CS 314 Principles of Programming Languages

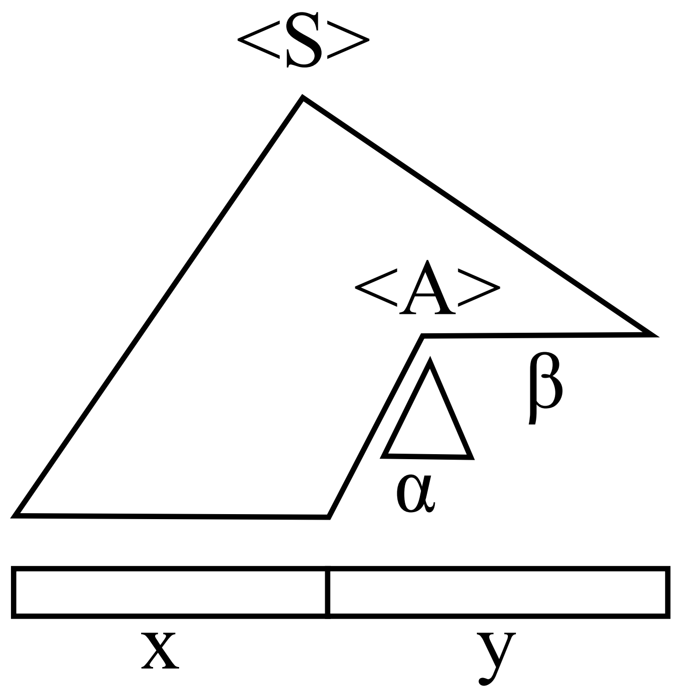
Lecture 6: LL(1) Parsing

Zheng (Eddy) Zhang
Rutgers University
February 5, 2018

Class Information

- Homework 2 due tomorrow.
- Homework 3 will be posted early next week.

Top - Down Parsing - LL(1)



<u>Basic Idea:</u>

- The parse tree is constructed from the root, expanding non-terminal nodes on the tree's frontier following a **leftmost** derivation.
- The input program is read from **left** to right, and input tokens are read (consumed) as the program is parsed.
- The next non-terminal symbol is replaced using one of its rules. The particular choice <u>has to be unique</u> and uses parts of the input (partially parsed program), for instance the first token of the remaining input.

Top - Down Parsing - LL(1) (cont.)

Example:

$S ::= a\ S\ b \mid \varepsilon$

How can we parse (automatically construct a leftmost derivation) the input string **a a a b b b** using a PDA (push-down automaton) and only the first symbol of the remaining input?

INPUT: | a a a b b b eof |

LL(1) Parsing Example

$S ::= a\ S\ b \mid \varepsilon$

S

Remaining Input:
a a a b b b

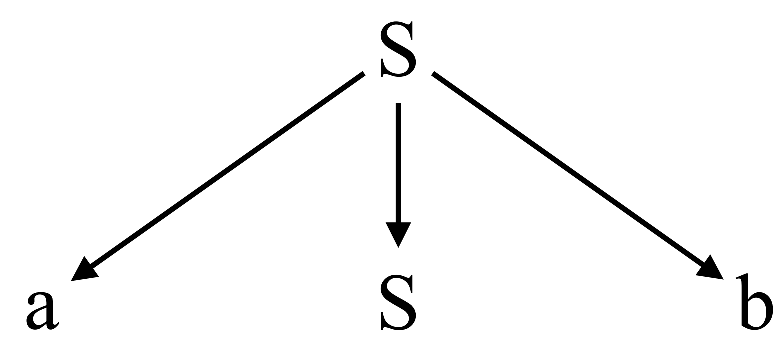
S

Sentential Form:
S

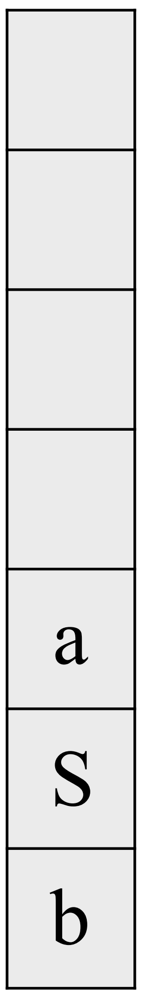
Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



