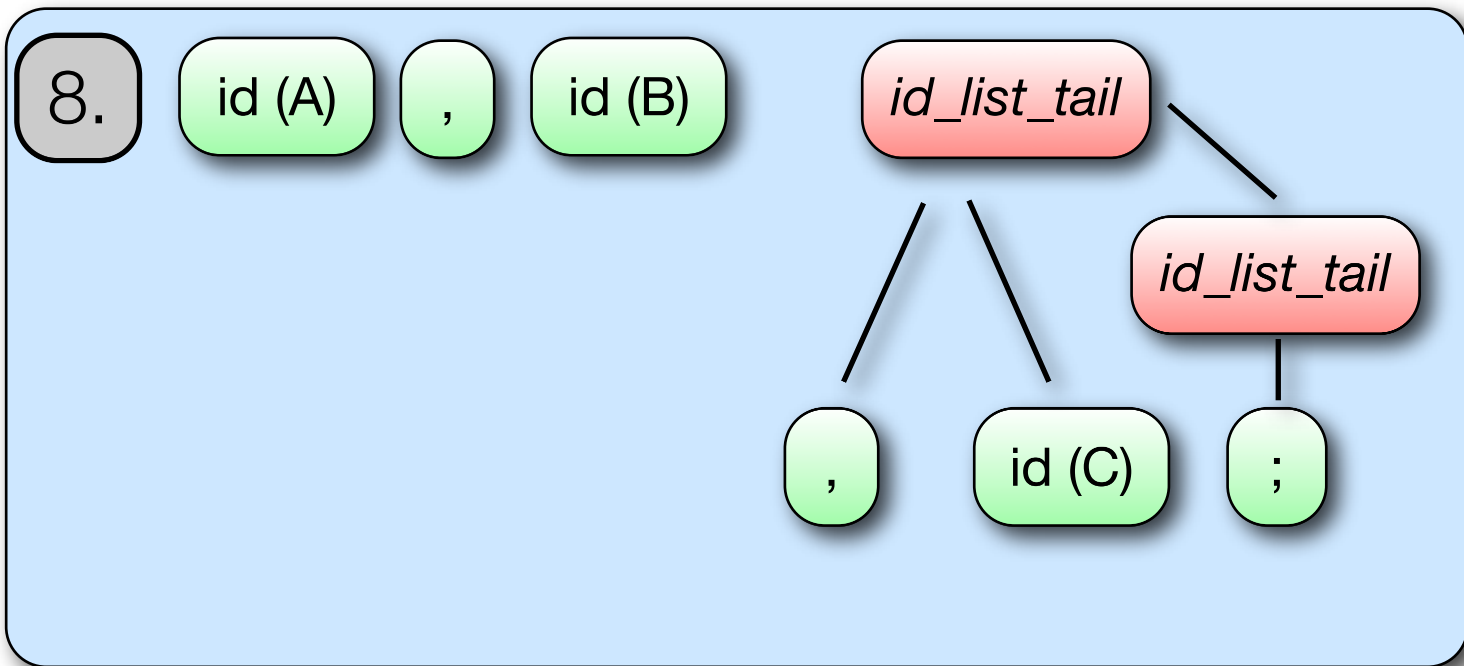
Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → ;

id_list_tail → , `id` *id_list_tail*

A, B, C;



Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → ;

id_list_tail → , `id` *id_list_tail*

A, B, C;

LR parsers are sometimes called **shift** because they "shift" the states

1. id (A)

2. id (A) ,

6. id (A) , id (B) , id (C) ;

7. id (A) , id (B) , id (C) *id_list_tail*

;

