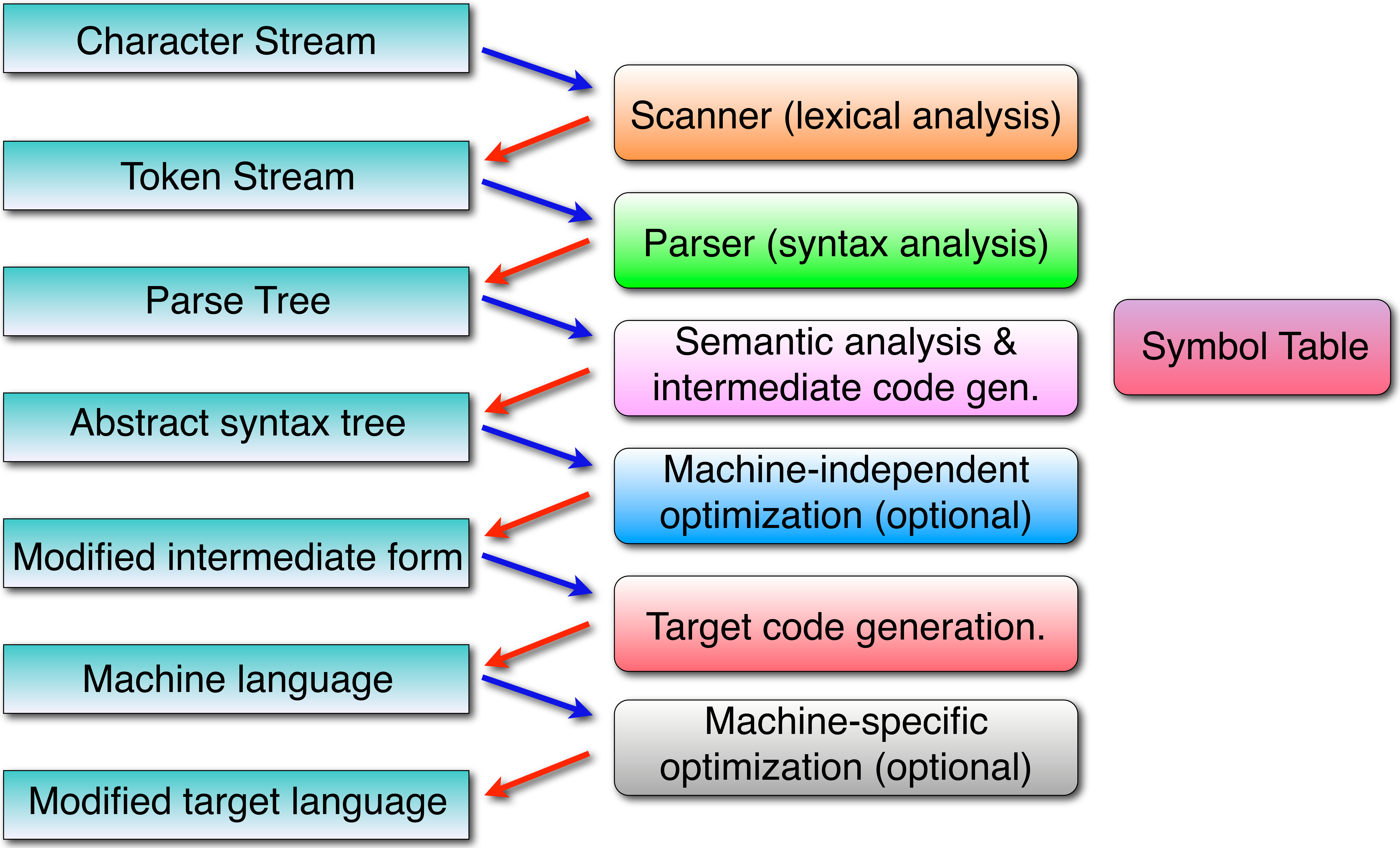
Syntax Analysis

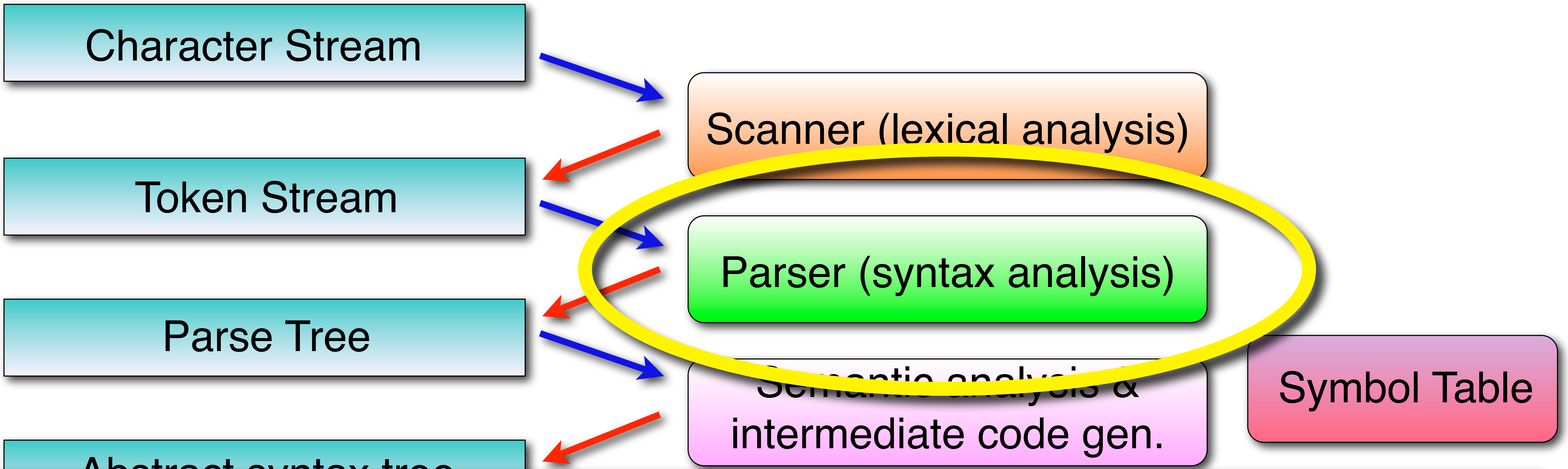
COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

The Big Picture



The Big Picture



Syntax Analysis: **Discovery of Program Structure**

Turn the stream of individual input tokens into a **complete, hierarchical representation** of the program (or compilation unit).

Syntax Specification and Parsing

Syntax Specification

How can we **succinctly** describe the structure of legal programs?

Syntax Recognition

How can a compiler discover if a program **conforms** to the specification?

Context-free Grammars

LL and LR Parsers

Context-Free Grammars

regular grammar + recursion

Review: grammar.

- ➡ Collection of **productions**.
- ➡ A production **defines** a non-terminal (on the left, the “**head**”) in terms of a string terminal and non-terminal symbols.
- ➡ Terminal symbols are elements of the **alphabet** of the grammar.
- ➡ A non-terminal can be the head of **multiple productions**.

Example: Natural Numbers

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

Context-Free Grammars

regular grammar + recursion

Regular grammars.

- ➡ Restriction: **no unrestricted recursion**.
- ➡ A non-terminal symbol cannot be defined in terms of itself. (except for special cases that equivalent to a Kleene Closure)
- ➡ **Serious limitation**: e.g., cannot express matching parenthesis.

Example: Natural Numbers

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

Context-Free Grammars

regular grammar + recursion

Context-free Grammars (CFGs) allow recursion.

Arithmetic expression with parentheses

expr → `id` | `number` | ‘−’ *expr* | ‘(‘ *expr* ‘)’ | *expr op expr*

op → ‘+’ | ‘−’ | ‘*’ | ‘/’

Context-Free Grammars

regular grammar + recursion

Recursion

"An **expression** is a a minus sign followed by an **expression**."

s) allow recursion.

Arithmetic expression with parentheses

expr → id | number | '–' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '–' | '*' | '/'

Context-Free Grammars

regula

Context-free Gr

Can express matching parenthesis requirement.

Arithmetic expression with parentheses

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Context-Free Grammars

regular grammar + recursion

Key difference to lexical grammar:
terminal symbols are tokens,
not individual characters.

...recursion.

Arithmetic expression with parentheses

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Context-Free Grammars

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

on.

Arithmetic expression with parentheses

expr → `id` | `number` | '–' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '–' | '*' | '/'

BNF vs. EBNF

Backus-Naur-Form (BNF)

- ➡Originally developed for ALGOL 58/60 reports.
- ➡Textual notation for context-free grammars.

expr → `id` | `number` | '−' *expr* | '`(`' *expr* '`)`' | *expr op expr*

is written as

```
<expr> ::= id | number | - <expr> |( <expr> ) |
           <expr> <op> <expr>
```

BNF vs. EBNF

Backus-Naur-Form (BNF)

- ➡Originally developed for ALGOL 58/60 reports.
- ➡Textual notation for context-free grammars.

expr → `id` | `number` | '–' *expr* | '(' *expr* ')' | *expr op expr*

is written as

```
<expr> ::= id | number | - <expr> |( <expr> ) |
           <expr> <op> <expr>
```

Strictly speaking, it does not include the Kleene Star and similar "notational sugar."

BNF vs. EBNF

Extended Backus-Naur-Form (EBNF)

- ➡ Many authors **extend** BNF to simplify grammars.
- ➡ One of the first to do so was Niklaus Wirth.
- ➡ There exists an **ISO standard for EBNF** (ISO/IEC 14977).
- ➡ **But many dialects exist.**

BNF vs. EBNF

Extended Backus-Naur-Form (EBNF)

- Many authors **extend** BNF to simplify grammars.
- One of the first to do so was Niklaus Wirth.
- There exists an **ISO standard for EBNF** (ISO/IEC 14977).
- **But many dialects exist.**

Features

- Terminal symbols are quoted.
- Use of '=' instead of '::=' to denote →.
- Use of ',' for concatenation.
- **[*A*]** means *A* can occur **optionally** (zero or one time).
- **{*A*}** means *A* can occur repeatedly (**Kleene Star**).
- Parenthesis are allowed for grouping.
- And then some…

BNF vs. EBNF

Extended Backus-Naur-Form (EBNF)

- Many authors **extend** BNF to simplify grammars.

> We will use mostly BNF-like grammars with the addition of the **Kleene Star**, **ε**, and **parenthesis**.

- Terminal symbols are quoted.
- Use of '=' instead of '::=' to denote →.
- Use of ',' for concatenation.
- **[*A*]** means *A* can occur **optionally** (zero or one time).
- **{*A*}** means *A* can occur repeatedly (**Kleene Star**).
- Parenthesis are allowed for grouping.
- And then some…

Example: EBNF to BNF Conversion

id_list → `id` (, `id`)*

is equivalent to

id_list → `id`

id_list → *id_list*, `id`

(Remember that non-terminals can be the head of multiple productions.)

Derivation

A grammar allows programs to be derived.

- ➡Productions are **rewriting rules**.
- ➡A program is **syntactically correct** if and only if it can be derived from the start symbol.

Derivation Process

- ➡Begin with string consisting only of **start symbol**.

```
while string contains a non-terminal symbol:
    Choose one non-terminal symbol X.
    Choose production where X is the head.
    Replace X with right-hand side of production.
```


Wednesday, April 14, 2010

Derivation

If we **always** choose the **left-most non-terminal symbol**, then it is called a **left-most derivation**.

ived.

d only if it can be

Derivation Process

➡Begin with string consisting only of **start symbol**.

```
while string contains a non-terminal symbol:
    Choose one non-terminal symbol X.
    Choose production where X is the head.
    Replace X with right-hand side of production.
```

Derivation

A grammar allows programs to be derived.

➡Productions are **rewriting rules**.

➡A program is **sy**
be derived from

Derivation Proce

➡Begin with strin

If we **always** choose the **right-most non-terminal symbol**, then it is called a **right-most** or **canonical** derivation.

```
while string contains a non-terminal symbol:
    Choose one non-terminal symbol X.
    Choose production where X is the head.
    Replace X with right-hand side of production.
```

Example Derivation

Arithmetic grammar:

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

op → '+' | '-' | '*' | '/'

Program

slope * x + intercept

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Program

slope * x + intercept

expr ⇒ *expr op expr*

Example Derivation

Arithmetic grammar:

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

op → '`+`' | '`-`' | '`*`' | '`/`'

<u>Program</u>

slope * x + intercept

⇒ denotes "derived from"

expr ⇒ *expr op expr*

Example Derivation

Arithmetic grammar:

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

op → '+' | '-' | '*' | '/'

Program

slope * x + intercept

expr ⇒ *expr op* ***expr***

⇒ *expr op* `id`

Example Derivation

Arithmetic grammar:

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

op → '+' | '-' | '*' | '/'

Program

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr* ***op*** `id`

⇒ *expr* + `id`

Wednesday, April 14, 2010

Example Derivation

Arithmetic grammar:

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

op → '+' | '-' | '*' | '/'

Program

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* `id`

⇒ ***expr*** + `id`

⇒ *expr op expr* + `id`

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*
op → '+' | '−' | '*' | '/'

Program
slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* `id`

⇒ *expr* + `id`

⇒ *expr op* ***expr*** + `id`

⇒ *expr op* `id` + `id`

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Program

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* `id`

⇒ *expr* + `id`

⇒ *expr op expr* + `id`

⇒ *expr* ***op*** `id` + `id`

⇒ *expr* * `id` + `id`

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Program

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`

Wednesday, April 14, 2010

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

Substitute **values** of identifier tokens.

<u>Program</u>

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`

Wednesday, April 14, 2010

Example Derivation

Arithmetic grammar:

expr → `id` | `number` | '−' *expr* | '(' *expr* ')' | *expr op expr*

op → '+' | '−' | '*' | '/'

This is a **right-most** derivation.

Program

slope * x + intercept

expr ⇒ *expr op expr*

⇒ *expr op* `id`

⇒ *expr* + `id`

⇒ *expr op expr* + `id`

⇒ *expr op* `id` + `id`

⇒ *expr* * `id` + `id`

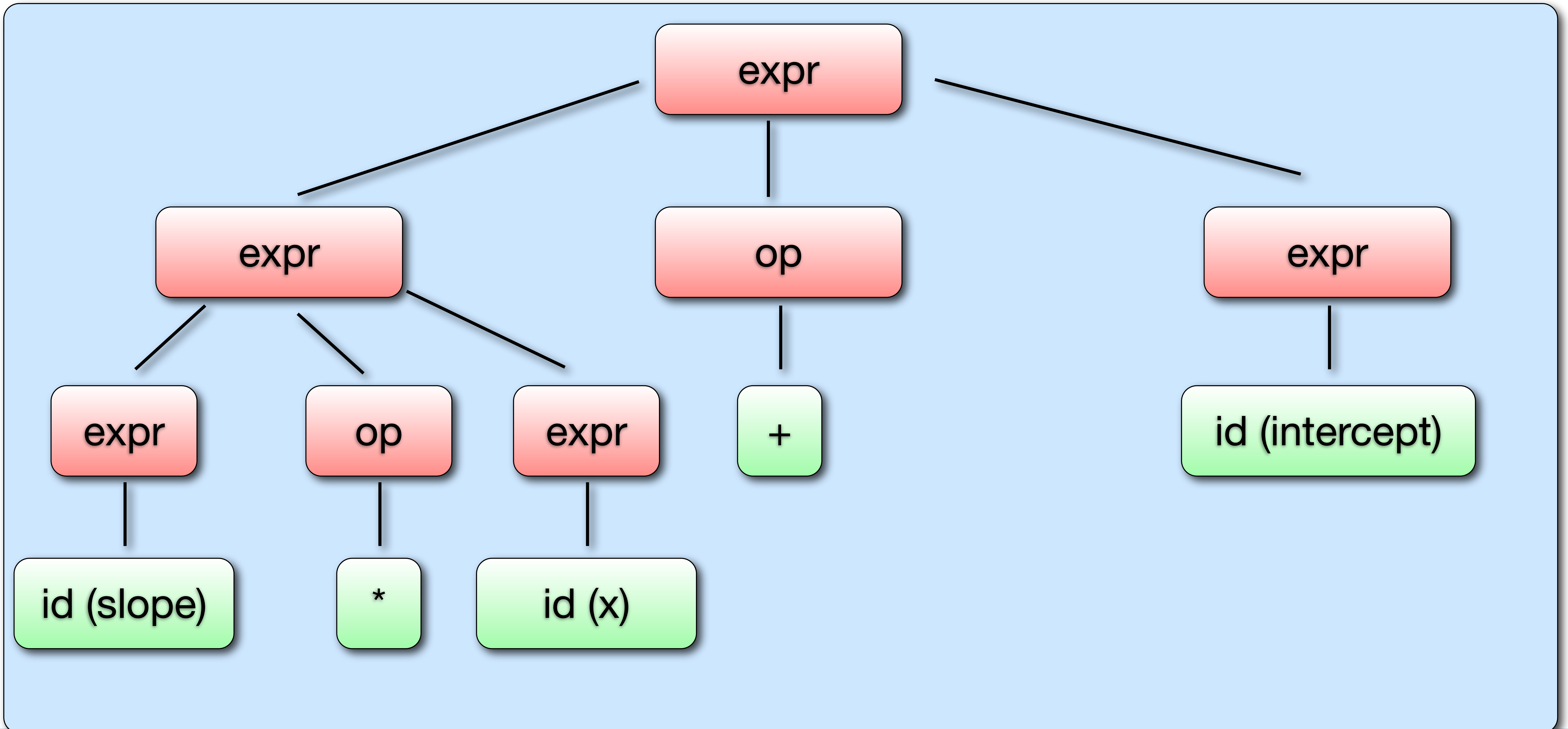
⇒ `id` * `id` + `id`

`slope * x + intercept`

Wednesday, April 14, 2010

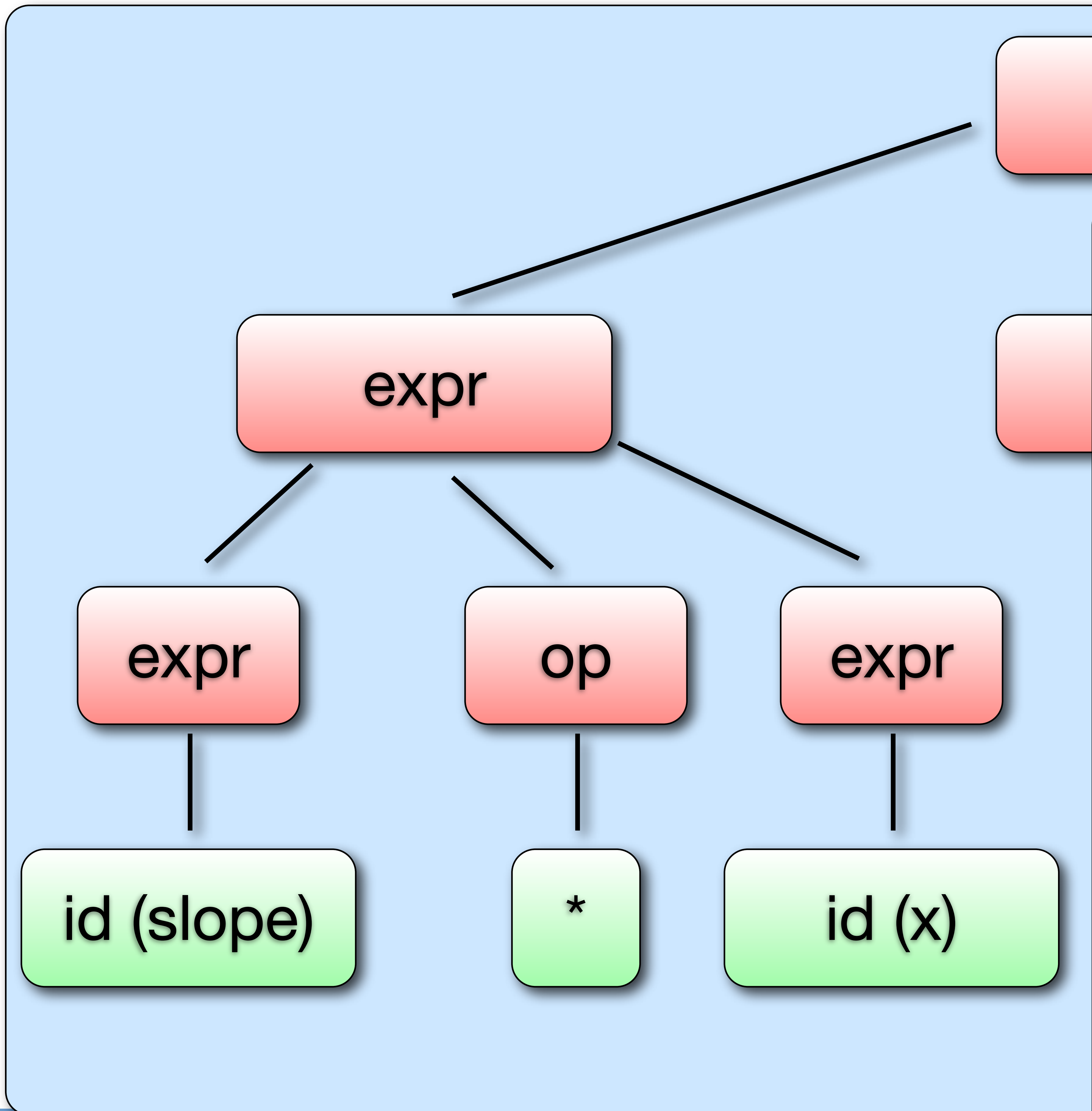
Parse Tree

A parse tree is a **hierarchical representation** of the derivation that does not show the derivation order.