Remaining Input:
a a a b b b

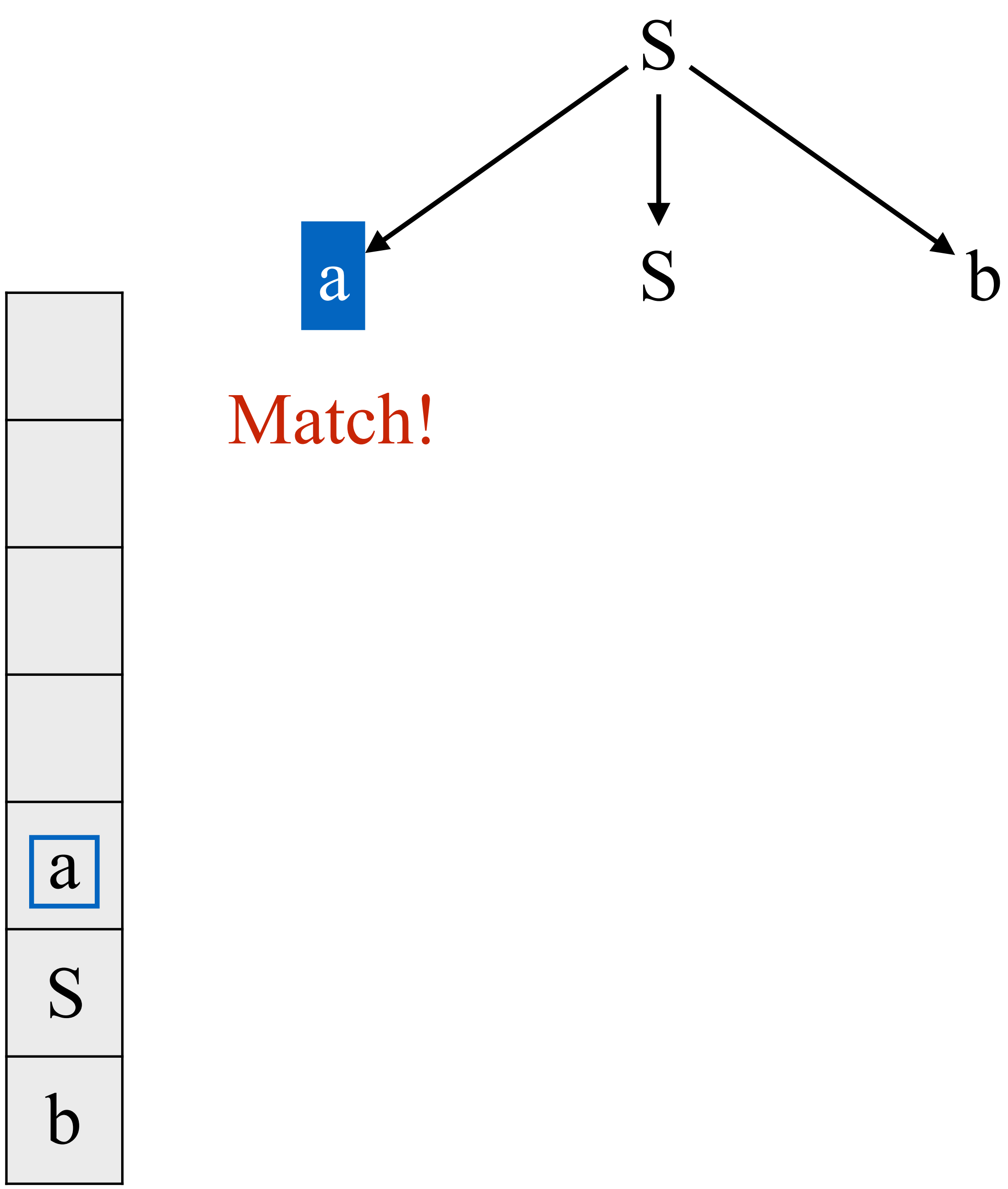


Sentential Form:
a S b

Applied Production:
S ::= a S b

LL(1) Parsing Example

$S ::= a\ S\ b \mid \varepsilon$



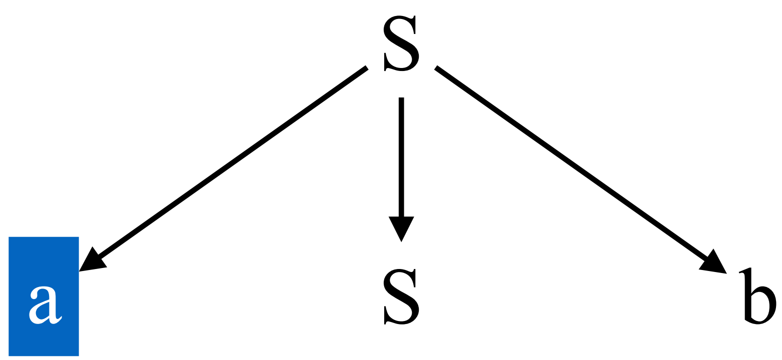
Remaining Input:
$\boxed{a}$ a a b b b

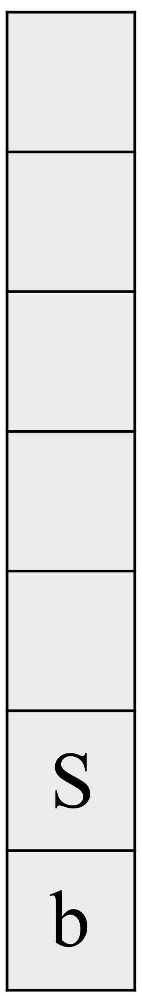
Sentential Form:
a S b

Applied Production:

LL(1) Parsing Example

$S ::= a\ S\ b \mid \varepsilon$





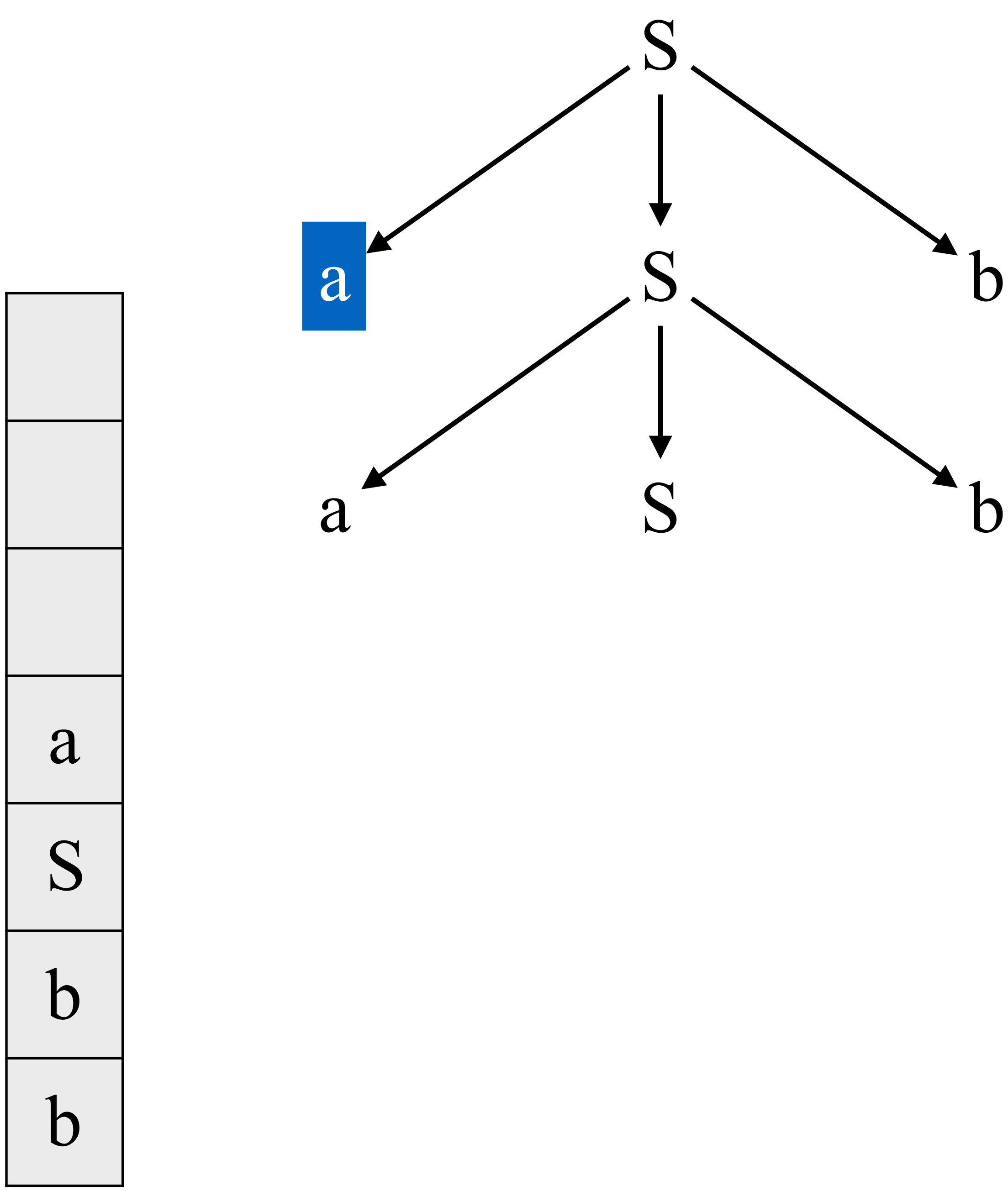
Remaining Input:
a a b b b

Sentential Form:
a S b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



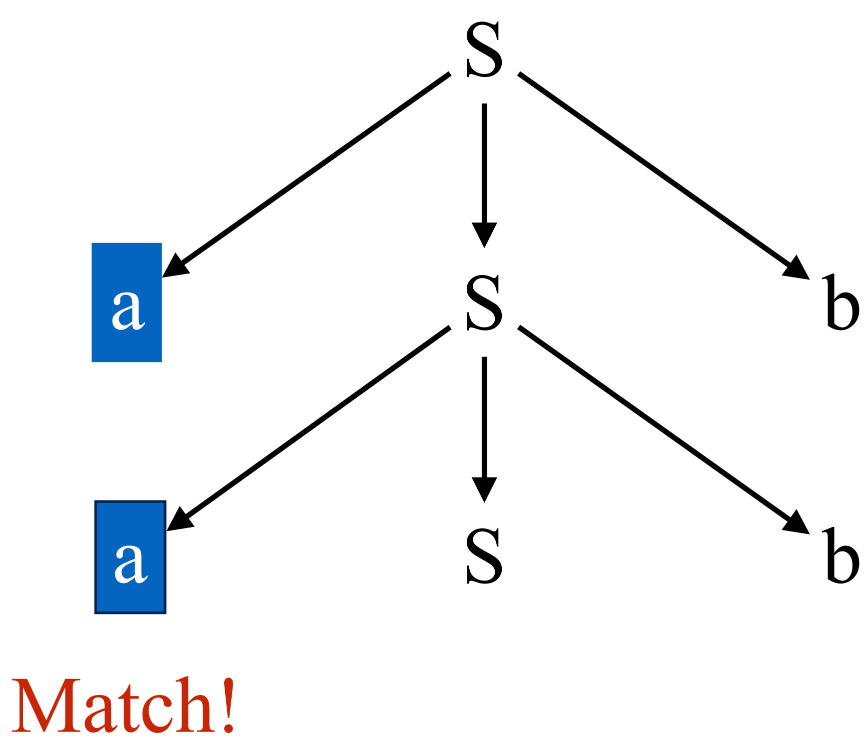
Remaining Input:
a a b b b

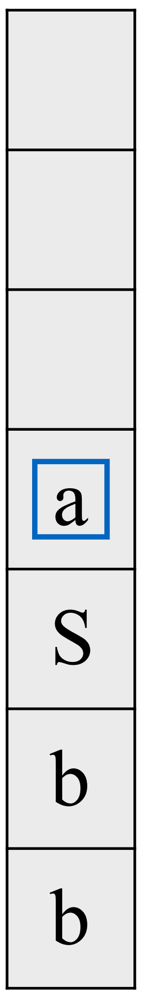