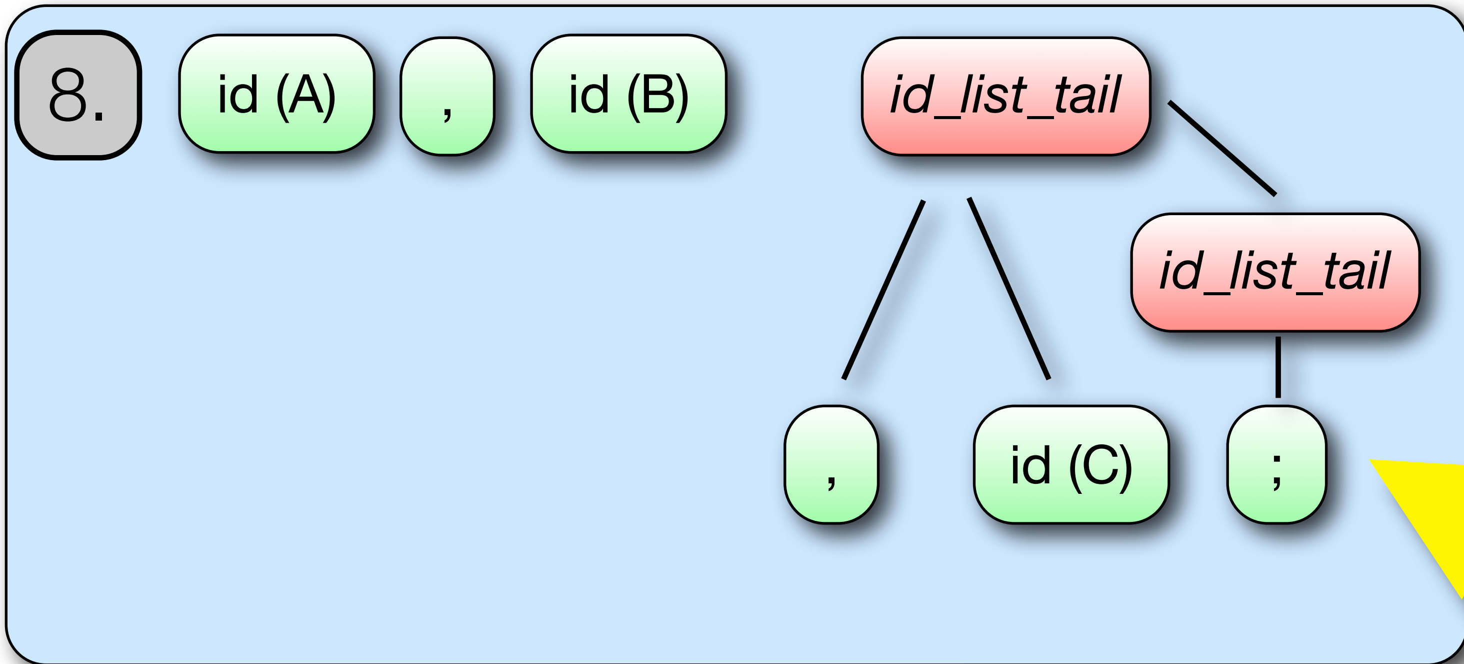
Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → , `id` *id_list_tail*

id_list_tail → ;

A, B, C;



Notice that tokens are added to the tree from the **right-most** to the **left-most**.