Parse Tree

A parse tree is a **hierarchical representation** of the derivation that does not show the derivation order.

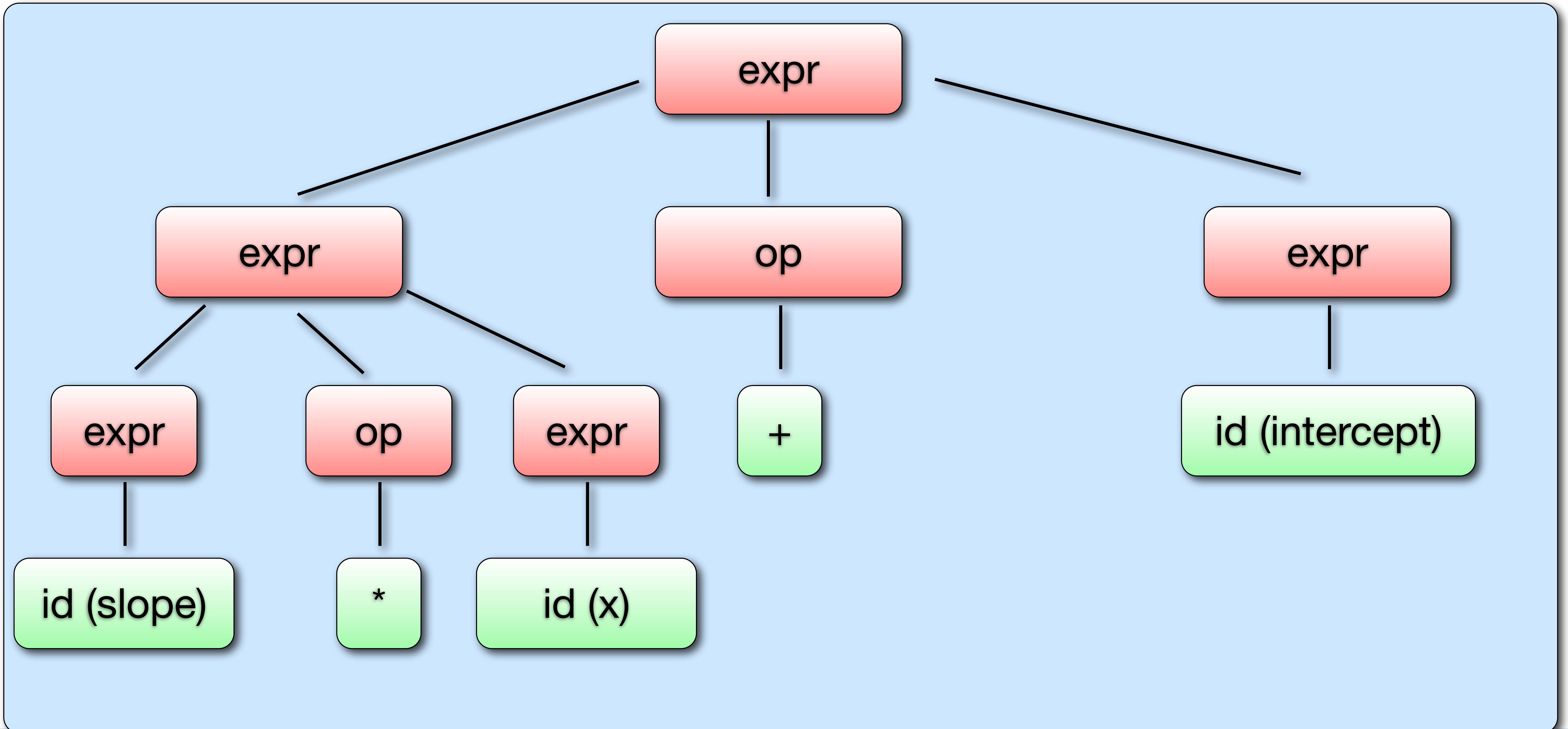


Properties

- ➡ Each **interior node** is a non-terminal symbol.
- ➡ Its **children** are the right-hand side of the production that it was replaced with.
- ➡ **Leaf nodes** are terminal symbols (tokens).
- ➡ **Many-to-one**: many derivations can yield identical parse trees.
- ➡ The parse tree **defines the structure** of the program.

Parse Tree

This parse tree represents the formula
`slope * x + intercept`.



Wednesday, April 14, 2010

Parse Tree

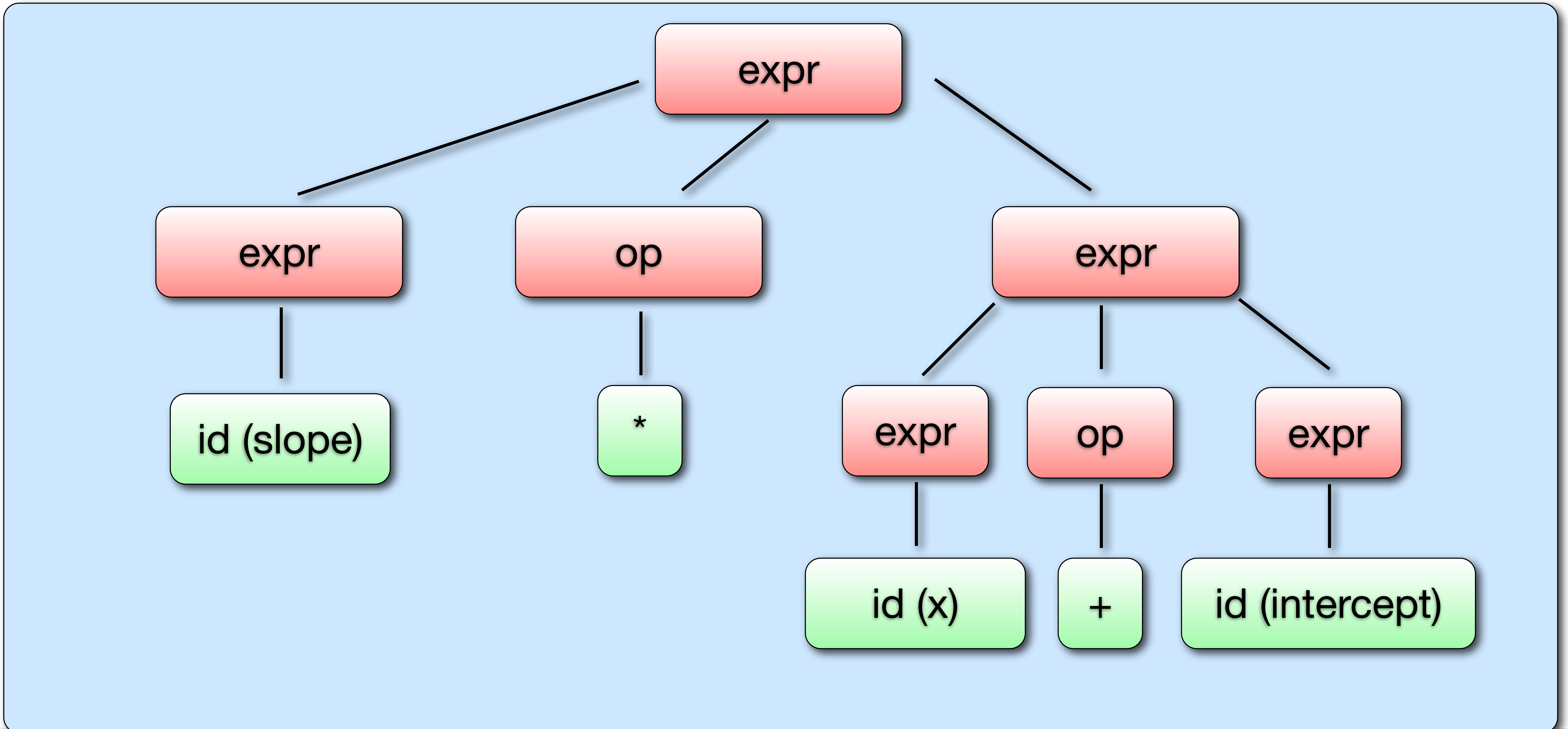
Let's do a **left-most derivation** of
`slope * x + intercept`.

Arithmetic grammar:

expr → `id` | `number` | '`-`' *expr* | '`(`' *expr* '`)`' | *expr op expr*
op → '`+`' | '`-`' | '`*`' | '`/`'

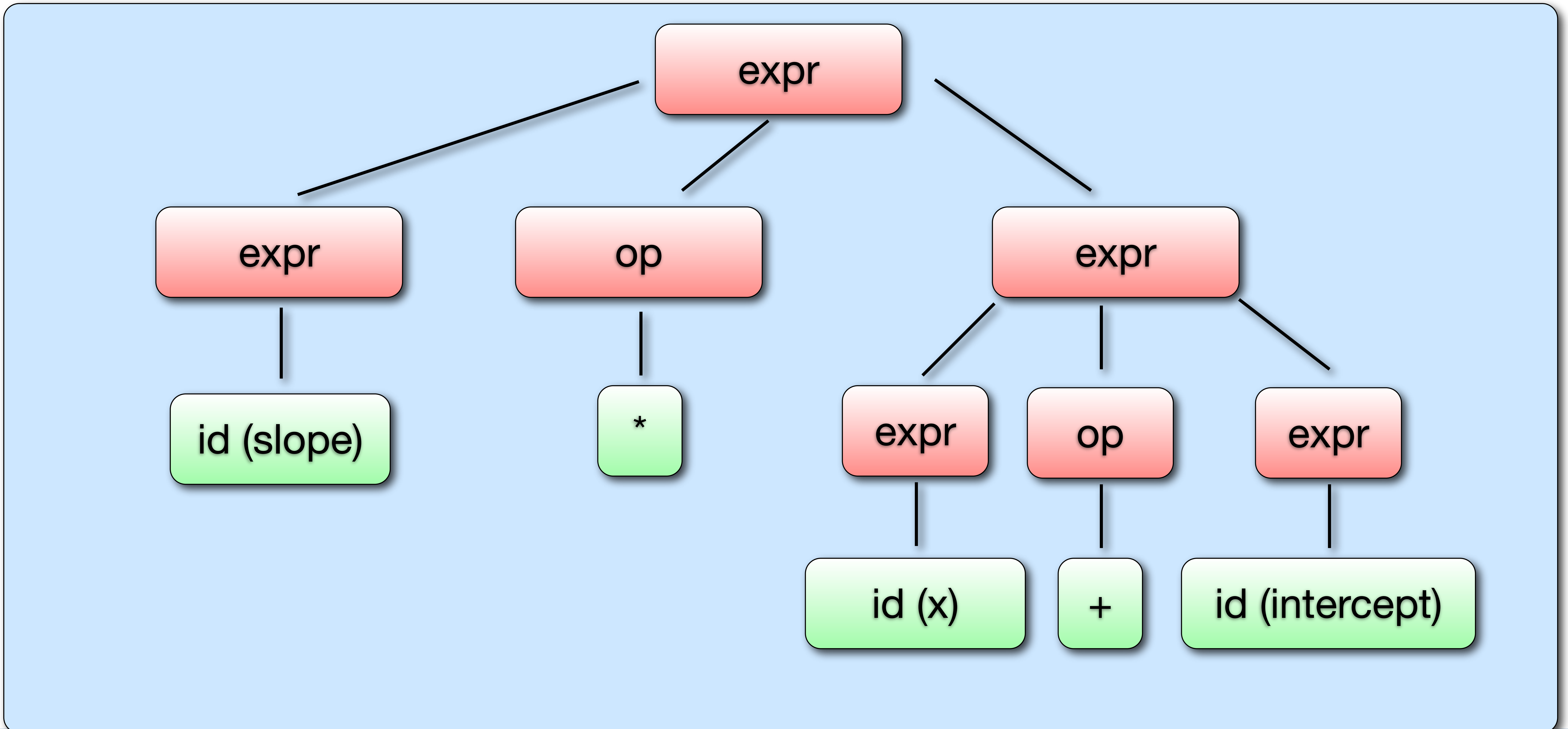
Parse Tree

Let's do a **left-most derivation** of
`slope * x + intercept.`



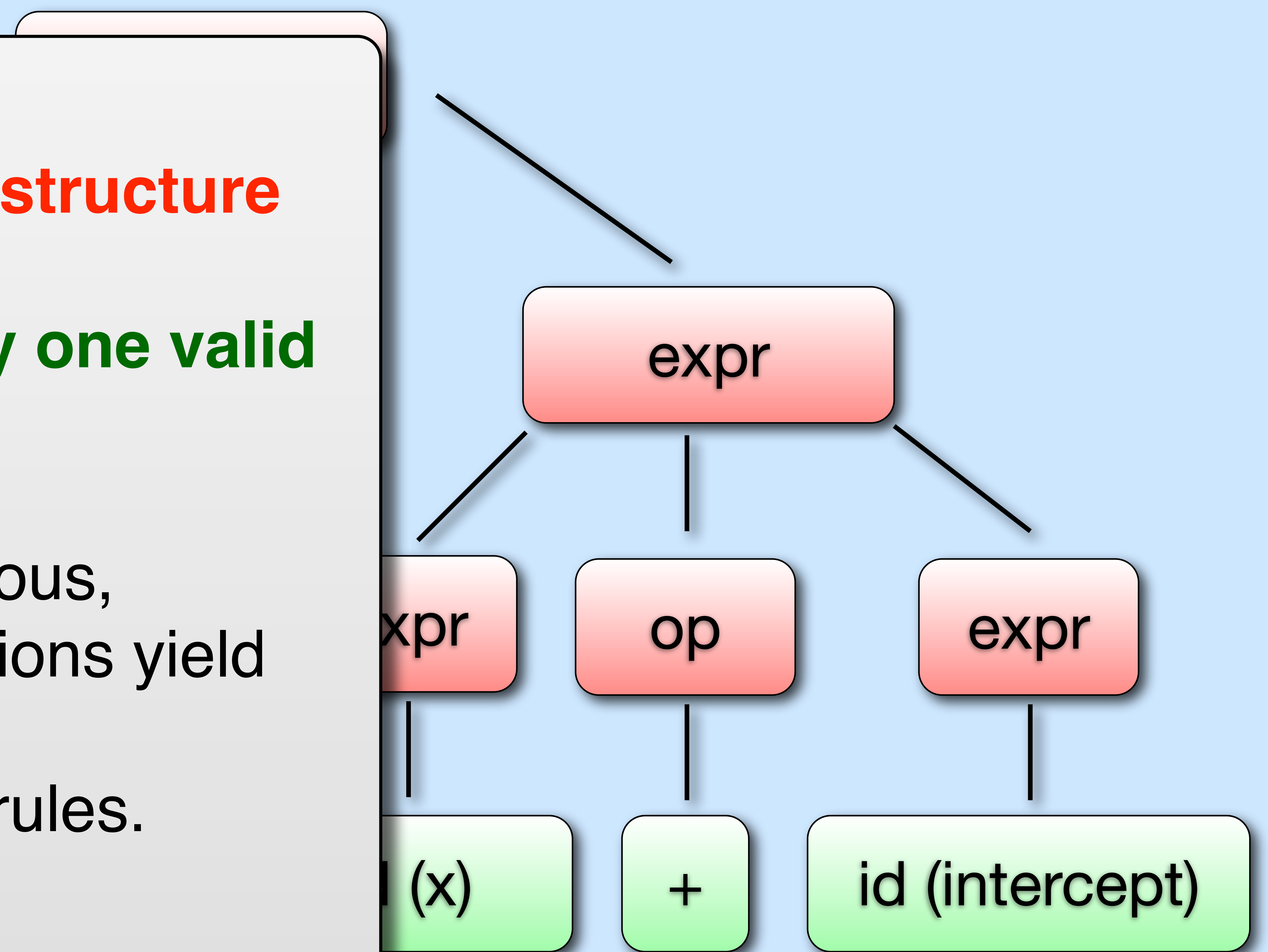
Parse Tree (Ambiguous)

This parse tree represents the formula **`slope * (x + intercept)`**, which is **not equal** to **`slope * x + intercept.`**



Parse Tree (Ambiguous)

This parse tree represents the formula **`slope * (x + intercept)`**, which is **not equal** to **`slope * x + intercept.`**



Ambiguity

- The parse tree **defines the structure** of the program.
- A program should have **only one valid interpretation**!
- Two solutions:
  - Make grammar unambiguous, i.e., ensure that all derivations yield **identical** parse trees.
  - Provide **disambiguating** rules.

Disambiguating the Grammar

- The problem with our original grammar is that it does **not fully express the grammatical structure** (i.e., associativity and precedence).
- To create an unambiguous grammar, we need to fully specify the grammar and differentiate between **terms** and **factors**.

Disambiguating the Grammar

- The problem with our original grammar is that it does **not fully express the grammatical structure** (i.e., associativity and precedence).
- To create an unambiguous grammar, we need to fully specify the grammar and differentiate between **terms** and **factors**.

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

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

add_op → + | −

mult_op → * | /

Disambiguating the Grammar

- The problem with our original grammar is that it does **not fully express the grammatical structure** (i.e., associativity and preced
- To crea … ecify the grammar and differentiate between **terms** and **factors**.