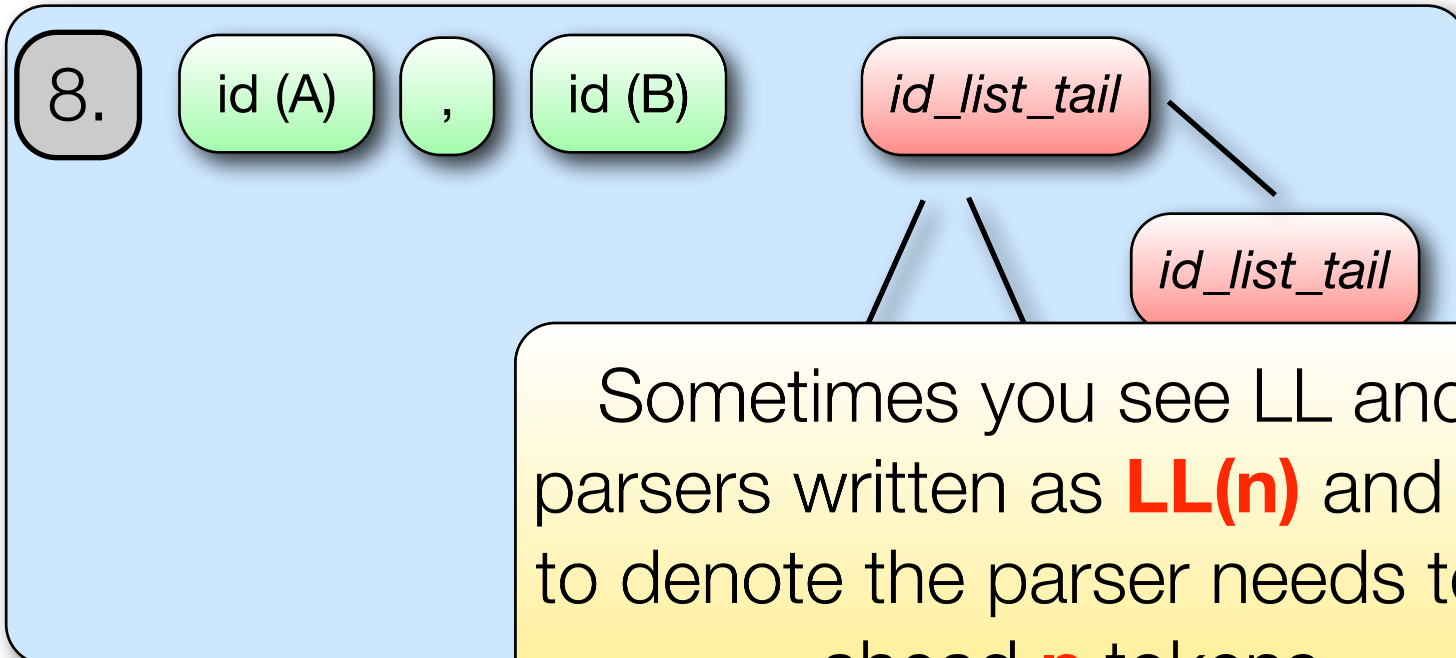
Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → ;

id_list_tail → , `id` *id_list_tail*

A, B, C;



Sometimes you see LL and LR parsers written as **LL(n)** and **LR(n)** to denote the parser needs to look ahead **n** tokens .

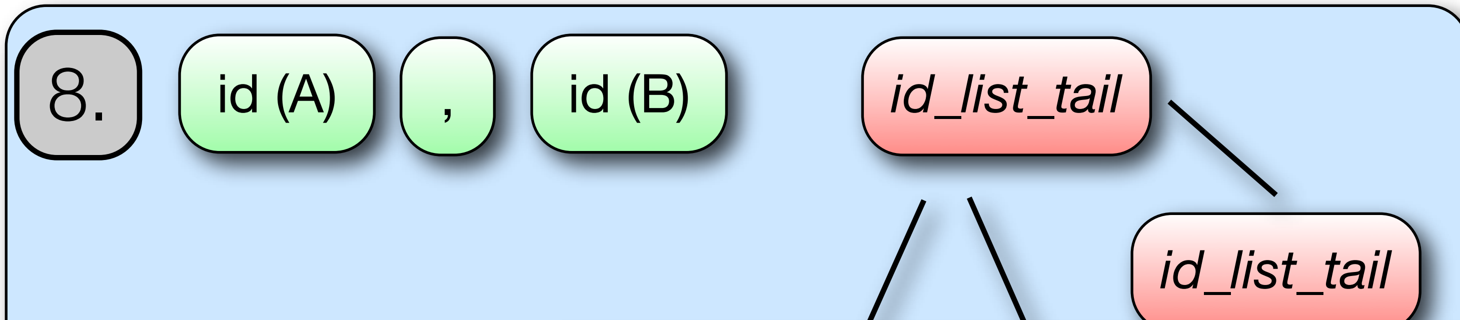
Bottom-up

id_list → `id` *id_list_tail*

id_list_tail → ;

id_list_tail → , `id` *id_list_tail*

A, B, C;



The problem with this grammar is that it can require an **arbitrarily large** number of terminals to be "shifted" before placing them into the tree

A better bottom-up grammar

This grammar limits the number of "suspended" non-terminals.

id_list → *id_list_prefix ;*

id_list_prefix → *id_list_prefix* **,** `id`

id_list_prefix → `id`

A better bottom-up grammar

Lets try parsing “A, B, C;”

id_list → *id_list_prefix ;*

id_list_prefix → *id_list_prefix* , `id`

id_list_prefix → `id`

A better bottom-up grammar

However, it **cannot** be parsed by LL (top-down) parser. Since when the parser discovers an “id” it does not “know” the number of “id_list_prefixs”

id_list → *id_list_prefix* ;

id_list_prefix → *id_list_prefix* , `id`

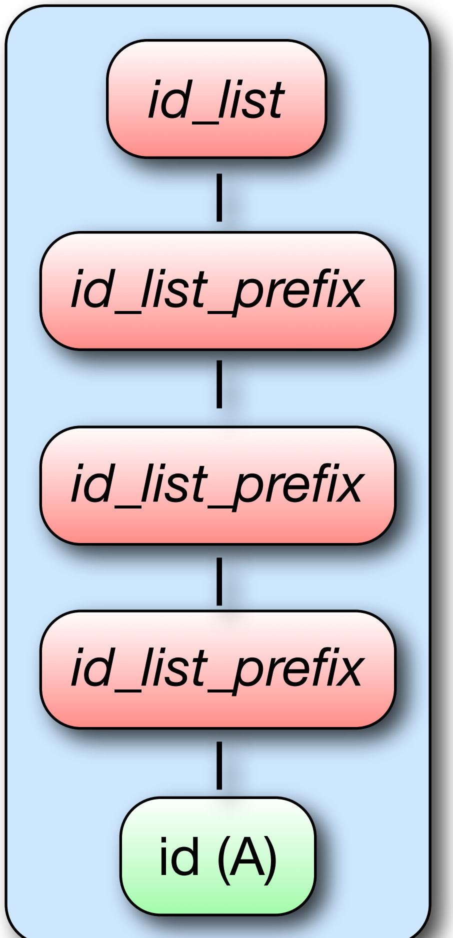
id_list_prefix → `id`

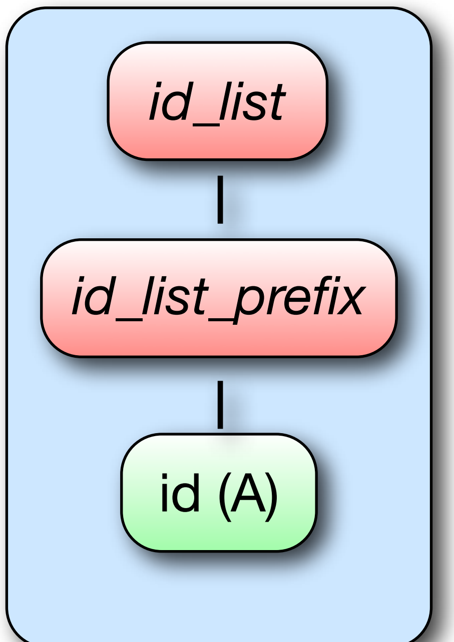
A better bottom-up gramm

id_list → *id_list_prefix* ;

id_list_prefix → *id_list_prefix* , `id`

id_list_prefix → `id`





Both of these are valid break downs, but we don't know which one. Therefore, is **NOT** a valid **LL grammar**, but it is a valid **LR grammar**.

program → *stmt_list* $$

stmt_list → *stmt stmt_list* | ε

stmt → `id:=` *expr* | `read id` | `write` *expr*

expr → *term term_tail*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

This grammar (for the calculator) unlike the previous calculator grammar is an LL grammar, because when an "id" is encountered **we know exactly where it belongs**.

program → *stmt_list* $$

stmt_list → *stmt stmt_list* | ε

stmt → `id:=` *expr* | `read id` | `write` *expr*

expr → *term term_tail*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

Let's try "c := 2*A+B"

Let's try "2*A+B"

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

mult_op → * | /

Let's try "c := 2*A+B"

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*

op → + | - | * | /

Recursive Descent & LL Parse Table

- There are two ways to code a parser for LL grammars:
  - **Recursive Descent**, which is a recursive program with case statements that correspond to each one-to-one to nonterminals.
  - **LL Parse Table**, which consist of an iterative driver program and a table that contains all of the nonterminals.