This gives **precedence** to multiply.

expr → *term* | *expr add_op term*

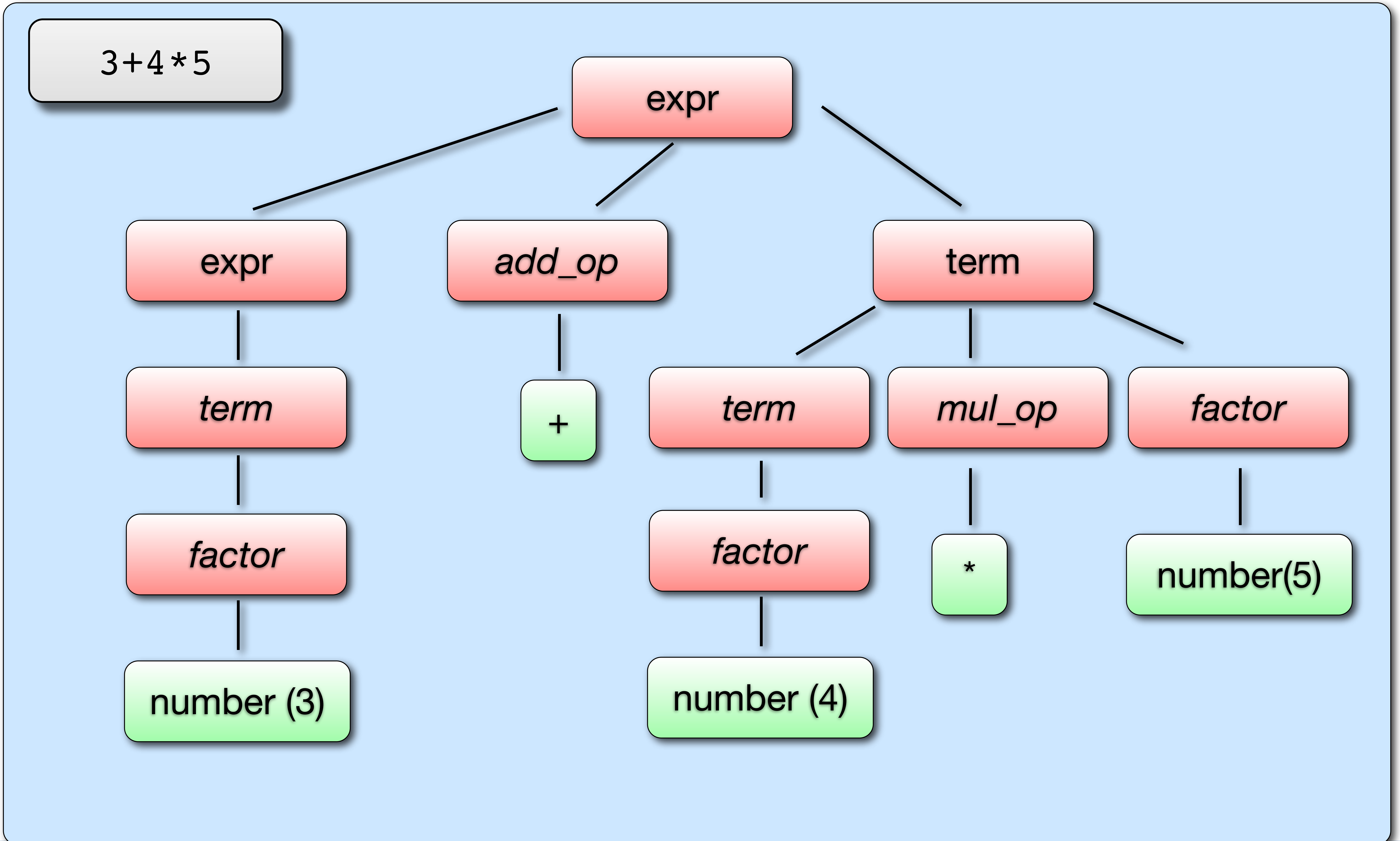
term → *factor* | *term mult_op factor*

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

add_op → + | −

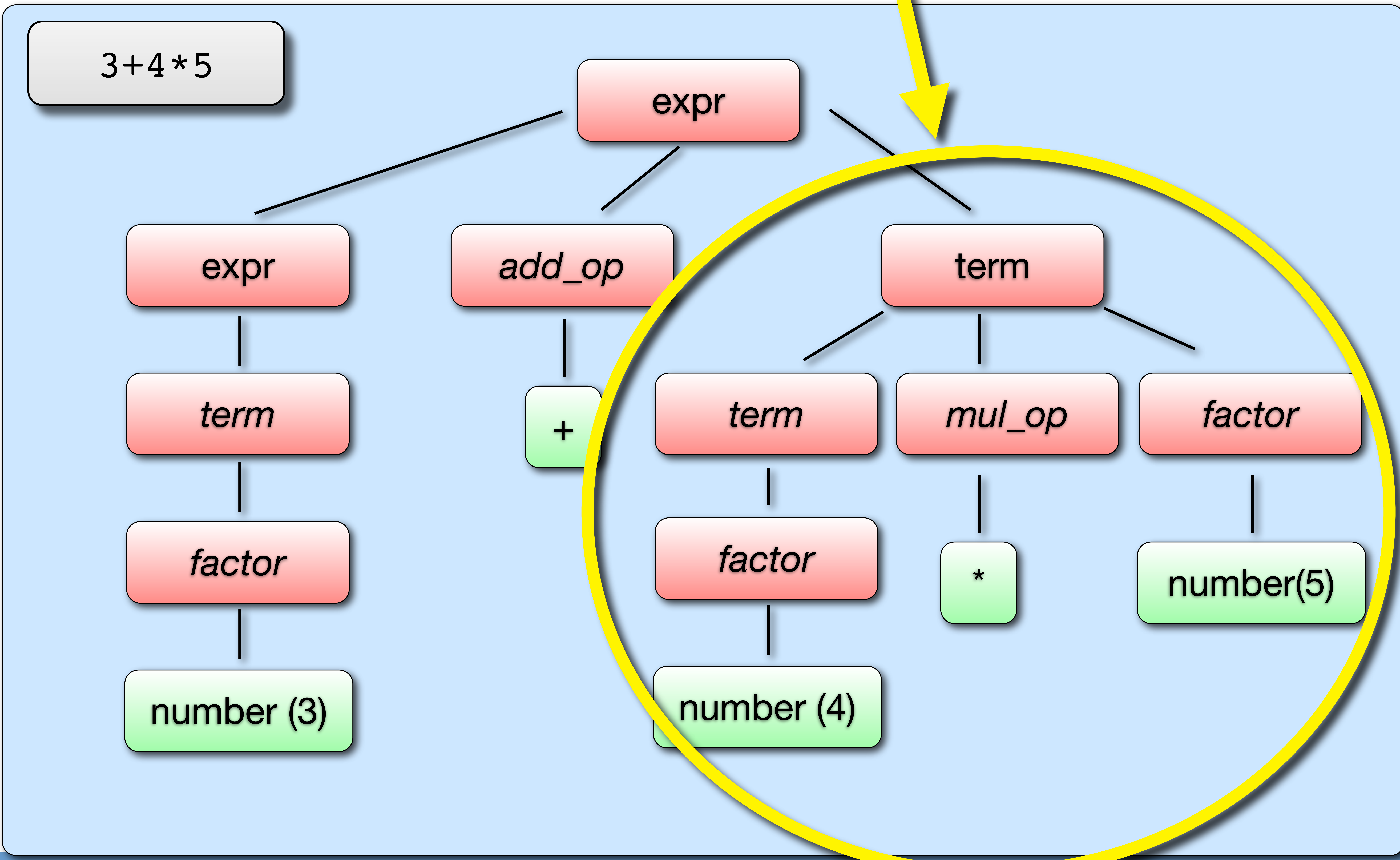
mult_op → * | /

Example Parse Tree



Ex

Multiplication precedes addition.



Another Example

Lets try deriving “3*4+5*6+7”.

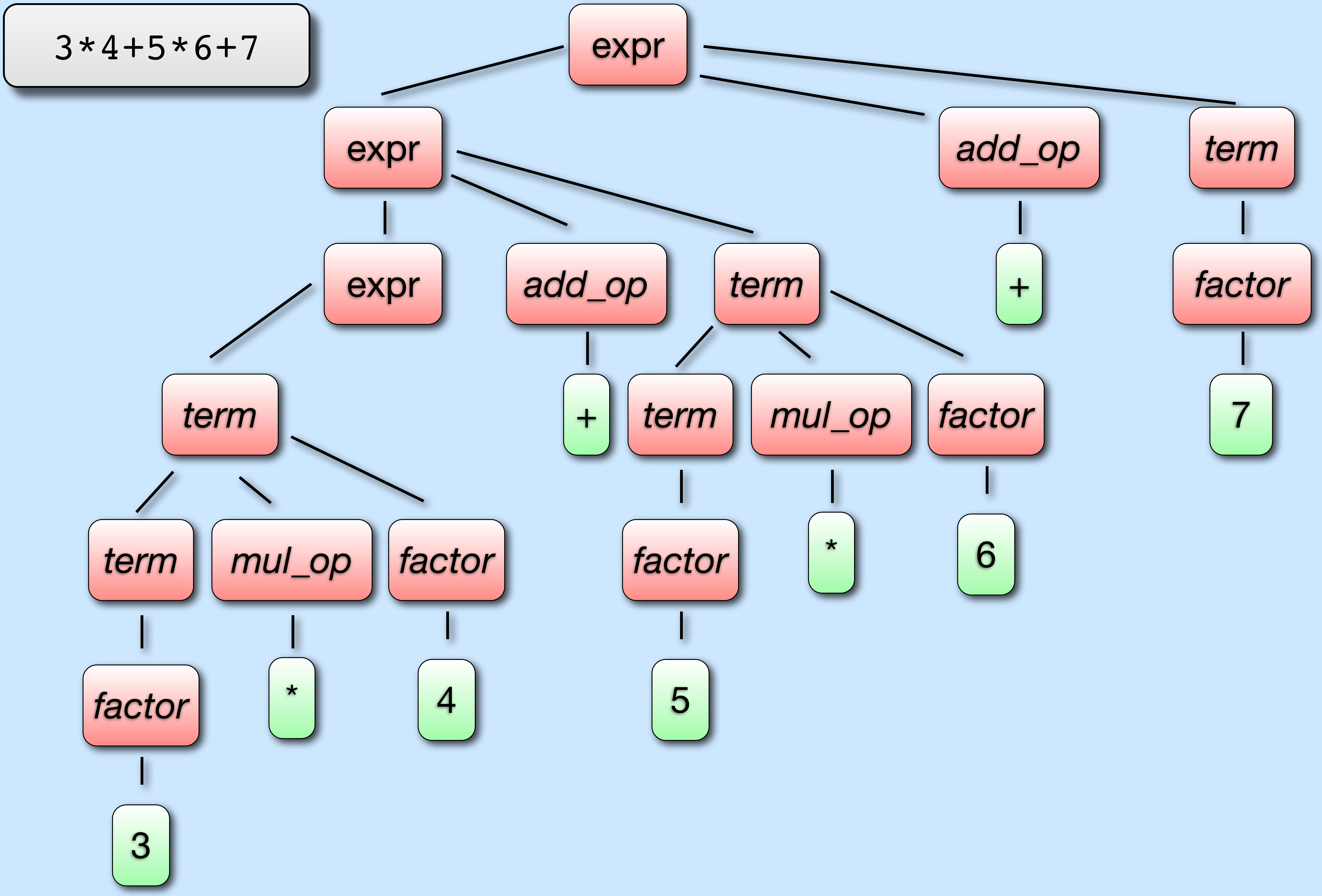
expr → *term* | *expr add_op term*

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

Parser

The purpose of the parser is to
construct the parse tree
that
corresponds to the input token stream.

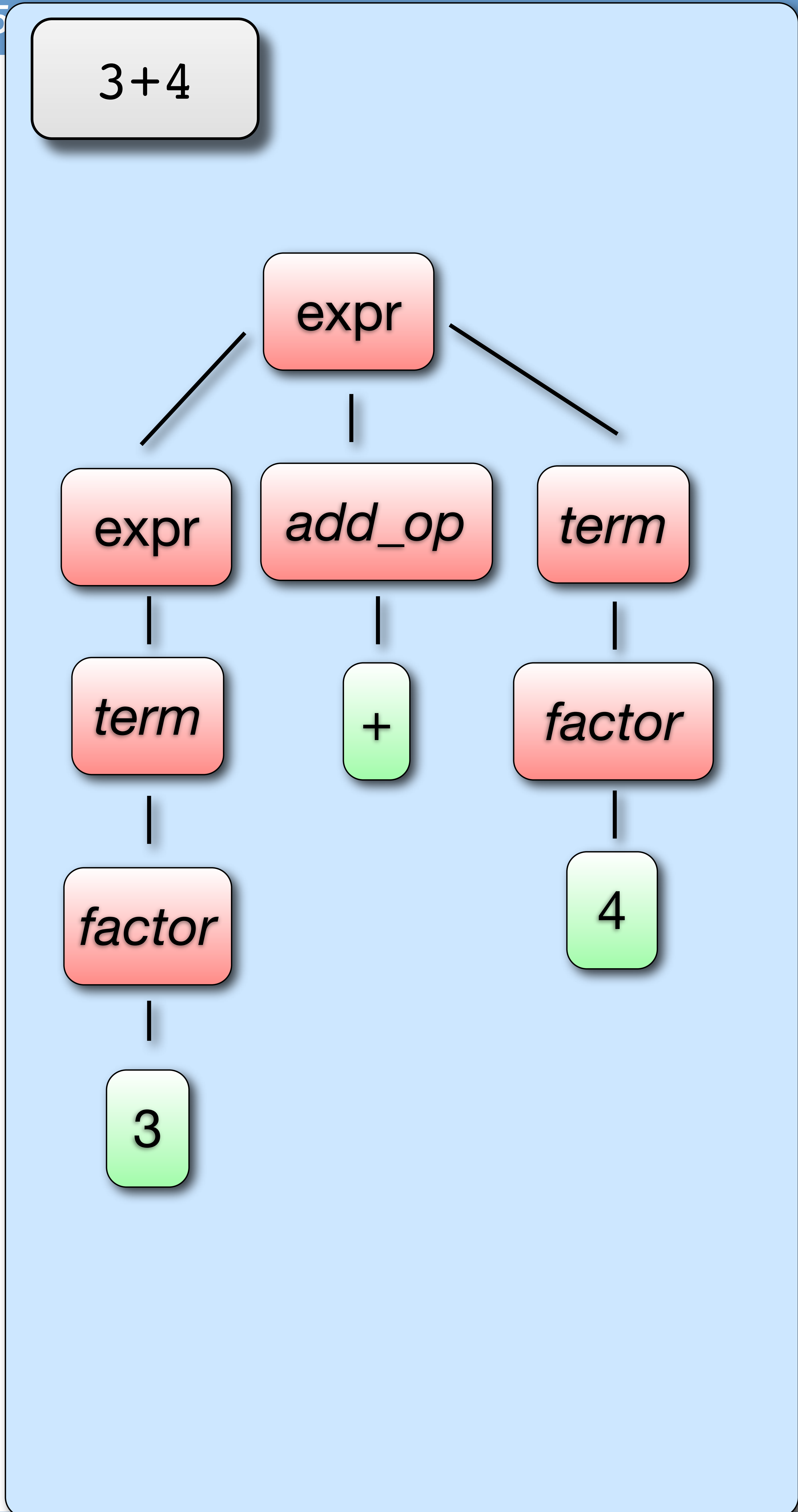
(If such a tree exists, i.e., for correct input.)

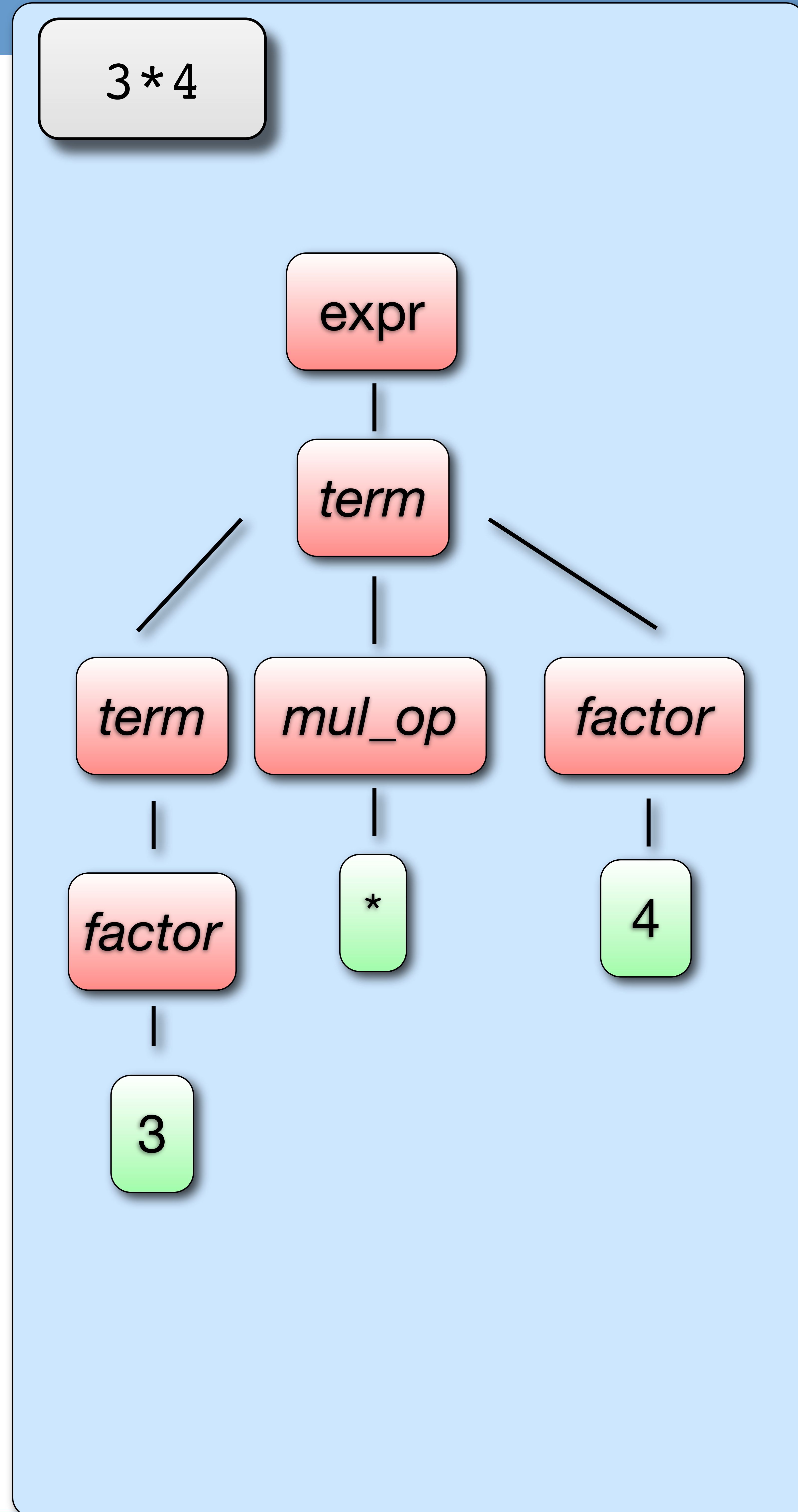
Parser

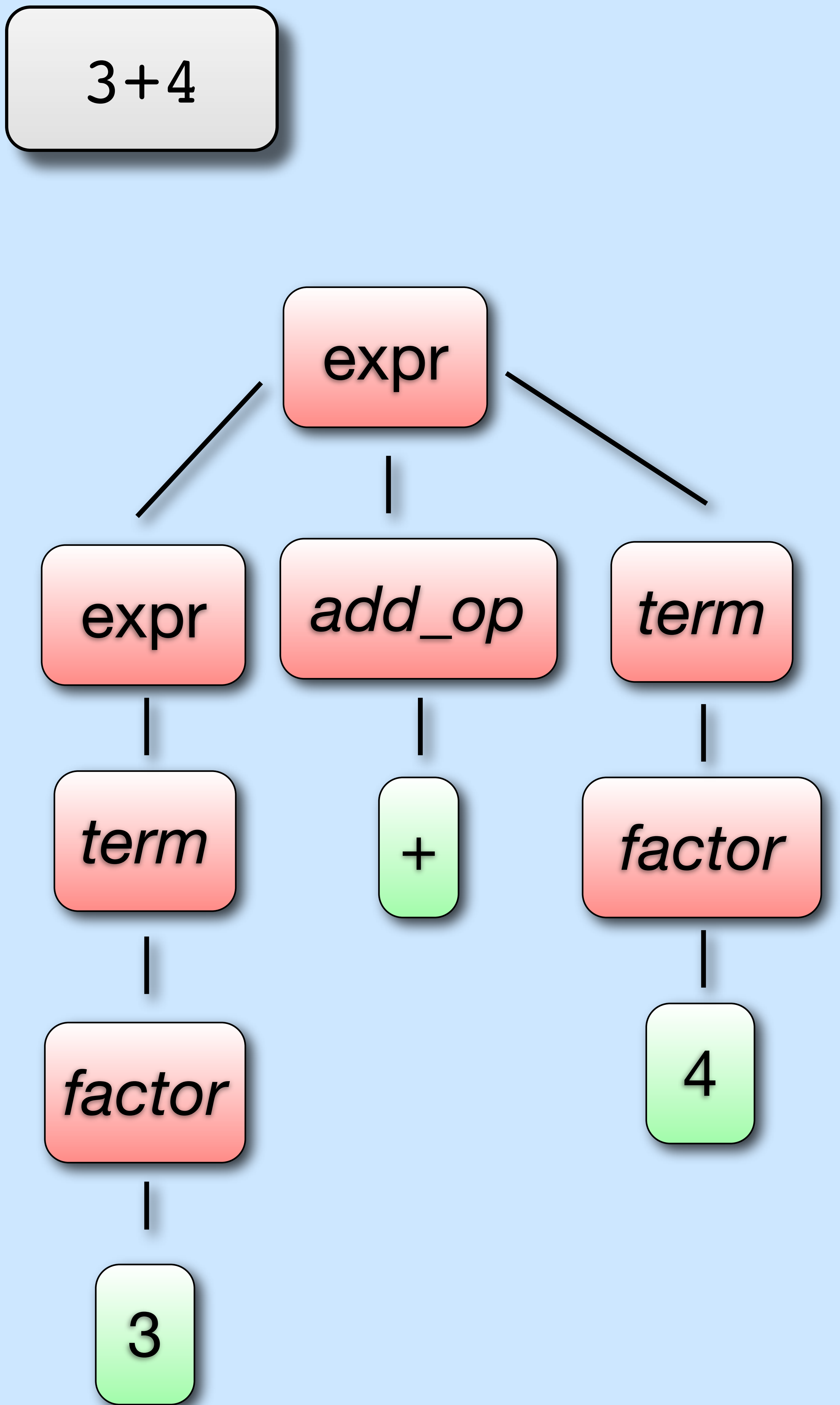
The purpose of the parser is to **construct the parse tree** that **corresponds to the input token stream**.

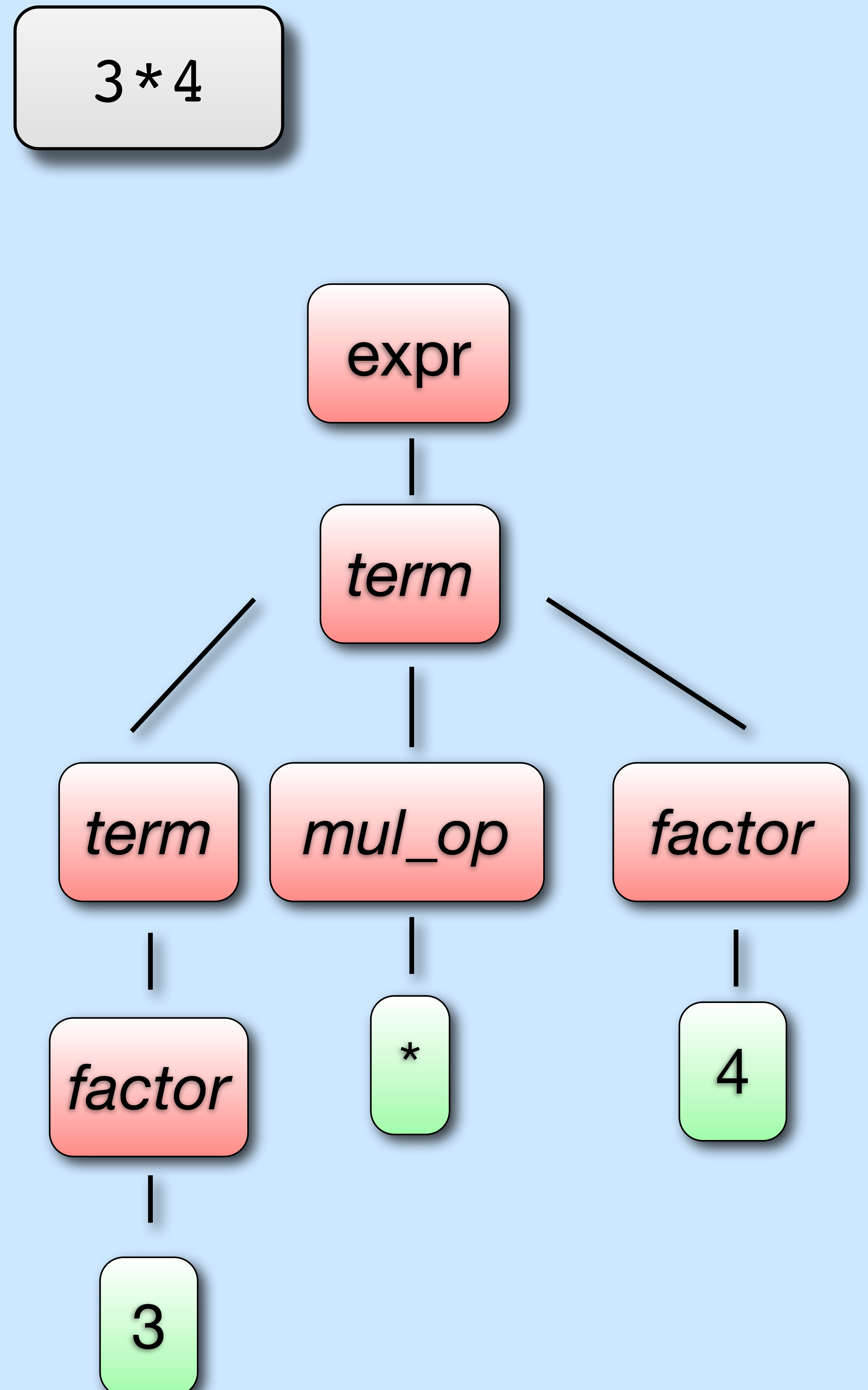
(If such a tree exists, i.e., for correct input.)

This is a **non-trivial problem**:
for example, consider "`3*4`" and "`3+4`".

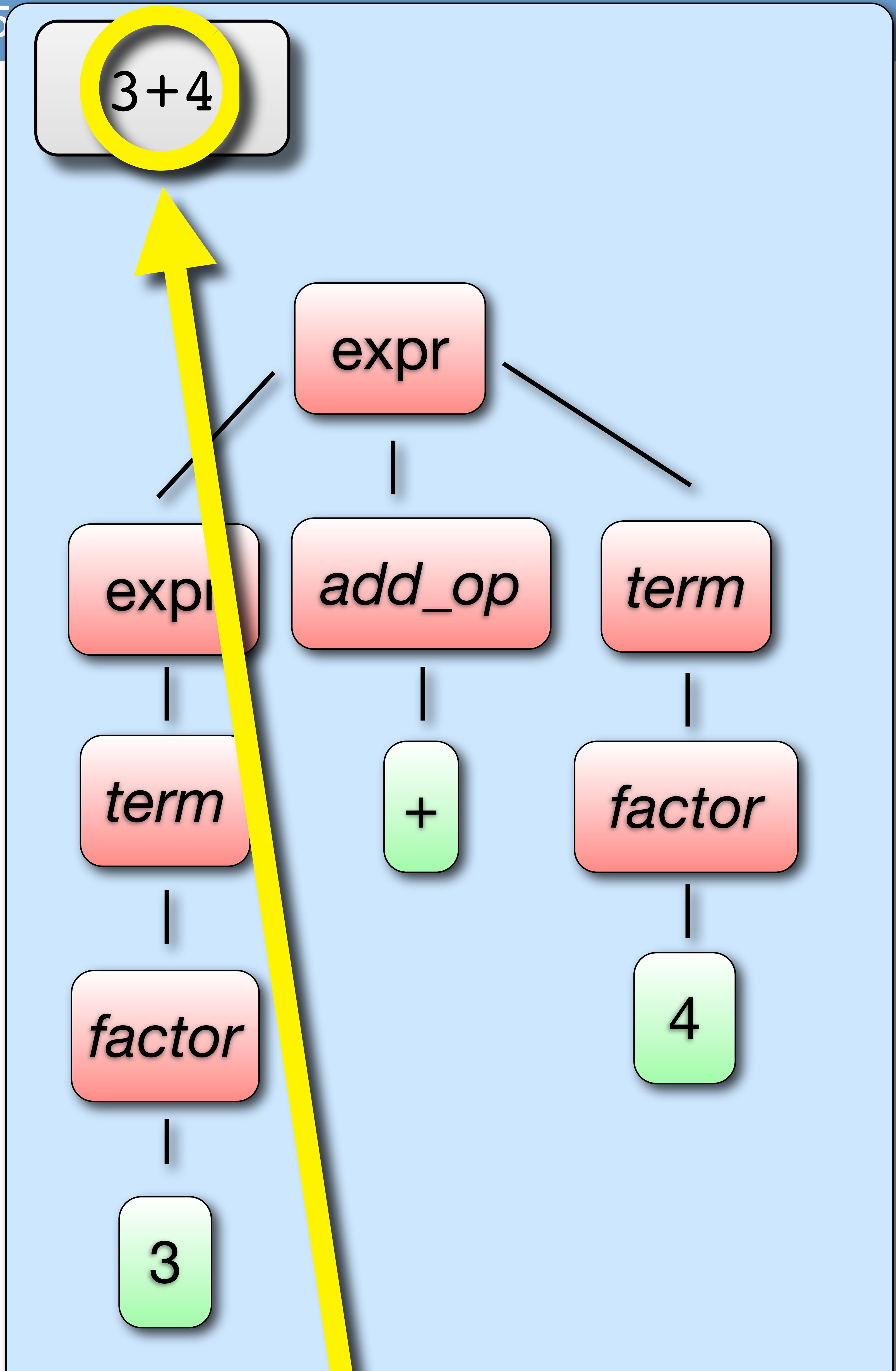

3+4
expr
expr
add_op
term
term
+
factor
factor
4
3

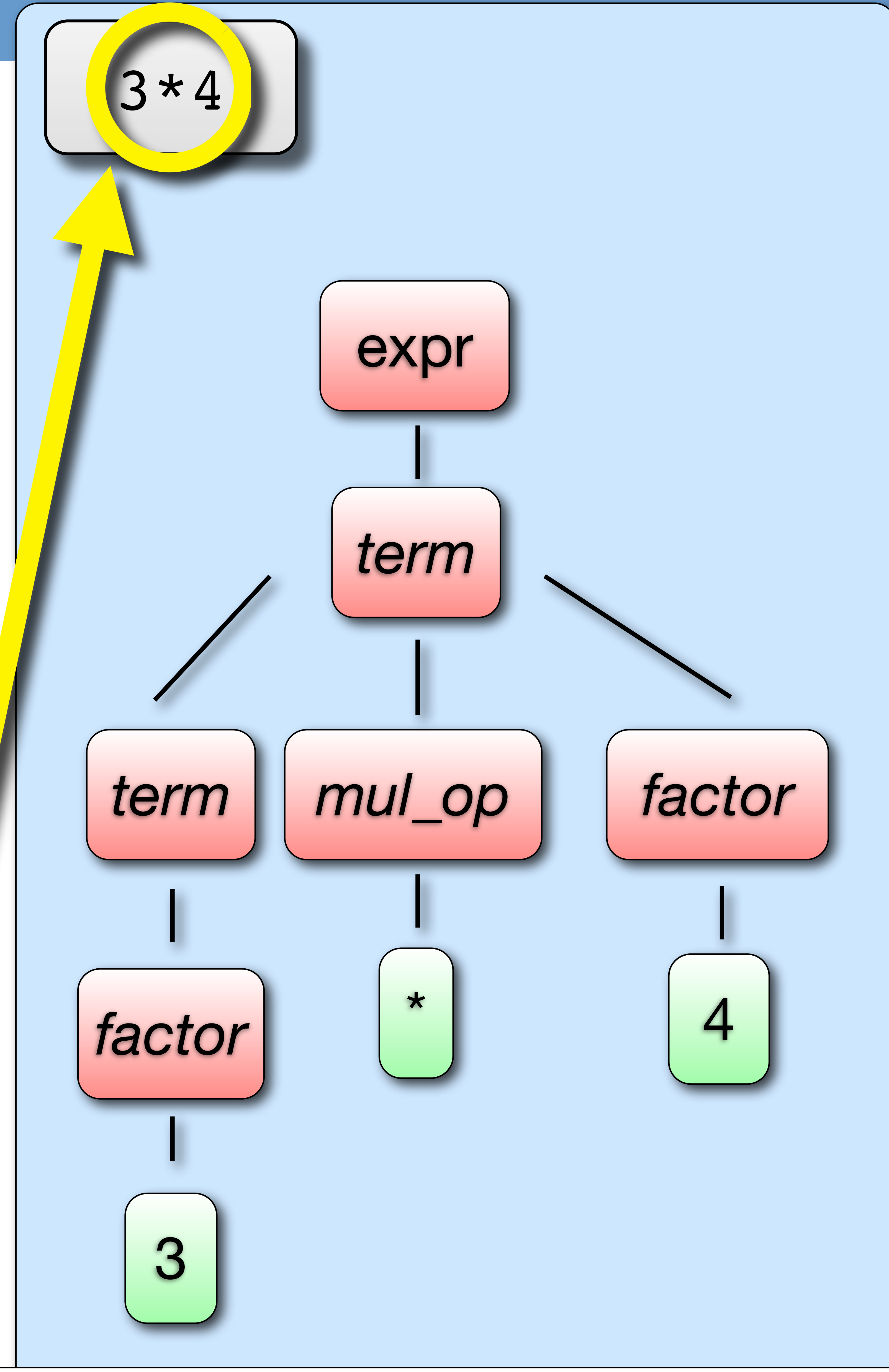

3*4
expr
term
term
mul_op
factor
factor
*
4
3



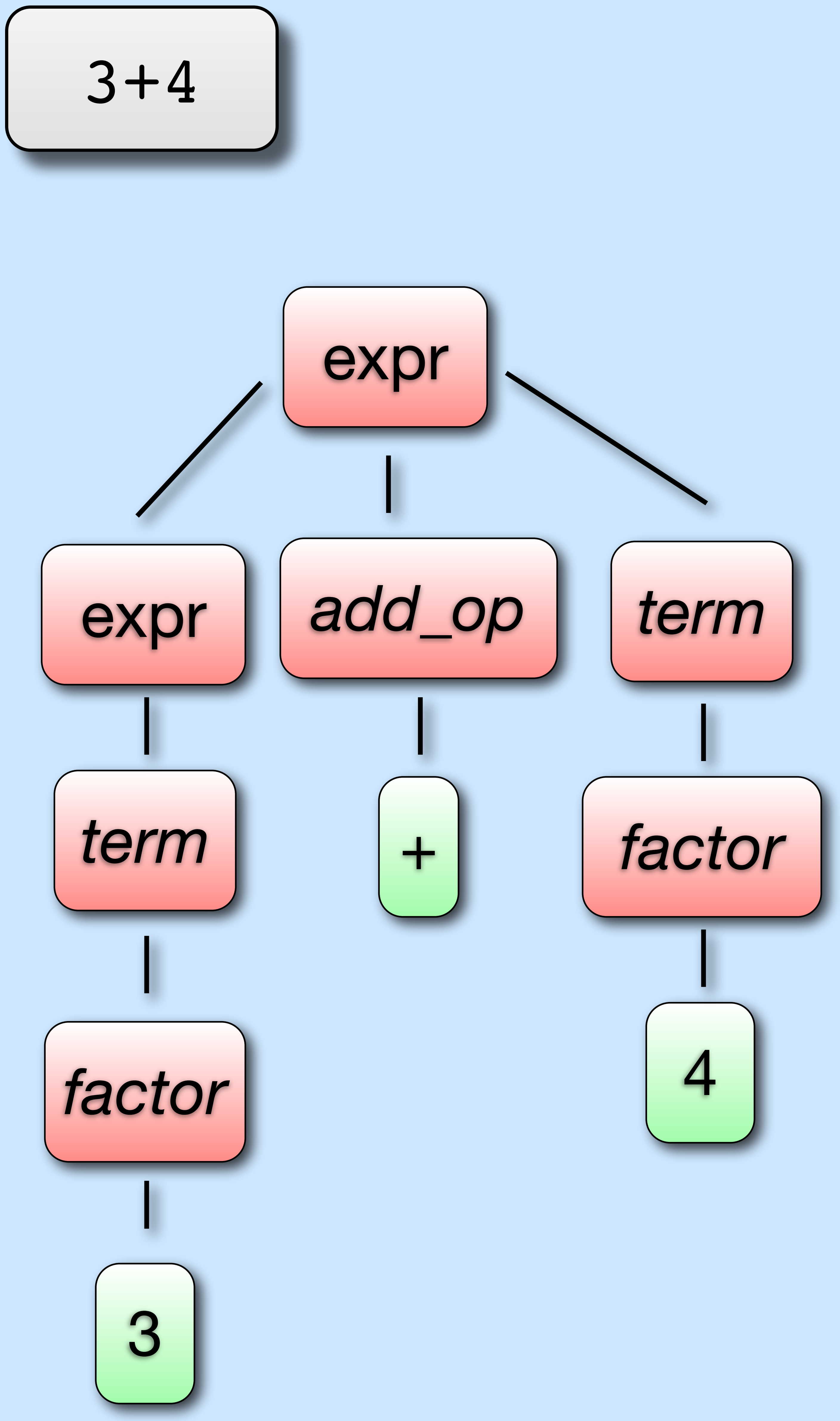


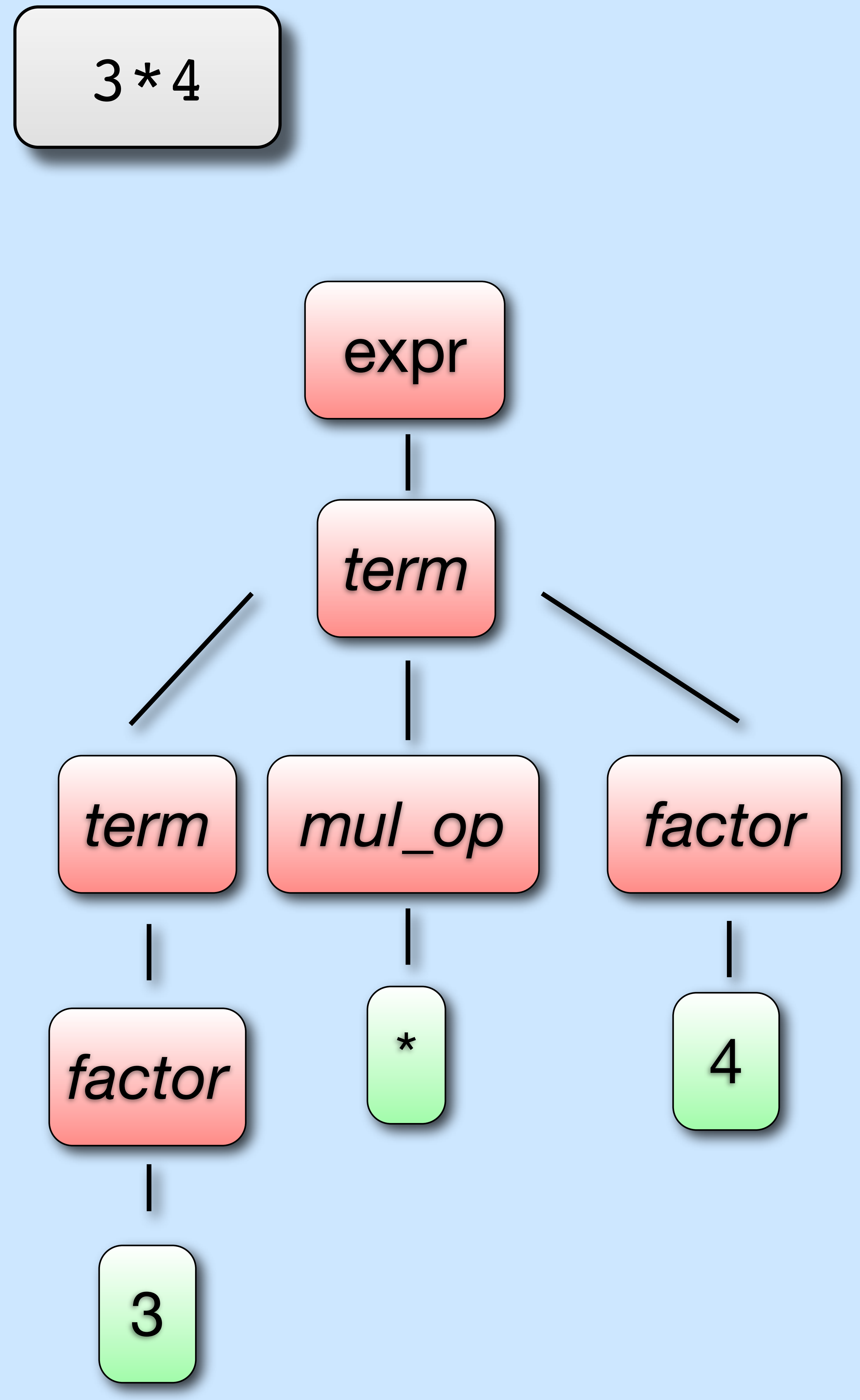
How can a computer derive these trees by examining **one token at a time**?





In order to derive these trees, the **first character** that we need to examine is the math operator **in the middle**.





Writing **ad-hoc parsers** is difficult, tedious, and error-prone.

Complexity of Parsing

Arbitrary CFGs can be parsed in $O(n^3)$ time.

- **n** is length of the program (in tokens).
- Earley's algorithm.
- Cocke-Younger-Kasami (CYK) algorithm.
- This is **too inefficient** for most purposes.

Efficient parsing is possible.

- There are (restricted) types of grammars that can be parsed in **linear time**, i.e., **O(n)**.
- Two important classes:
  - **LL**: "Left-to-right, Left-most derivation"
  - **LR**: "Left-to-right, Right-most derivation"
- These are **sufficient** to express most programming languages.

Complexity of Parsing

The class of all grammars for which a **left-most** derivation always yields a parse tree.

➡ This is **too inefficient** for most purposes.

Efficient parsing is possible.

➡ There are (restricted) types of grammars that can be parsed in **linear time**, i.e., **O(n)**.

➡ Two important classes:

- **LL**: "Left-to-right, Left-most derivation"
- **LR**: "Left-to-right, Right-most derivation"

➡ These are **sufficient** to express most programming languages.

Complexity of Parsing

Arbitrary CFGs can be parsed in $O(n^3)$ time.

- **n** is length of the program (in tokens).

The class of all grammars for which a **right-most** derivation always yields a parse tree.

Efficient parsing is possible.