program → *stmt_list* $$

stmt_list → *stmt stmt_list* | ε

stmt → `id:=` *expr* | `read id` | `write` *expr*

expr → *term term_tail*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

Recursive Descent

```
procedure program()
  case in_tok of
    id, read, write, $$:
      stmt_list()
      match($$)
  else
    return error
```

```
procedure stmt()
  case in_tok of
      id: match(id); match(:=); expr()
      read: match(read); match(id)
      write: match(write); expr()
    else
      return error
```

```
procedure stmt_list()
  case in_tok of
    id, read, write:
      stmt(); stmt_list();
    $$:
      skip
    else
      return error
```

```
procedure match(expec)
    if in_tok = expec
      consume in_tok
    else
      return error
```

Recursive Descent

procedure program
  case in_tok **of**
    `id, read, wr`
      stmt_list()
      match($$)
  else
    return error

- **in_tok** is a global variable that is the current token
- **consume** changes in_tok to the next token.

n(:=); expr()
match(`id`)
e); expr()

return error

procedure stmt_list()
  case in_tok **of**
    `id, read, write:`
      stmt(); stmt_list();
    `$$:`
      skip
    else
      return error

procedure match(expec)
  if in_tok = expec
    consume in_tok
  else
    return error

Recursive Descent

```
procedure program()
  case in_tok of
    id, read, write, $$:
      stmt_list()
      match($$)
  else
    return error
```

```
procedure stmt()
  case in_tok of
    id: match(id); match(:=); expr()
    read: match(read); match(id)
    write: match(write); expr()
  else
    return error
```

```
procedure stmt_list()
  case in_tok of
    id, read, write:
      stmt(); stmt_list();
    $$:
      skip
  else
    return error
```

```
procedure match(expec)
  if in_tok = expec
    consume in_tok
  else
    return error
```

Recursive De

The question is how do we label the case statements?

```
procedure program()
   case in_tok of
      id, read, write, $$:
         stmt_list()
         match($$)
   else
      return error
```

```
procedure stmt()
   case in_tok of
      id: match(id); match(:=); expr()
      read: match(read); match(id)
      write: match(write); expr()
   else
      return error
```

```
procedure stmt_list()
   case in_tok of
      id, read, write:
         stmt(); stmt_list();
      $$:
         skip
      else
         return error
```

```
procedure match(expec)
      if in_tok = expec
         consume in_tok
      else
         return error
```

First, Follow, and Predict

- Three functions allow us to label the branches
  - **FIRST(a)**: The terminals (and ε) that can be the first tokens of the non-terminal symbol **a**.
  - **FOLLOW(A)**: The terminals that can follow the terminal or non-terminal symbol **A**
  - **PREDICT(A → a)**: The terminals that can be the first tokens as a result of the production **A → a**

program → *stmt_list* $$

stmt_list → *stmt stmt_list* | ε

stmt → `id:=` *expr* | `read id` | `write` *expr*

expr → *term term_tail*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

- FIRST(*program*) = {`id`, `read`, `write`, `$$`}
- FOLLOW(*program*) = {ε}
- PREDICT(*program* → *stmt_list* `$$`) = {`id`, `read`, `write`, `$$`}
- FOLLOW(`id`) = {`+`, `-`, `*`, `/`, `)`, `:=`, `id`, `read`, `write`, `$$`}

program → *stmt_list* $$

stmt_list → *stmt stmt_list* | ε

stmt → `id:=` *expr* | `read id` | `write` *expr*

expr → *term term_tail*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

- FIRST(*factor_tail*) ={`*`, `/`, ε}
- FOLLOW(*factor_tail*) = {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
- PREDICT(*factor_tail* → *m_op factor factor_tail*) = {`*`, `/`}
- PREDICT(*factor_tail* → ε) = {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}

program → *stmt_list* $$

stmt_list → *st*

stmt → `id:=`

expr → *term t*

term_tail → *add_op term term_tail* | ε

term → *factor factor_tail*

factor_tail → *mult_op factor factor_tail* | ε

factor → `(`*expr*`)` | `id` | `literal`

add_op → `+` | `-`

mult_op → `*` | `/`

Since *factor_tail* can be "transformed" into an empty statement **PREDICT(*factor_tail* → ε)** equals **FOLLOW(*factor_tail*)**

tail) = `{+,-,),id,read,write,$$}`

- PREDICT(*factor_tail* → *m_op factor factor_tail*) = `{*,/}`
- PREDICT(*factor_tail* → ε) = `{+,-,),id,read,write, $$}`