- There are (restricted) types of grammars that can be parsed in **linear time**, i.e., **$O(n)$**.
- Two important classes:
  - **LL**: "Left-to-right, Left-most derivation"
  - **LR**: "Left-to-right, Right-most derivation"
- These are **sufficient** to express most programming languages.

LL-Parsers vs. LR-Parsers

LL-Parsers

- ➡ Find **left-most** derivation.
- ➡ Create parse-tree in **top-down** order, **beginning at the root**.
- ➡ Can be either **constructed manually** or automatically generated with tools.
- ➡ Easy to understand.
- ➡ LL grammars sometimes appear "**unnatural**."
- ➡ Also called **predictive** parsers.

LR-Parsers

- ➡ Find **right-most** derivation.
- ➡ Create parse-tree in **bottom-up** order, **beginning at leaves**.
- ➡ Are usually **generated by tools**.
- ➡ Operating is less intuitive.
- ➡ LR grammars are often "**natural**."
- ➡ Also called **shift-reduce** parsers.
- ➡ **Strictly more expressive**: every LL grammar is also an LR grammar, but the converse is not true.

Both are used in practice.
We focus on LL.

LL vs. LR Example

A simple grammar for a list of identifiers.

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → ;

Input

```
A, B, C;
```

LL Example

id_list → `id` *id_list_tail*

id_list_tail → , `id` *id_list_tail*

id_list_tail → ;

the
(as of yet empty)
parse tree

`A , B , C ;`

current token

LL Example

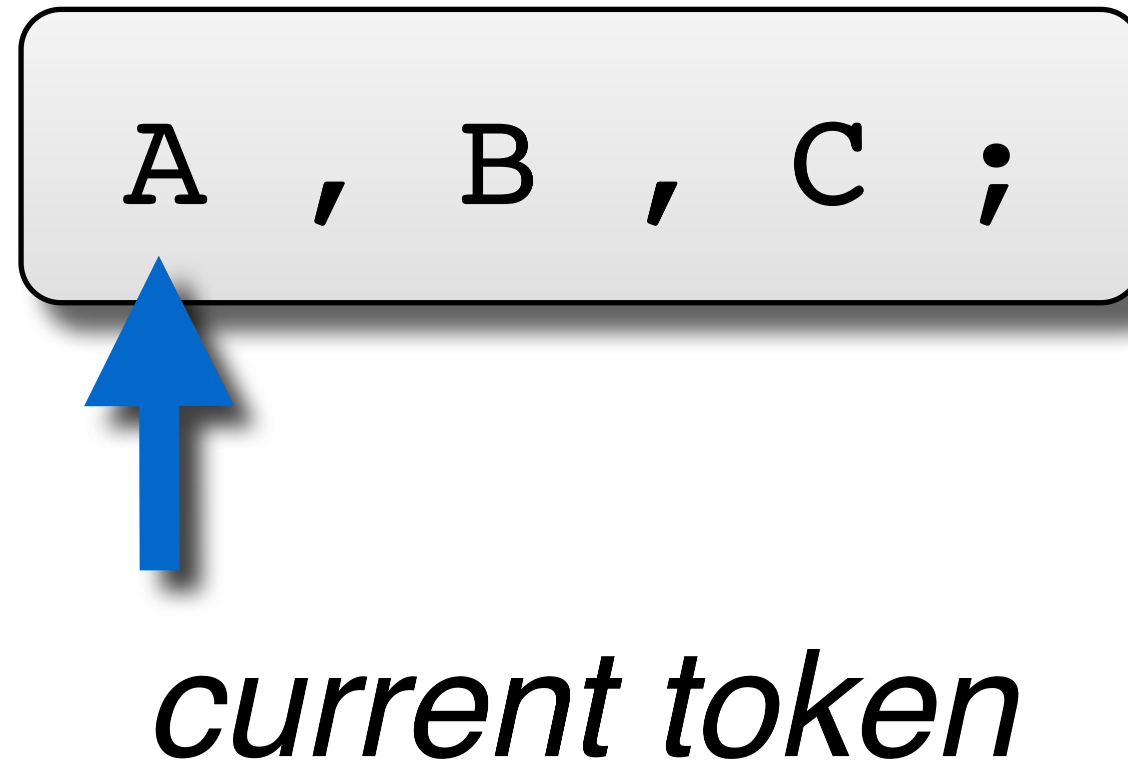
id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → `;`

Step 1

Begin with **root** (start symbol).



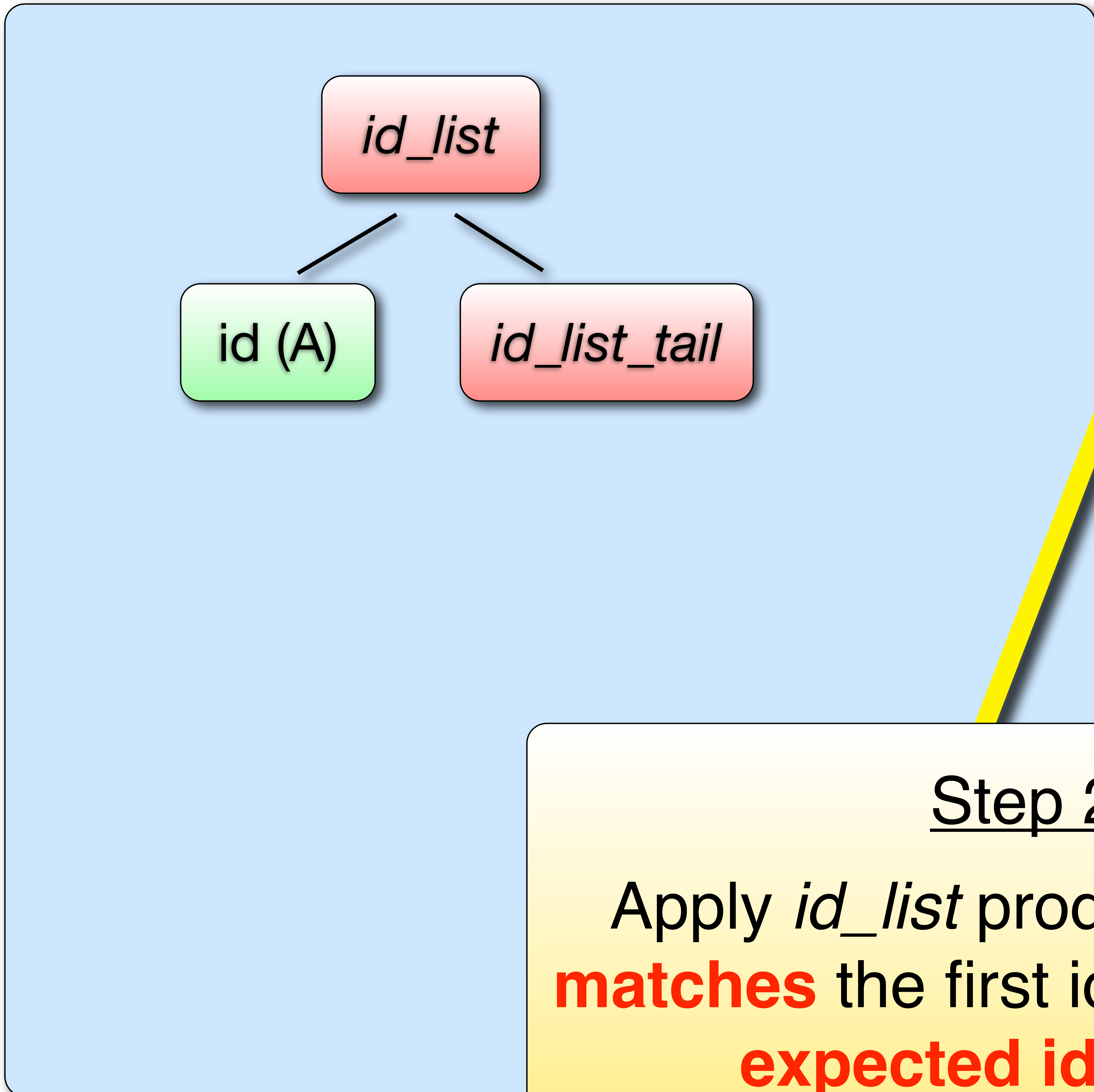
Wednesday, April 14, 2010

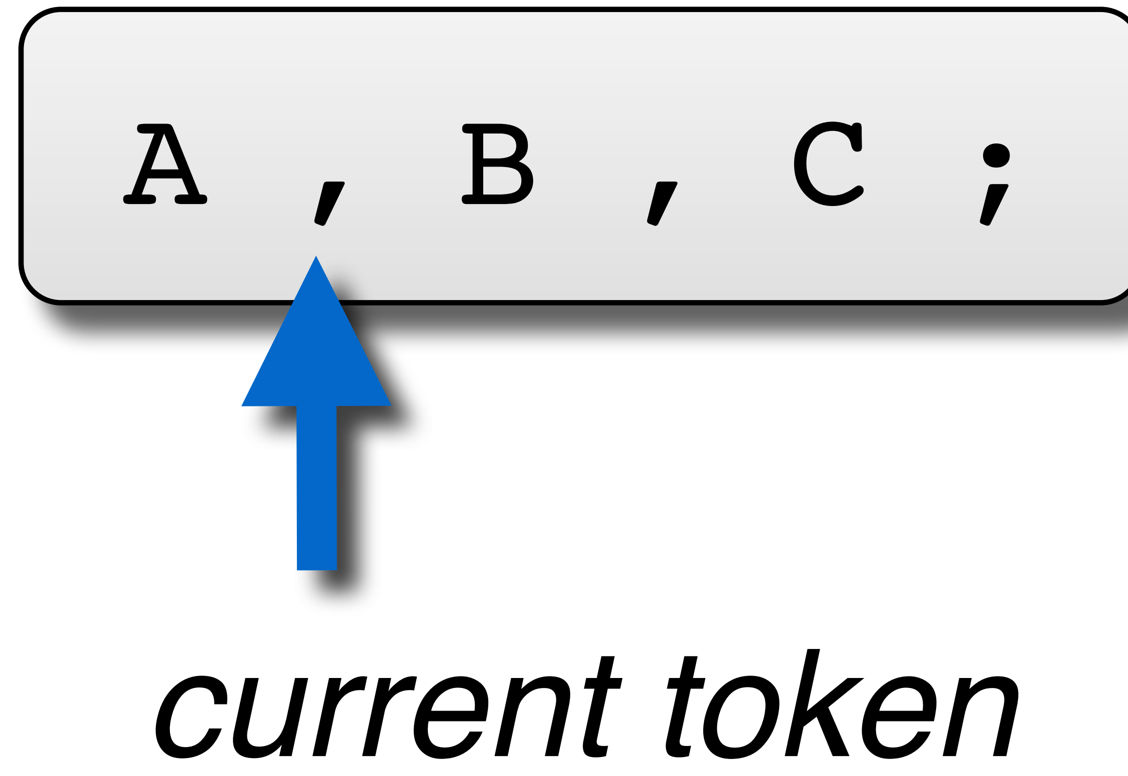
LL Example

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → ;





Step 2

Apply *id_list* production. This **matches** the first identifier to the **expected id token**.

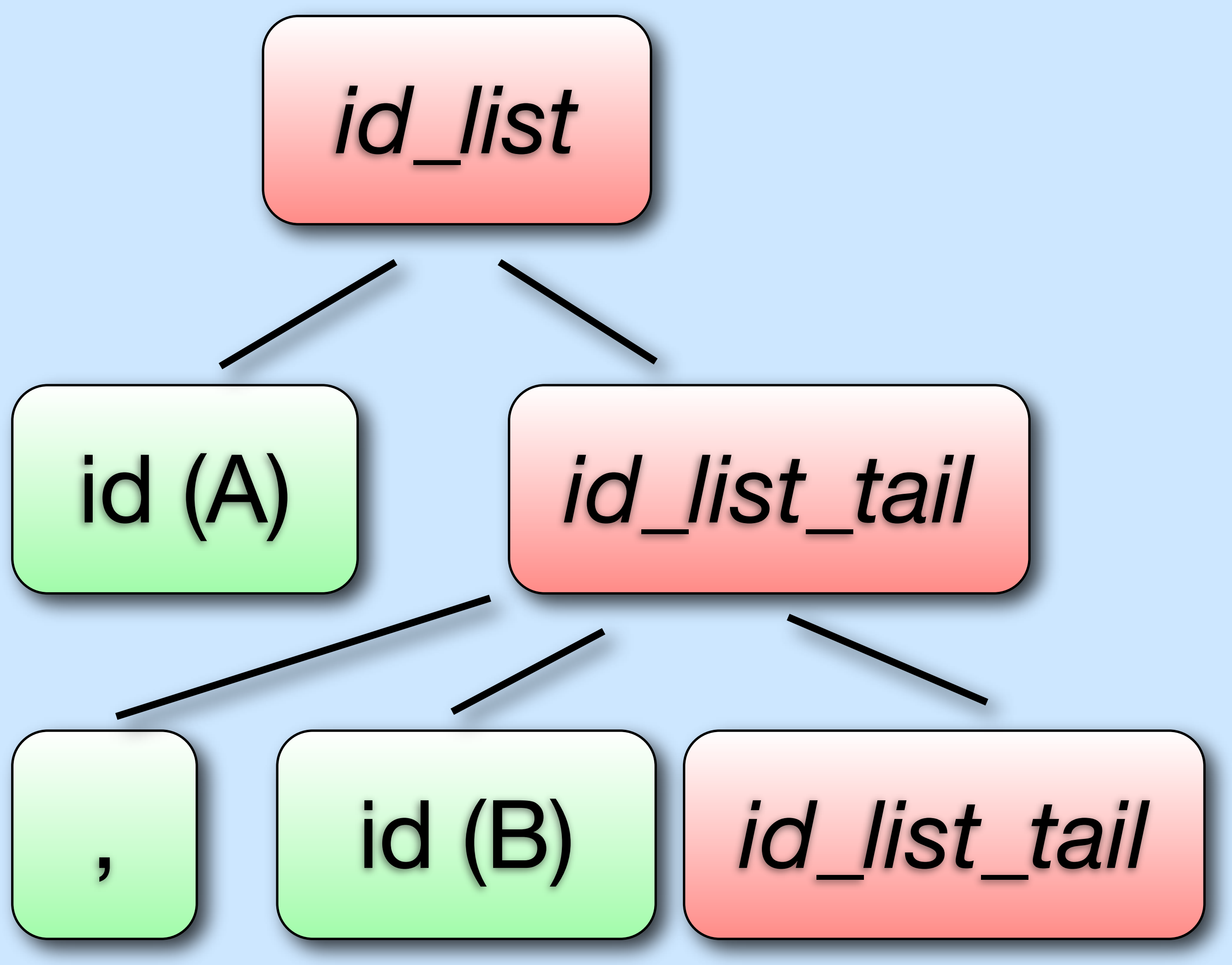
Wednesday, April 14, 2010

LL Example

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → ;



`A , B , C ;`

↑ *current token*

Step 3

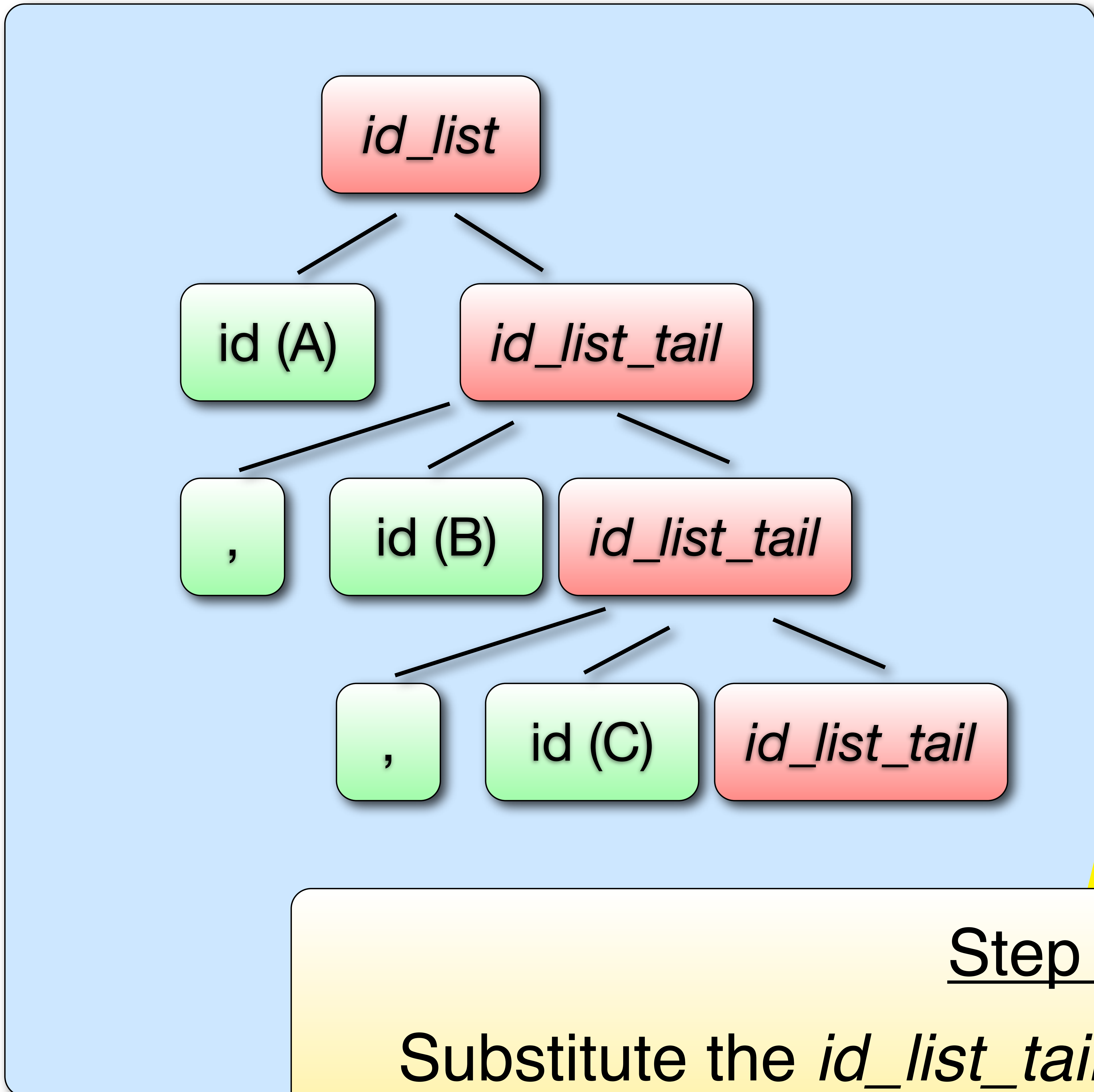
Apply a production for *id_list_tail*.
There are two to choose from.
Predict that the first one applies.
This matches two more tokens.

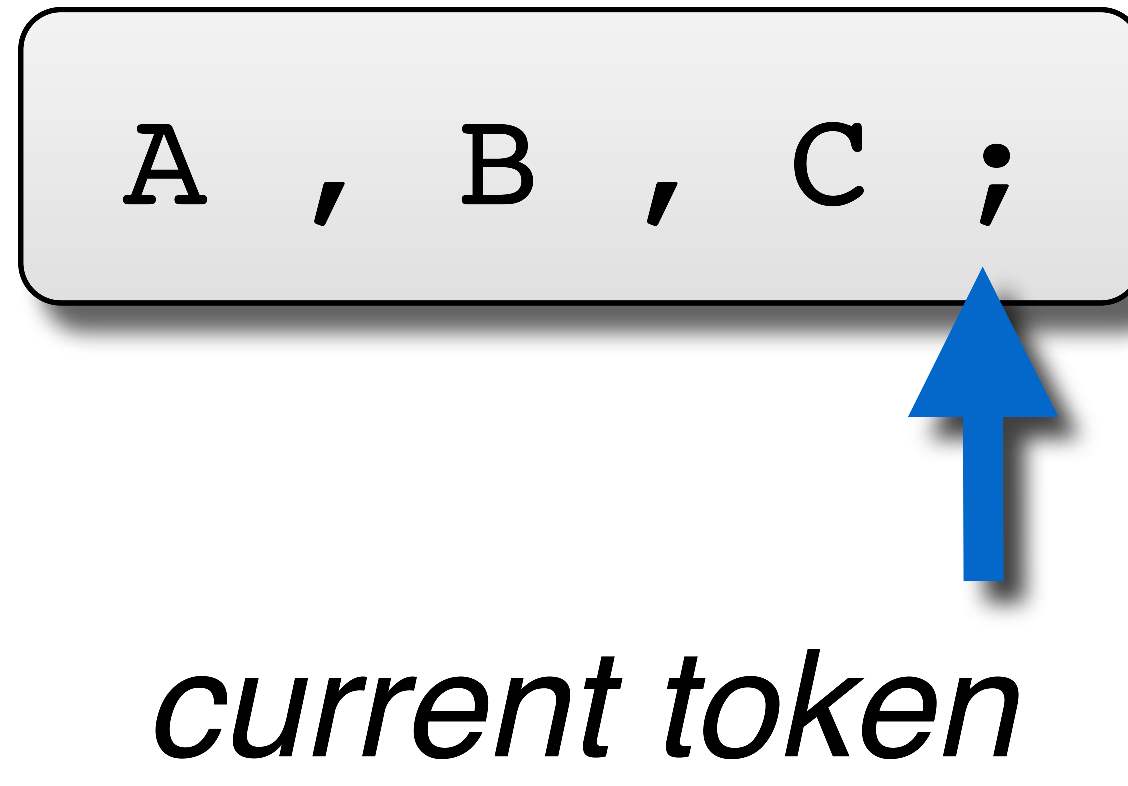
LL Example

id_list → id *id_list_tail*

id_list_tail → , id *id_list_tail*

id_list_tail → ;





<u>Step 4</u>

Substitute the *id_list_tail*, **predicting** the first production again. This matches a comma and C.

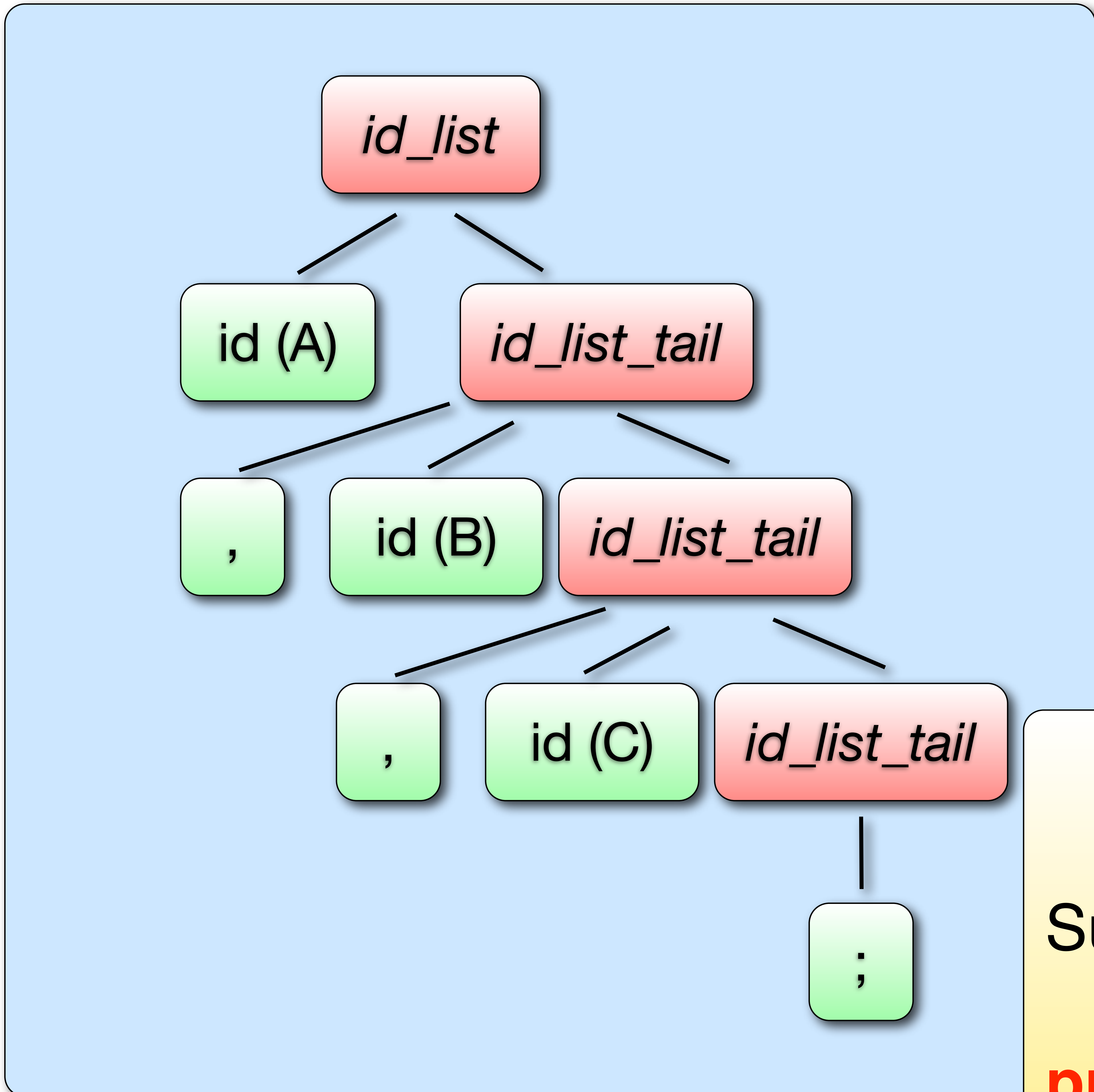
Wednesday, April 14, 2010

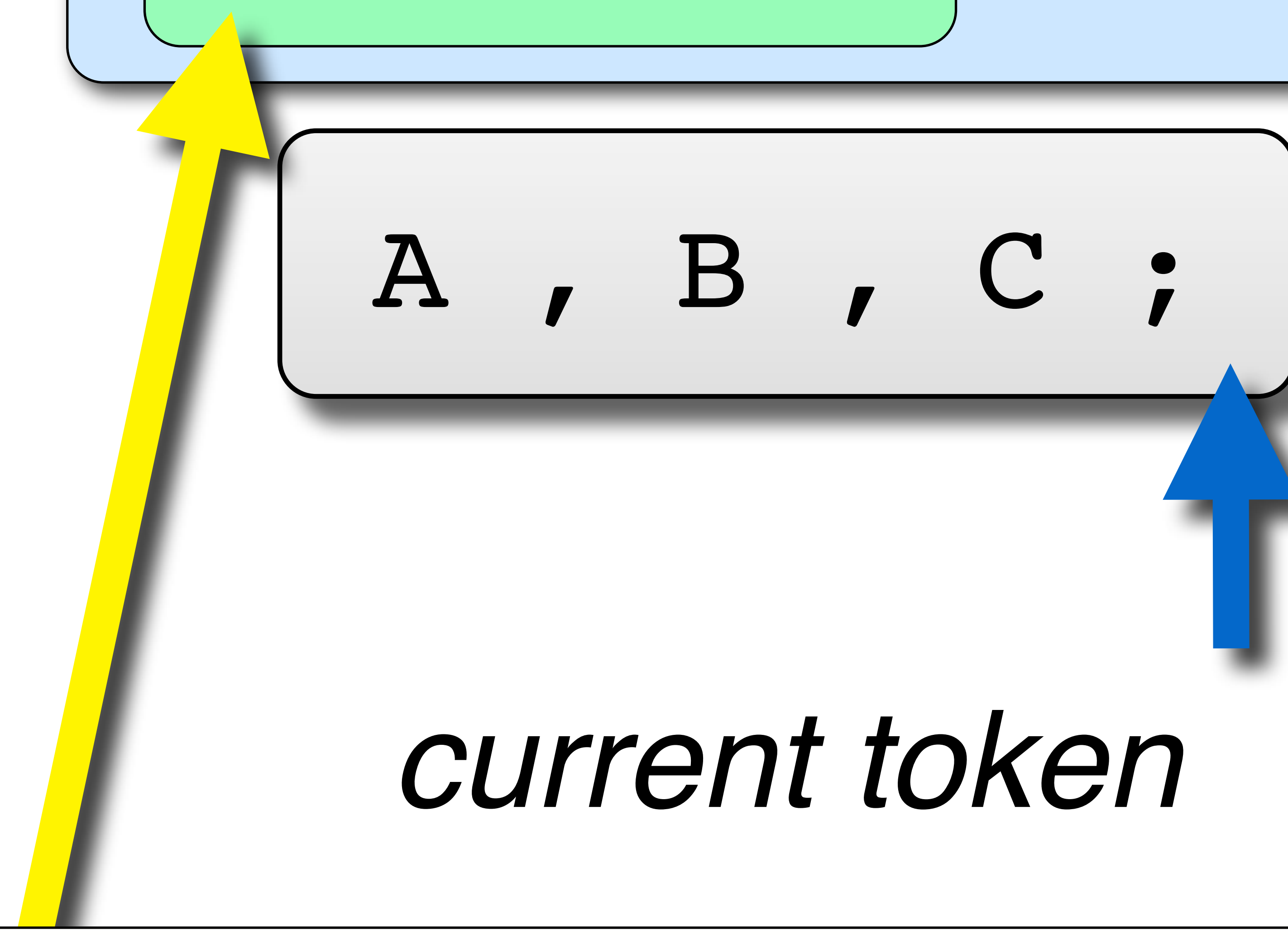
LL Example

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → `;`

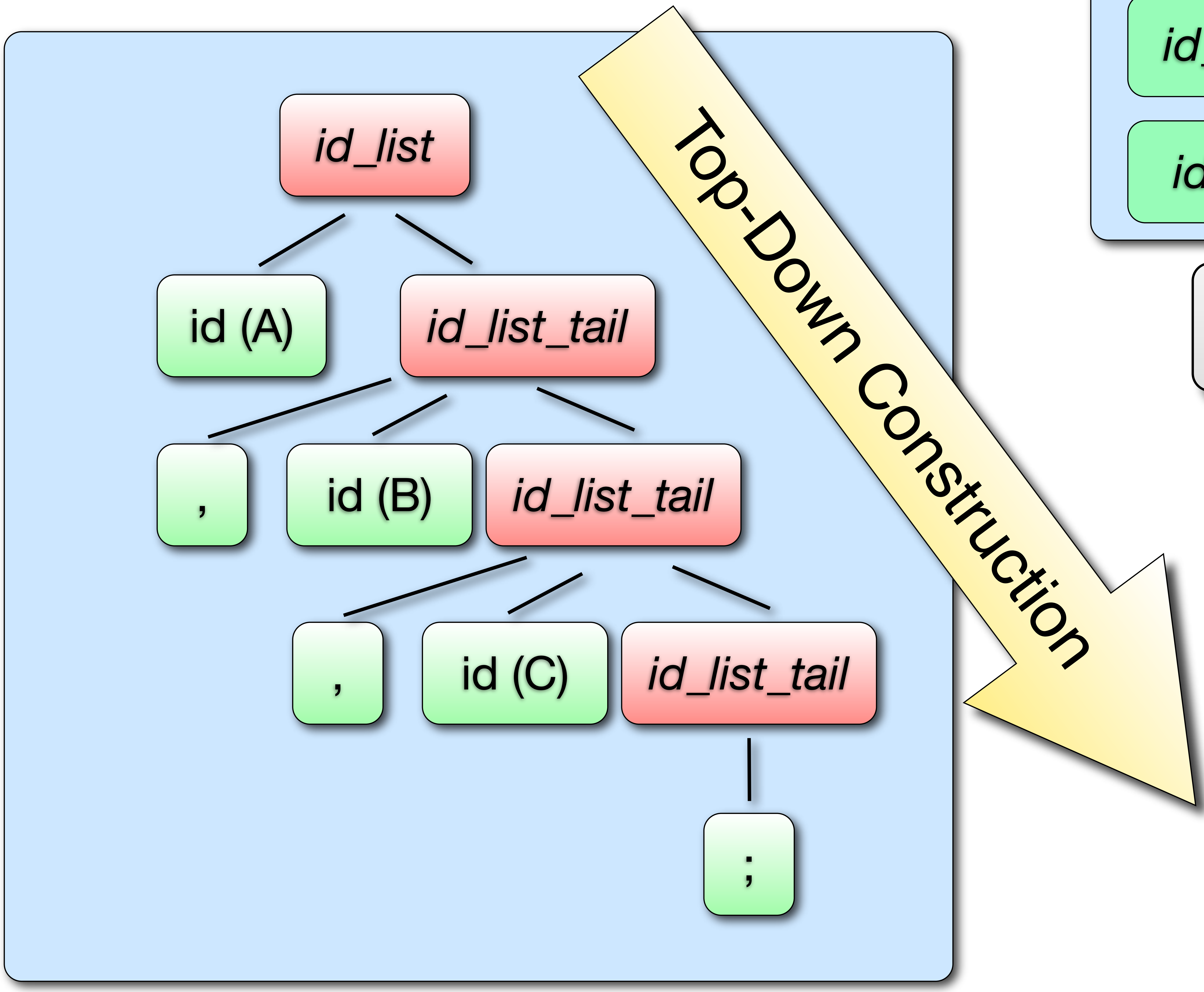




Step 5

Substitute the final *id_list_tail*. This time **predict the other production**, which matches the ';'.

LL Parse Tree



id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → ;

`A , B , C ;`

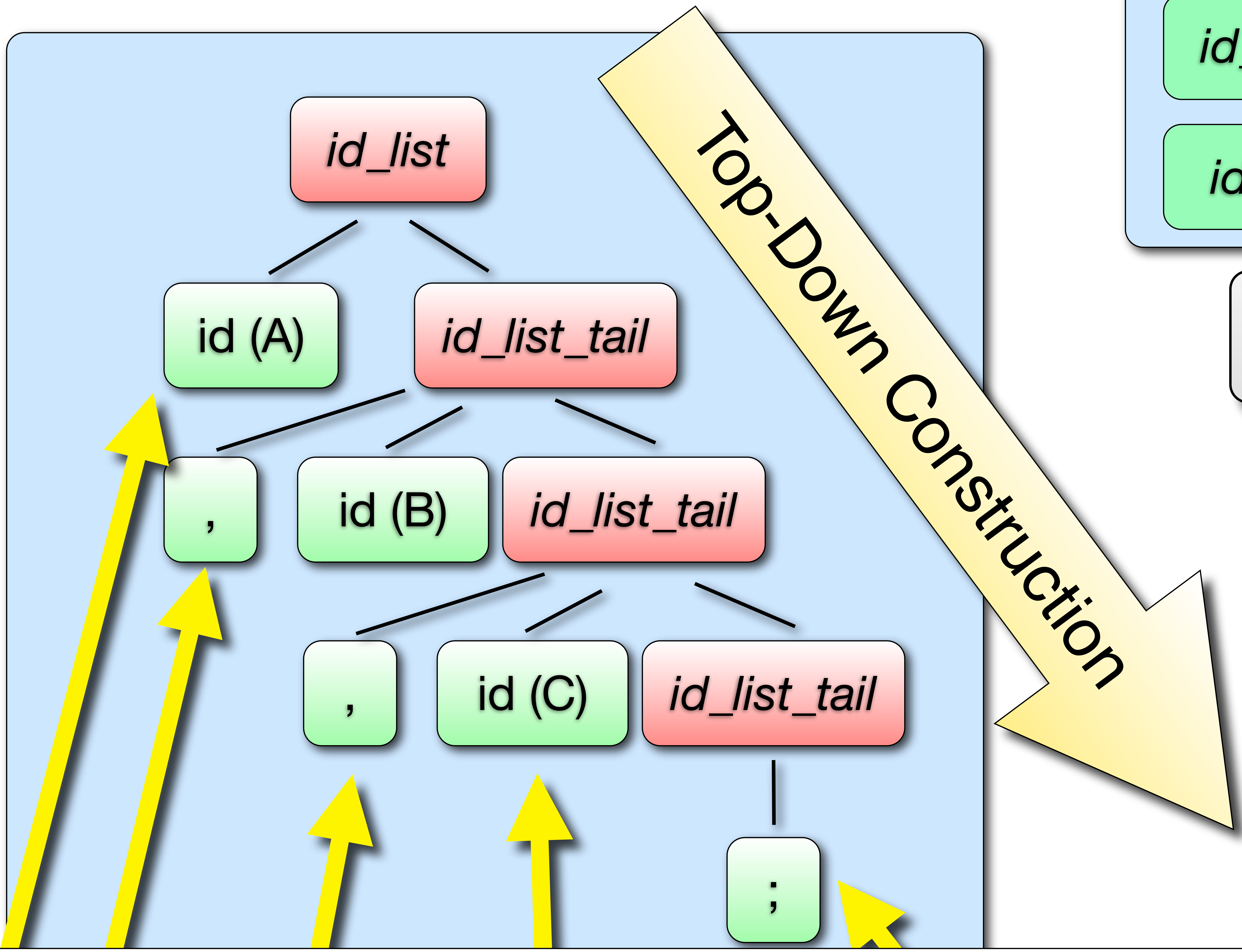
LL Parse Tree

id_list → id *id_list_tail*

id_list_tail → , id *id_list_tail*

id_list_tail → ;

A , B , C ;



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

LR Example

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → `;`

`A , B , C ;`

forest (a stack)

LR Example

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → `;`

`A , B , C ;`

Forest = set of (partial) trees.

forest (a stack)

LR Example

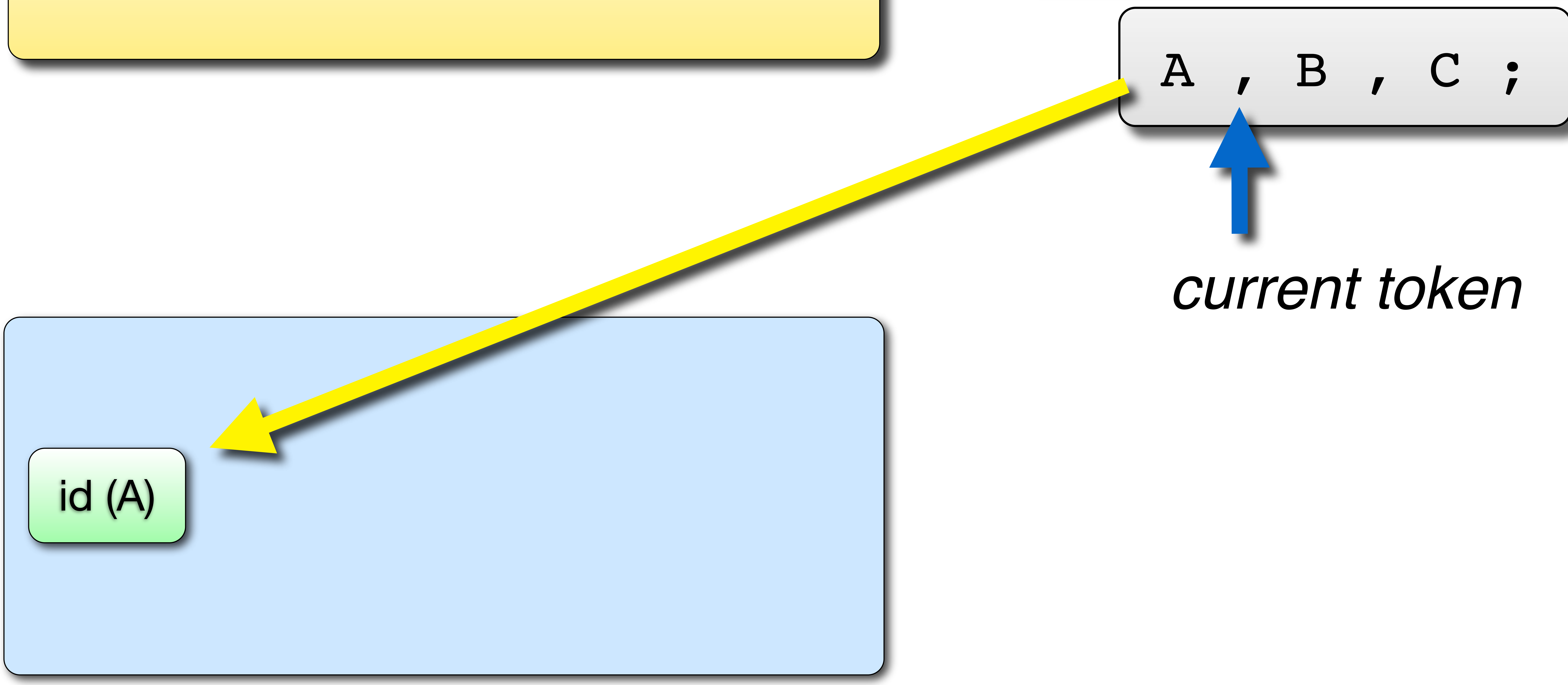
$id\_list \rightarrow$ `id` id_list_tail

$id\_list\_tail \rightarrow$, `id` id_list_tail

$id\_list\_tail \rightarrow$;

Step 1

Shift encountered token into forest.