Recursive D

These are all of the PREDICT() values from every production.

```
procedure program()
  case in_tok of
    id, read, write, $$:
      stmt_list()
      match($$)
  else
    return error
```

```
procedure stmt()
  case in_tok of
    id: match(id); match(:=); expr()
    read: match(read); match(id)
    write: match(write); expr()
  else
    return error
```

```
procedure stmt_list()
  case in_tok of
    id, read, write:
      stmt(); stmt_list();
    $$:
      skip
  else
    return error
```

```
procedure match(expec)
    if in_tok = expec
        consume in_tok
    else
        return error
```

Constructing FIRST, FOLLOW, and PREDICT

- To construct the FIRST, FOLLOW, and PREDICT tables we iterate through the grammar building on knowledge.
  - First, we define all of the "**obvious**" FIRST and FOLLOW values
    - For example, $$ ∈ FOLLOW (*stmt_list*) and {`id`, `read`, `write`}∈ FIRST(*stmt*)
  - Next, **we build on this**,
    - For example, {`id`, `read`, `write`}∈ FIRST(*stmt_list*) since *stmt_list* can begin with *stmt* and {`id`, `read`, `write`}∈ FIRST(*stmt*)
  - We then continue on until we get **no more knowledge**.

Let’s try making tables for this grammar

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

factor → `id` | `number` | - *factor* | (*expr*)

add_op → + | -

mult_op → * | /

Table-Driven

```
p_stack: stack of symbols;
p_stack.push(st_symbol);
loop
    exp_sym := p_stack.pop
        if exp_sym = terminal
            match(exp_sym);
            if exp_sym=$$ return
        else
            if table[exp_sym,in_tok].action = error
                return error
            else
                prediction := table[exp_sym,in_tok].prod;
                foreach sym in prod_table[prediction]
                    p_stack.push(sym)
```

Parse Stack	Input Stream
program	`read A read B`
stmt_list $$	`read A read B`
stmt stmt_list $$	`read A read B`
`read id` *stmt_list* $$	`read A read B`
`id` *stmt_list* $$	`A read B`

Writing an LL(1) Grammar--Left Recursion

- **Left recursion** is where the leftmost symbol is a recursive non-terminal symbol.

id_list → *id_list_prefx*;

id_list_prefix → *id_list_prefx*, `id`

id_list_prefix → `id`

- This can cause a grammar **not to be LL(1)**.
- It is **desirable for LR grammars**.

Writing an LL(1) Grammar-- Eliminating Left Recursion

- To eliminate left recursion replace it with **right recursion**.

id_list → *id_list_prefx*;

*id_list*_prefix → *id_list_prefx,* `id`

*id_list*_prefix → `id`

id_list → `id` *id_list_tail*

id_list_tail → , `id` *id_list_tail*

id_list_tail → ;

Writing an LL(1) Grammar--Common Prefix

- **Common prefixes** occur when there is more than one prefix for a given nonterminal.

$$stmt \rightarrow \texttt{id} \ \texttt{:=} \ expr$$

$$stmt \rightarrow \texttt{id} \ \texttt{(}arguments)$$

- Again, this causes a grammar **not to be LL(1)**.

Writing an LL(1) Grammar--Left factoring

- **Common prefixes** To get rid of common prefixes we use a technique called **left factoring**.

stmt → `id` `:=` *expr*

stmt → `id` (*arguments*)

stmt → `id` *stmt_list_tail*

stmt_list_tail → `:=` *expr* | (*arguments*)

Dangling else

- Even if left recursion and common prefixes don't exist a language may not be LL(1).
- In Pascal, there is the problem that an else statement in if-then-else statements is optional. Because **we don't know which if to match else to**.

```
if AAA then
  if BBB then
    CCC
else
  DDD
```

Dangling else

- In Pascal there is **NO LL(1)** parser that can handle this problem.
- Even though a proper LR(1) parser can handle this, it may not handle it in a **method the programmer desires**.

```
if AAA then
  if BBB then
    CCC
else
  DDD
```

Dangling else

- Thus, to write this code correctly (based on indention) “begin” and “end” statements must be added.

```
if AAA then
  if BBB then
    CCC
else
  DDD
```

```
if AAA then
  begin
    if BBB then
      CCC
  end
else
  DDD
```