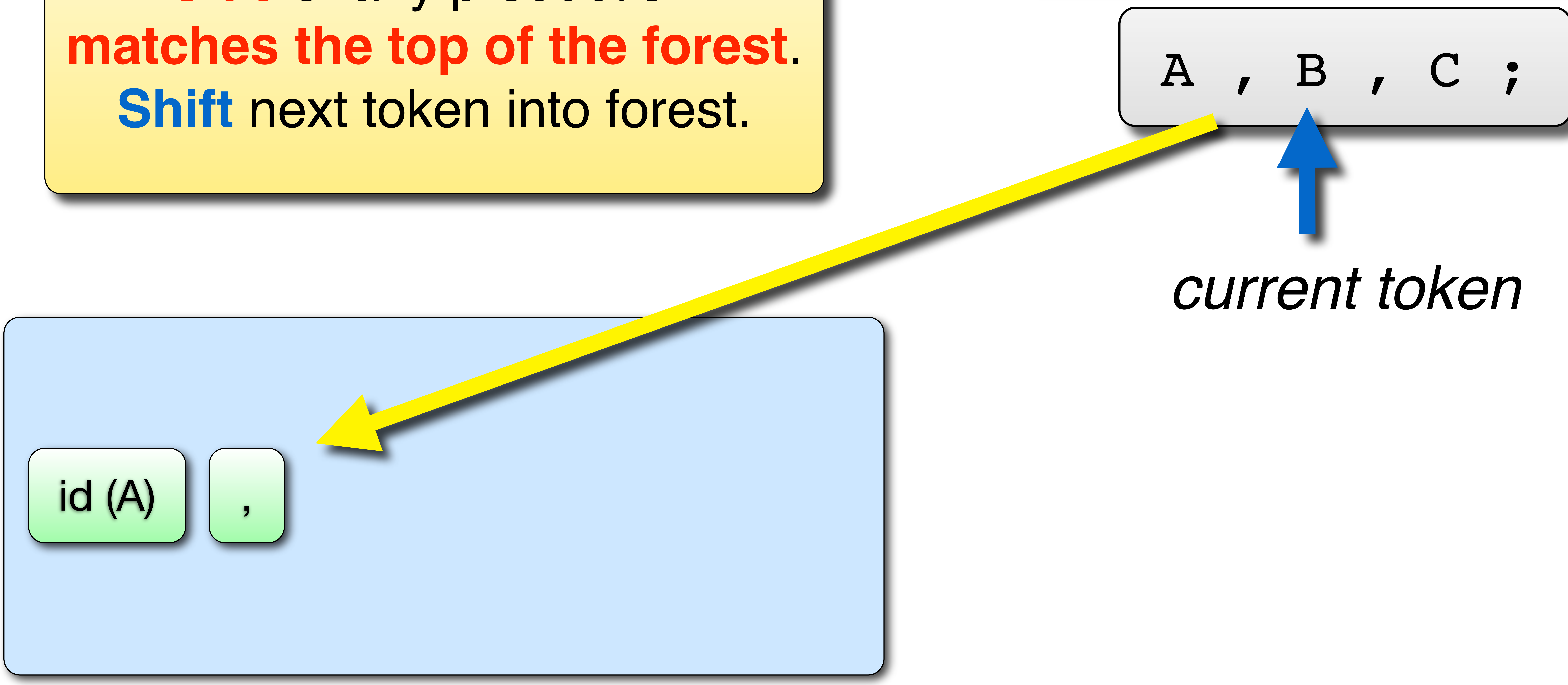
forest (a stack)

LR Example

Step 2

Determine that **no right-hand side** of any production **matches the top of the forest**. **Shift** next token into forest.

- *id_list* → `id` *id_list_tail*
- *id_list_tail* → `, id` *id_list_tail*
- *id_list_tail* → `;`



forest (a stack)

LR Example

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → `;`

<u>Steps 3-6</u>

No right hand side **matches** top of forest. **Repeatedly shift** next token into forest.

`A , B , C ;`

current token

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

forest (a stack)

LR Example

<u>Step 7</u>

Detect that **last production matches the top of the forest**.
Reduce top token to partial tree.

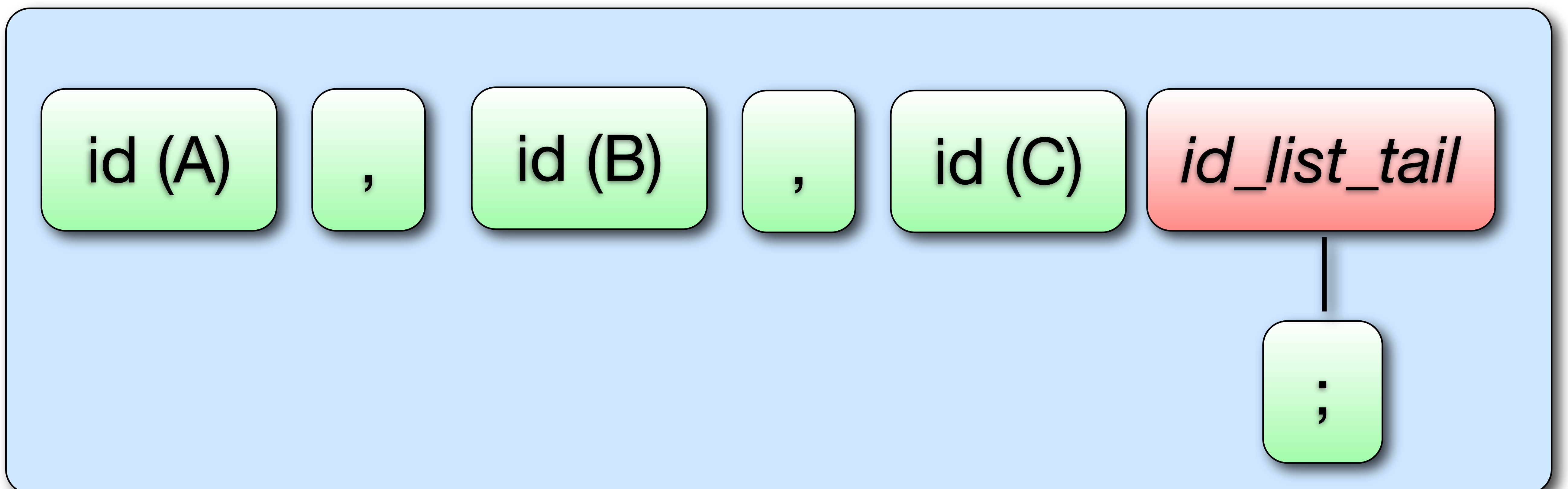
id_list → `id` *id_list_tail*

id_list_tail → , `id` *id_list_tail*

id_list_tail → ;

`A , B , C ;`

current token



forest (a stack)

LR Example

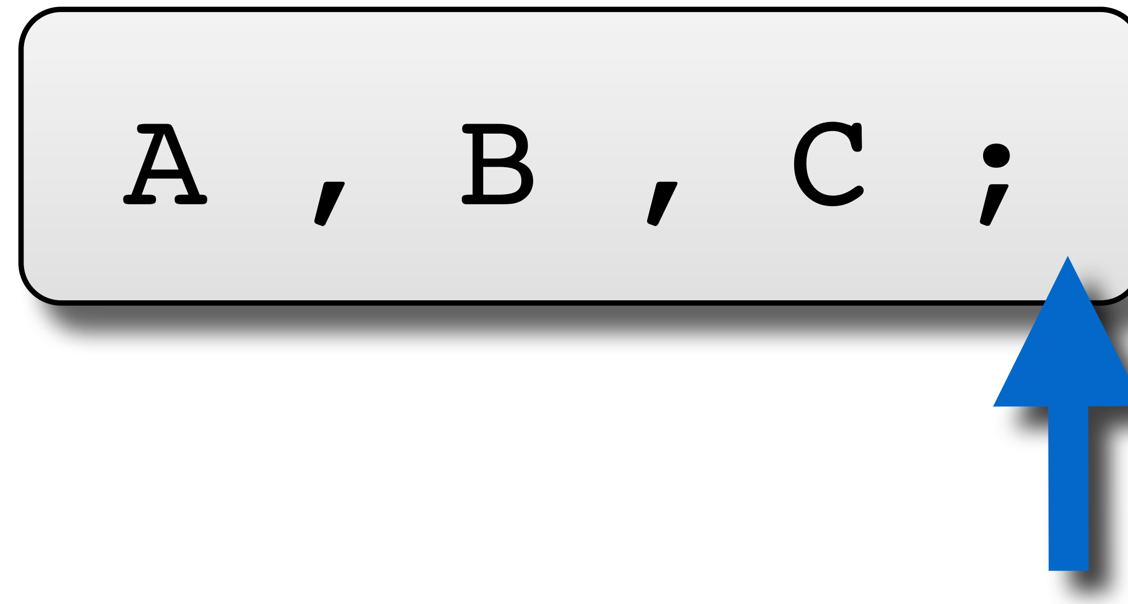
<u>Step 8</u>

Detect that **second production matches**. **Reduce top of forest.**

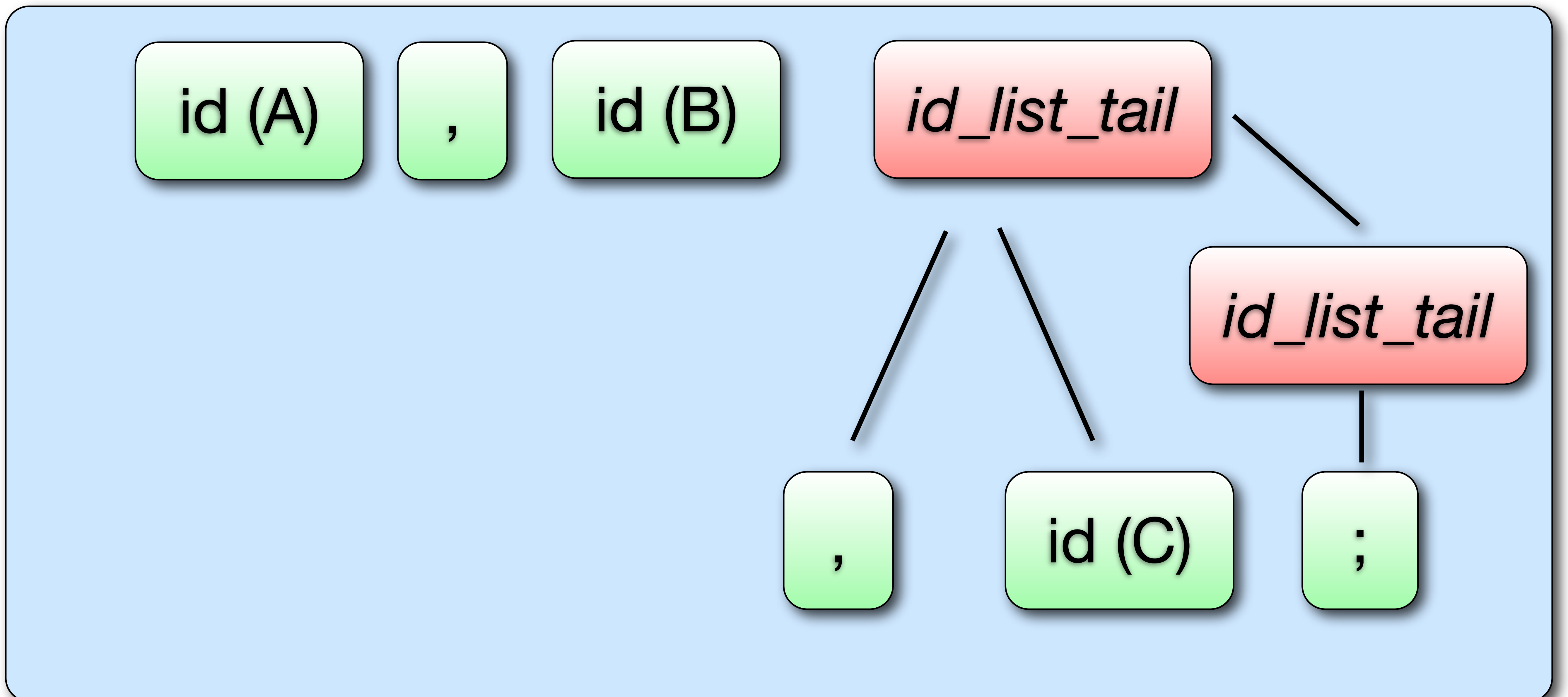
id_list → id *id_list_tail*

id_list_tail → , id *id_list_tail*

id_list_tail → ;



current token



forest (a stack)

LR Example

Step 9

Detect that **second production matches**. **Reduce top of forest.**

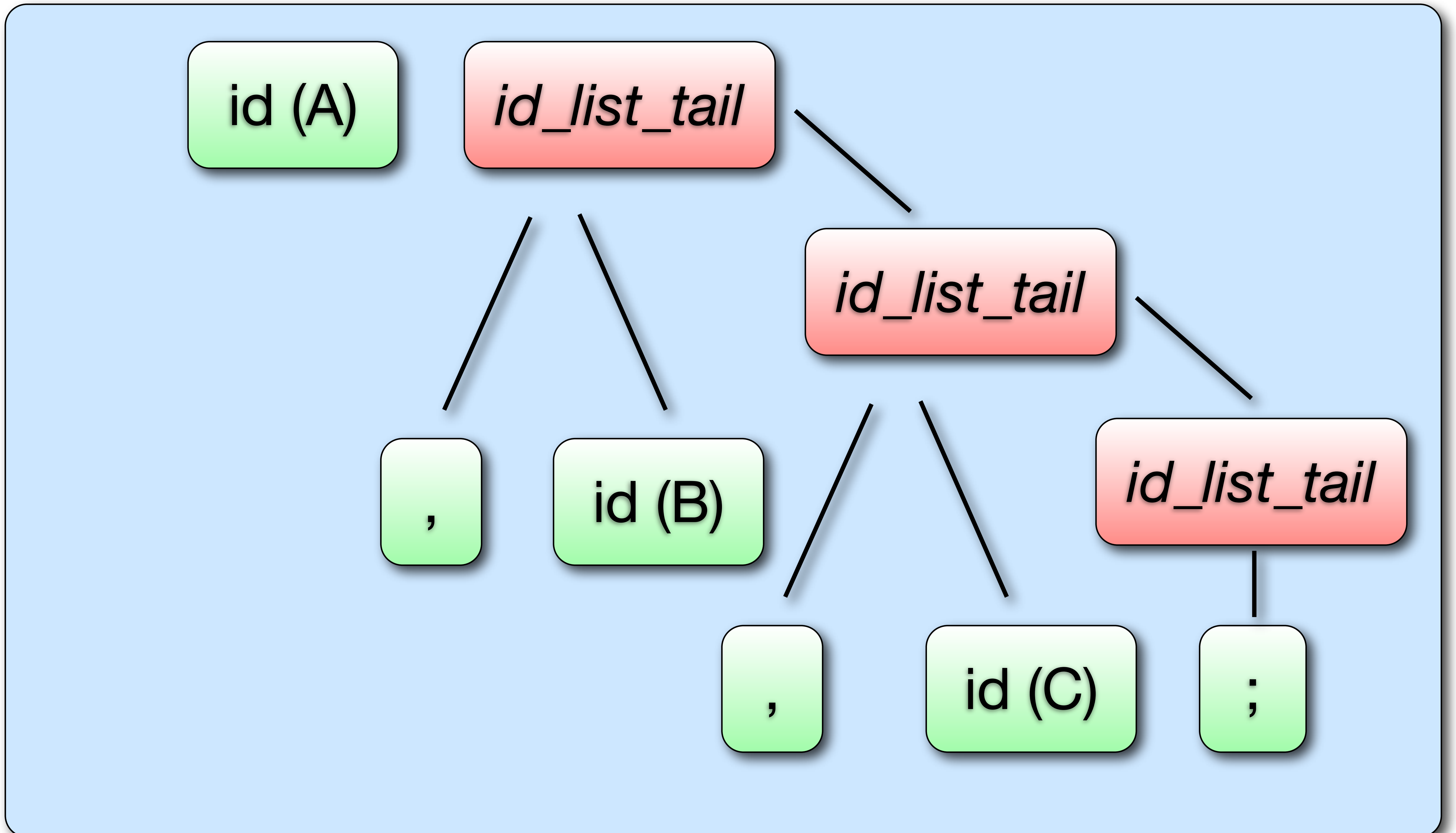
id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

id_list_tail → ;

`A , B , C ;`

current token



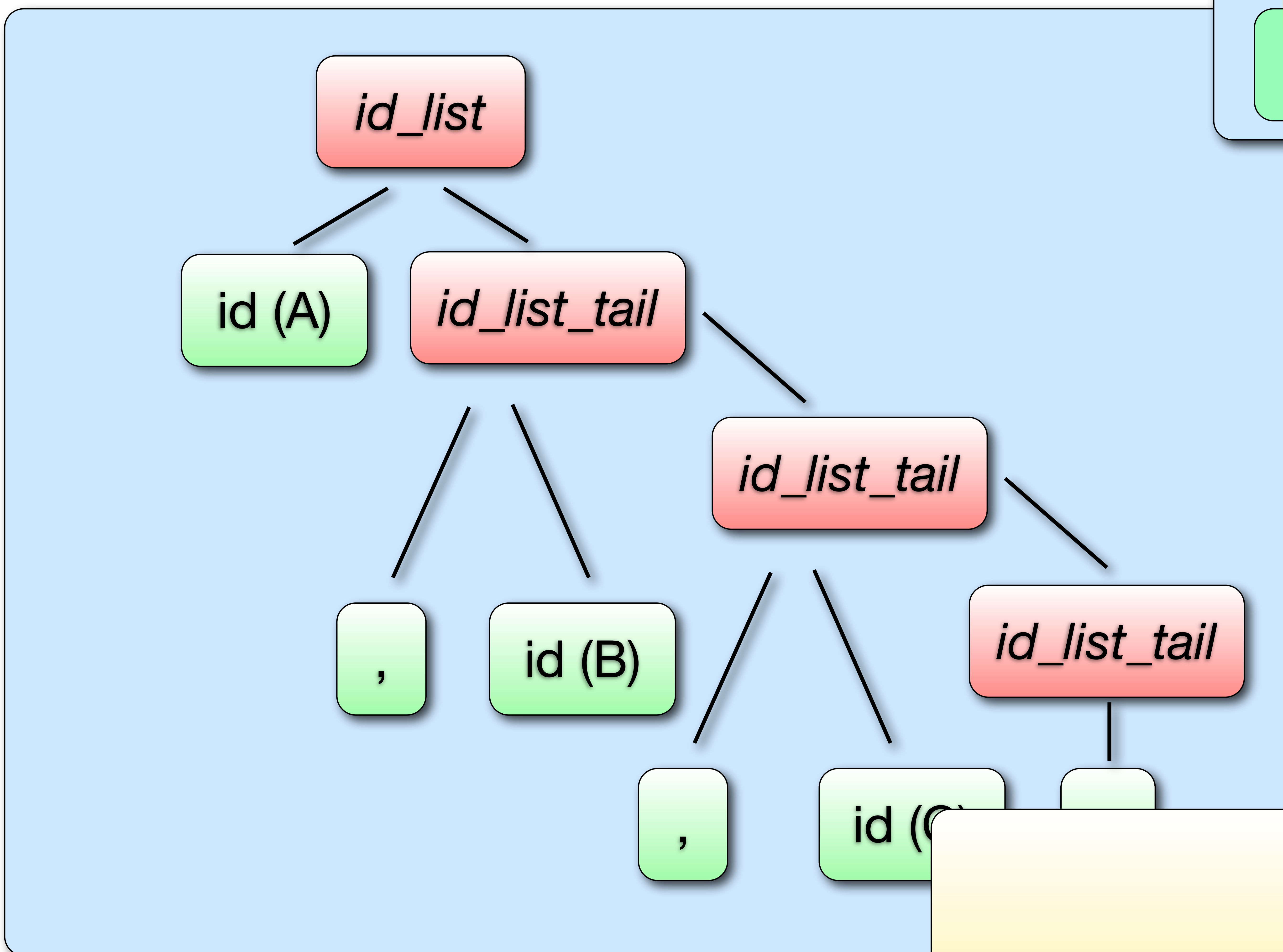
forest (a stack)

LR Example

id_list → id *id_list_tail*

id_list_tail → , id *id_list_tail*

id_list_tail → ;



forest (a stack)

A , B , C ;

current token

Step 10

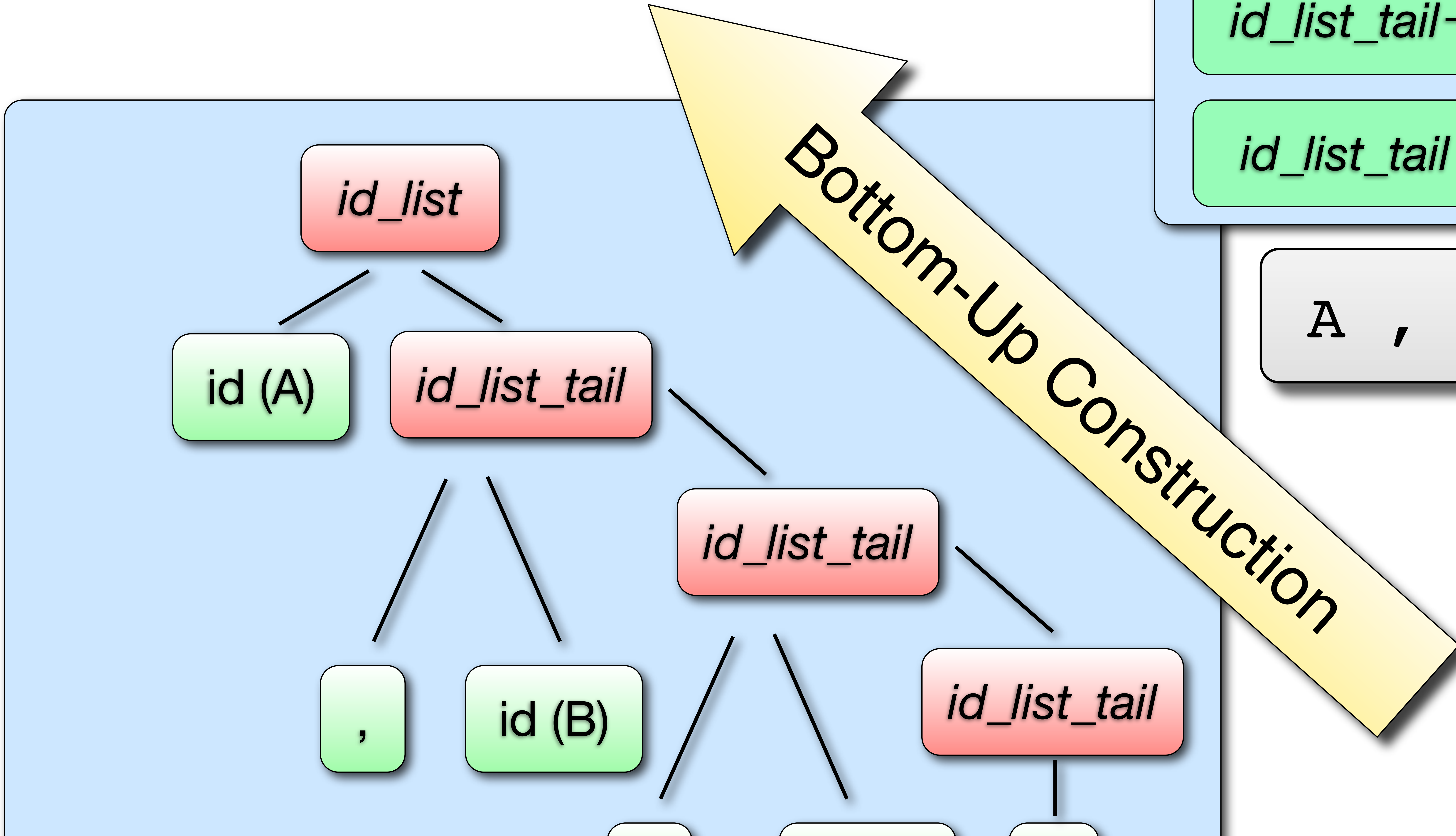
Detect that **first production matches**. **Reduce top of forest.**

LR Parse Tree

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

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 **reduction** takes place.

An Equivalent Grammar

better suited to LR parsing

id_list → *id_list_prefix* ;

id_list_prefix → *id_list_prefix* , `id`

id_list_prefix → `id`

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

An Equivalent Grammar

better suited to LR parsing

id_list → *id_list_prefix ;*

id_list_prefix → *id_list_prefix* , `id`

id_list_prefix → `id`

However, this creates a problem for the LL parser.

When the parser discovers an "`id`" it cannot predict the number of *id_list_prefix* productions that it needs to match.

Two Approaches to LL Parser Construction

Recursive Descent.

- ➡ A mutually **recursive** set of subroutines.
- ➡ One subroutine per non-terminal.
- ➡ **Case statements** based on current token to **predict** subsequent productions.

Table-Driven.

- ➡ Not recursive; instead has an explicit stack of expected symbols.
- ➡ A loop that processes the top of the stack.
- ➡ Terminal symbols on stack are simply matched.
- ➡ Non-terminal symbols are replaced with productions.
- ➡ Choice of production is driven by table.

Recursive Descent Example

"recursive descent"
=
"climb from root to leaves, calling a subroutine for every level"

Identifier List Grammar.

- ➡ Recall our LL-compatible original version.

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → `;`

Recursive Descent Approach.

- ➡ We need **one subroutine for each non-terminal**.
- ➡ Each subroutine **adds tokens** into the growing parse tree and/or **calls further subroutines to resolve non-terminals**.

Recursive Descent Example

"recursive descent"
=
"climb from root to leaves, calling a subroutine for every level"

Identifier List Grammar.

- ➡ Recall our LL-compatible original version.

id_list_tail → ;

Possibly itself.
Recursive descent: either directly or indirectly.

Recursive Descent Approach.

- ➡ We need **one subroutine for each non-terminal**.
- ➡ Each subroutine **adds tokens** into the growing parse tree and/or **calls further subroutines to resolve non-terminals**.

Recursive Descent Example

Helper routine “match”.

- Used to **consume** expected terminals/ tokens.
- Given an **expected token type** (e.g., `id`, ‘;’, or ‘,’), checks if next token is of correct type.
- Raises **error** otherwise.

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → ;

```
subroutine match(expected_type):
 token = get_next_token()
 if (token.type == expected_type):
   make token a child of left-most non-terminal in tree
 else:
   throw ParseException(“expected “ + expected_type)
```

Recursive Descent Example

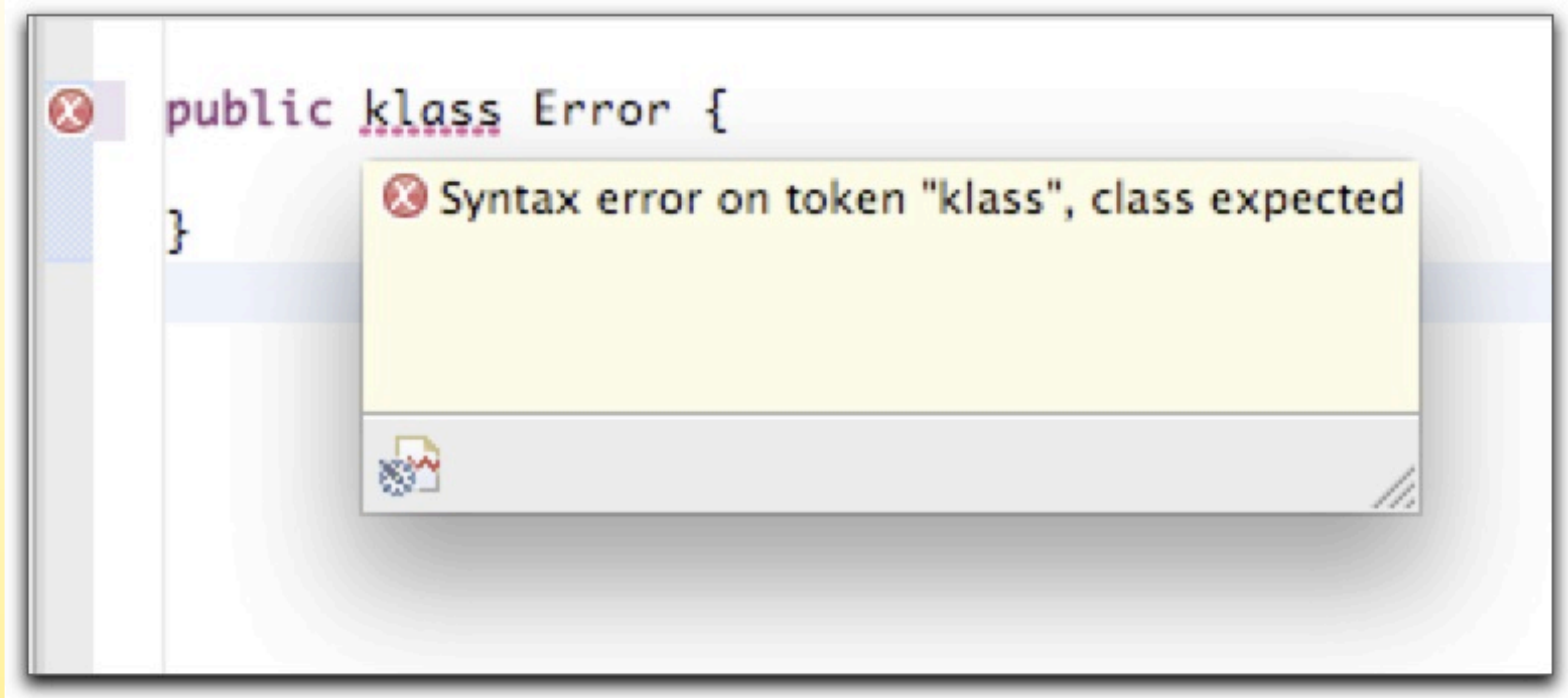
Helper routin
- ➡ Used to **co** tokens.
- ➡ Given an **ex** or ‘,’), check
- ➡ Raises **erro**

id_list_tail

id *id_list_tail*

;

Example of a match failure:



Eclipse: `class` token expected, but got `id`

```
subroutin
 token = g
 if (token.type == expected_type):
   make token a child of left-most non-terminal in tree
 else:
   throw ParseException("expected " + expected_type)
```


Wednesday, April 14, 2010

Recursive Descent Example

Parsing *id_list*.

- Trivial, there is only one production.
- Simply match an id, and then delegate parsing of the tail to the subroutine for *id_list_tail*.

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → `;`

```
subroutine parse_id_list():
  match(ID_TOKEN)
  parse_id_list_tail()
```

Recursive Descent Example

Parsing *id_list*.

- **Trivial**, there is only one production.
- Sim
  pars
  id_l

id_list → `id` *id_list_tail*

list_tail → `, id` *id_list_tail*

list_tail → ;

This **delegation** is the "**descent**" part in recursive descent parsing.

```
subroutine parse_id_list():
  match(ID_TOKEN)
  parse_id_list_tail()
```

Recursive Descent Example

Parsing *id_list_tail*.

- ➡ There are **two productions** to choose from.
- ➡ This require **predicting** which one is the correct one.
- ➡ This requires **looking ahead** and examining the **next token** (without consuming it).

id_list → `id` *id_list_tail*

id_list_tail → `,` `id` *id_list_tail*

id_list_tail → ;

```
subroutine parse_id_list_tail():
  type = peek_at_next_token_type()
  case type of
    COMMA_TOKEN:
      match(COMMA_TOKEN); match(ID_TOKEN); parse_id_list_tail()
    SEMICOLON_TOKEN:
      match(SEMICOLON_TOKEN);
```

Recursive Descent Example

Parsing *id_list_tail*.

- ➡ There are **two productions** to choose from.
- ➡ This require **predicting** wh the correct one.
- ➡ This requires **looking ahea** examining the **next token** (consuming it).

id_list → `id` *id_list_tail*

id_list_tail → `, id` *id_list_tail*

This **delegation** is the "**recursive**" part in recursive descent parsing.

```
subroutine parse_id_list_tail():
  type = peek_at_next_token_type()
  case type of
    COMMA_TOKEN:
      match(COMMA_TOKEN); match(ID_TOKEN); parse_id_list_tail()
    SEMICOLON_TOKEN:
      match(SEMICOLON_TOKEN);
```

Recur…

Parsing *id_list_tail*.

- ➡ There are **two produ…** from.
- ➡ This require **predicti…** the correct one.
- ➡ This requires **lookin…** examining the **next t…** consuming it).

> We need one token "**lookahead.**"
>
> Parsers that require **k** tokens lookahead are called **LL(k)** (or **LR(k)**) parsers.
>
> Thus, this is a **LL(1)** parser.

```
subroutine parse_id_list_tail():
  type = peek_at_next_token_type()
  case type of
    COMMA_TOKEN:
      match(COMMA_TOKEN); match(ID_TOKEN); parse_id_list_tail()
    SEMICOLON_TOKEN:
      match(SEMICOLON_TOKEN);
```

LL(k) Parsers

Recall our non-LL compatible grammar.

➡Better for LR-parsing, but **problematic for predictive parsing**.

id_list → *id_list_prefix ;*

id_list_prefix → *id_list_prefix* , `id`

id_list_prefix → `id`

Cannot be parsed by LL(1) parser.

➡**Cannot predict** which *id_list_production* to choose if next token is of type `id`.

➡However, **a LL(2) parser can parse this grammar**. Just look at the second token ahead and disambiguate based on ‘,’ vs. ‘;’.

LL(k) Parsers

Recall our non-LL compatible grammar.

➡ E **rsing**.

Bottom-line:

can enlarge class of supported grammars by using **k > 1** lookahead, but at the expense of **reduced performance** / backtracking.

Most production LL parsers use **k = 1**.

Ca

➡ **Cannot predict** which *id_list_production* to choose if next token is of type `id`.

➡ However, **a LL(2) parser can parse this grammar**. Just look at the second token ahead and disambiguate based on ‘,’ vs. ‘;’.

Predict Sets

```
subroutine parse_id_list_tail():
  type = peek_at_next_token_type()
  case type of
    COMMA_TOKEN:
      match(COMMA_TOKEN); match(ID_TOKEN); parse_id_list_tail()
    SEMICOLON_TOKEN:
      match(SEMICOLON_TOKEN);
```

The question is how do we **label the case statements** in general, i.e., for **arbitrary** LL grammars?

First, Follow, and Predict

sets of terminal symbols

FIRST(*A*):

- The terminals that can be the first token of a valid derivation starting with symbol ***A***.
- Trivially, for each terminal ***T***, **FIRST(*T*)** = {***T***}.

FOLLOW(*A*):

- The terminals that can follow the symbol ***A*** in any valid derivation. (***A*** is usually a non-terminal.)

PREDICT(*A* → α):

- The terminals that can be the first tokens as a result of the production ***A*** → α. (α is a string of symbols)
- The terminals in this set form the **label in the case statements** to predict ***A*** → α.

First, Follow, and Predict

sets of terminal symbols

Note: For a non-terminal ***A***, the set **FIRST(*A*)** is the union of the predict sets of all productions with ***A*** as the head:

if there exist three productions $A \rightarrow \alpha$, $A \rightarrow \beta$, and $A \rightarrow \lambda$, then

FIRST(*A*) =

$$\text{PREDICT}(A \rightarrow \alpha) \cup \text{PREDICT}(A \rightarrow \beta) \cup \text{PREDICT}(A \rightarrow \lambda)$$

PREDICT($A \rightarrow \alpha$):

- ➡The terminals that can be the first tokens as a result of the production $A \rightarrow \alpha$. (α is a string of symbols)
- ➡The terminals in this set form the **label in the case statements** to predict $A \rightarrow \alpha$.

PREDICT($A \rightarrow \alpha$)

If α is ε, i.e., if A is derived to "nothing":

PREDICT($A \rightarrow \varepsilon$) = FOLLOW($A$)

Otherwise, if α is a string of symbols that starts with X:

PREDICT($A \rightarrow X$…) = FIRST(X)

Inductive Definition of FIRST(A)

If A is a **terminal** symbol, then:

FIRST(A) = {A}

If A is a **non-terminal** symbol and there exists a production $A \rightarrow X\ldots$, then

FIRST(X) $\subseteq$ FIRST(A)

(X can be terminal or non-terminal)

Notation: X is the first symbol of the production body.

If A is a **terminal** symbol, then:

FIRST(A) = {A}

If A is a **non-terminal** symbol and there exists a production $A \rightarrow X\ldots$, then

FIRST(X) ⊆ FIRST(A)

(X can be terminal or non-terminal)

Inductive Definition of FOLLOW(*A*)

If the substring ***AX*** exists anywhere in the grammar, then

FIRST(*X*) ⊆ FOLLOW(*A*)

If there exists a production ***X*** → ***...A***, then

FOLLOW(*X*) ⊆ FOLLOW(*A*)

Notation: *A* is the last symbol of the production body.

If the substring *AX* exists anywhere in the grammar, then

FIRST(*X*) ⊆ FOLLOW(*A*)

If there exists a production *X* → …*A*, then

FOLLOW(*X*) ⊆ FOLLOW(*A*)

Computing First, Follow, and Predict

Inductive Definition.

- ➡FIRST, FOLLOW, and PREDICT are defined in terms of each other.
- ➡**Exception**: FIRST for **terminals**.
- ➡This the base case for the induction.

Iterative Computation.

- ➡**Start with FIRST for terminals** and set all other sets to be **empty**.
- ➡**Repeatedly apply all definitions** (i.e., include known subsets).
- ➡Terminate when sets do not change anymore.

Predict Set Example

id_list → *id_list_prefix*;

[1] *id_list_prefix* → *id_list_prefix*, `id`

[2] *id_list_prefix* → `id`

Predict Set Example

id_list → *id_list_prefix*;

[1] *id_list_prefix* → *id_list_prefix*, `id`

[2] *id_list_prefix* → `id`

Base case: FIRST(`id`) = {`id`}

Induction for [2]: FIRST(`id`) ⊂ FIRST(*id_list_prefix*) = {`id`}

Induction for [1]: FIRST(*id_list_prefix*) ⊂ FIRST(*id_list_prefix*)

Predict sets for (2) and (1) are identical: not LL(1)!

Left Recursion

Leftmost symbol is a recursive non-terminal symbol.

➡ This causes a grammar **not to be LL(1)**.

➡ Recursive descent would enter **infinite recursion**.

➡ It is **desirable for LR grammars**.

id_list → *id_list_prefix*;

*id_list*_prefix → *id_list_prefix,* `id`

*id_list*_prefix → `id`

“To parse an *id_list_prefix*, call the parser for *id_list_prefix*, which calls the parser for *id_list_prefix*, which…”

Leftm … **symbol.**

- ➡This …
- ➡Recu … **ion**.
- ➡It is **desirable for LL grammars**.

id_list → *id_list_prefix*;

*id_list*_prefix → *id_list_prefix*, `id`

*id_list*_prefix → `id`

Left-Factoring

Introducing "tail" symbols to avoid left recursion.

➡Split a recursive production in an **unambiguous prefix** and an **optional tail**.

expr → term | expr add_op term

is equivalent to

prefix	*expr → term expr_tail*
tail	*expr_tail → ε*
tail	*expr_tail → add_op expr*

Another Predict Set Example

[1] *cond* → `if` *expr* `then` *statement*

[2] *cond* → `if` *expr* `then` *statement* `else` *statement*

Another Predict Set Example

[1] *cond* → `if` *expr* `then` *statement*

[2] *cond* → `if` *expr* `then` *statement* `else` *statement*

PREDICT([1]) = {`if`} PREDICT([2]) = {`if`}

If the next token is an **`if`**, **which production** is the right one?

Common Prefix Problem

Non-disjoint predict sets.

➡ In order to predict which production will be applied, **all predict sets** for a given non-terminal need to be **disjoint**!

If there exist two productions $A \rightarrow \alpha$, $A \rightarrow \beta$ such that there exists a terminal x for which

$$x \in \text{PREDICT}(A \rightarrow \alpha) \cap \text{PREDICT}(A \rightarrow \beta),$$

then an LL(1) parser cannot properly predict which production must be chosen.

Can also be addressed with left-factoring…

Dangling else

Even if left recursion and common prefixes have been removed, a language may not be LL(1).

- ➡In many languages an **else statement** in if-then-else statements **is optional**.
- ➡Ambiguous grammar: **which if to match else to**?

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

Dangling else

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

- Can be handled with a tricky LR grammar.
- There exists **no LL(1)** parser that can parse such statements.
- Even though a proper LR(1) parser can handle this, it may not handle it in a **method the programmer desires**.
- Good language design avoids such constructs.

Dangling else

- To write this code correctly (based on indention) “begin” and “end” statements must be added.
- This is LL compatible.

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

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

Dangling else

statement → … | *cond* | …

cond → `if` *expr* `then` *block_statement*

cond → `if` *expr* `then` *block_statement* `else` *block_statement*

block_statement → `begin` *statement** `end`

A grammar that avoids the "dangling else" problem.