Sentential Form:
a a S b b

Applied Production:
S ::= a S b

LL(1) Parsing Example

S ::= a S b | ε





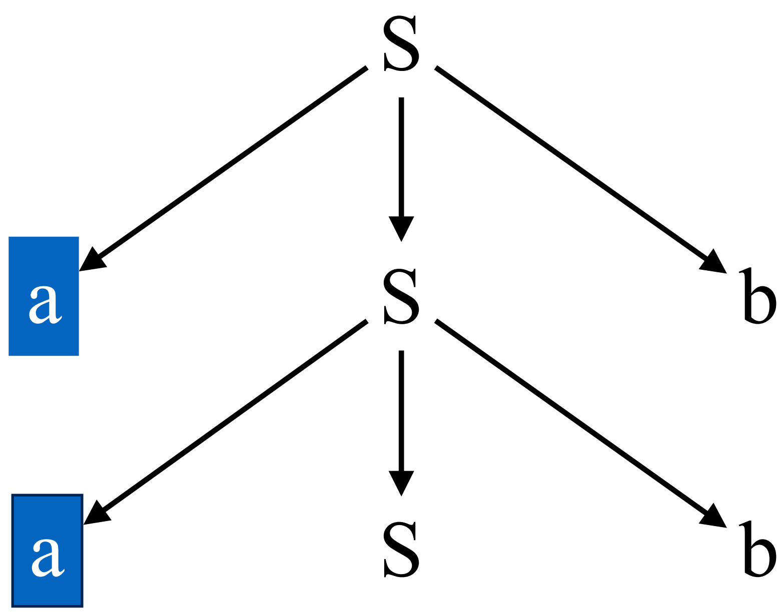
Remaining Input:
a a b b b

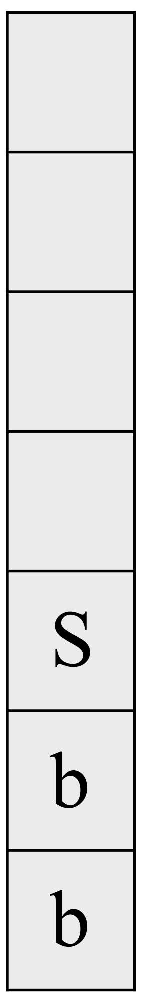
Sentential Form:
a a S b b

Applied Production:

LL(1) Parsing Example

$$S ::= a\ S\ b \mid \varepsilon$$





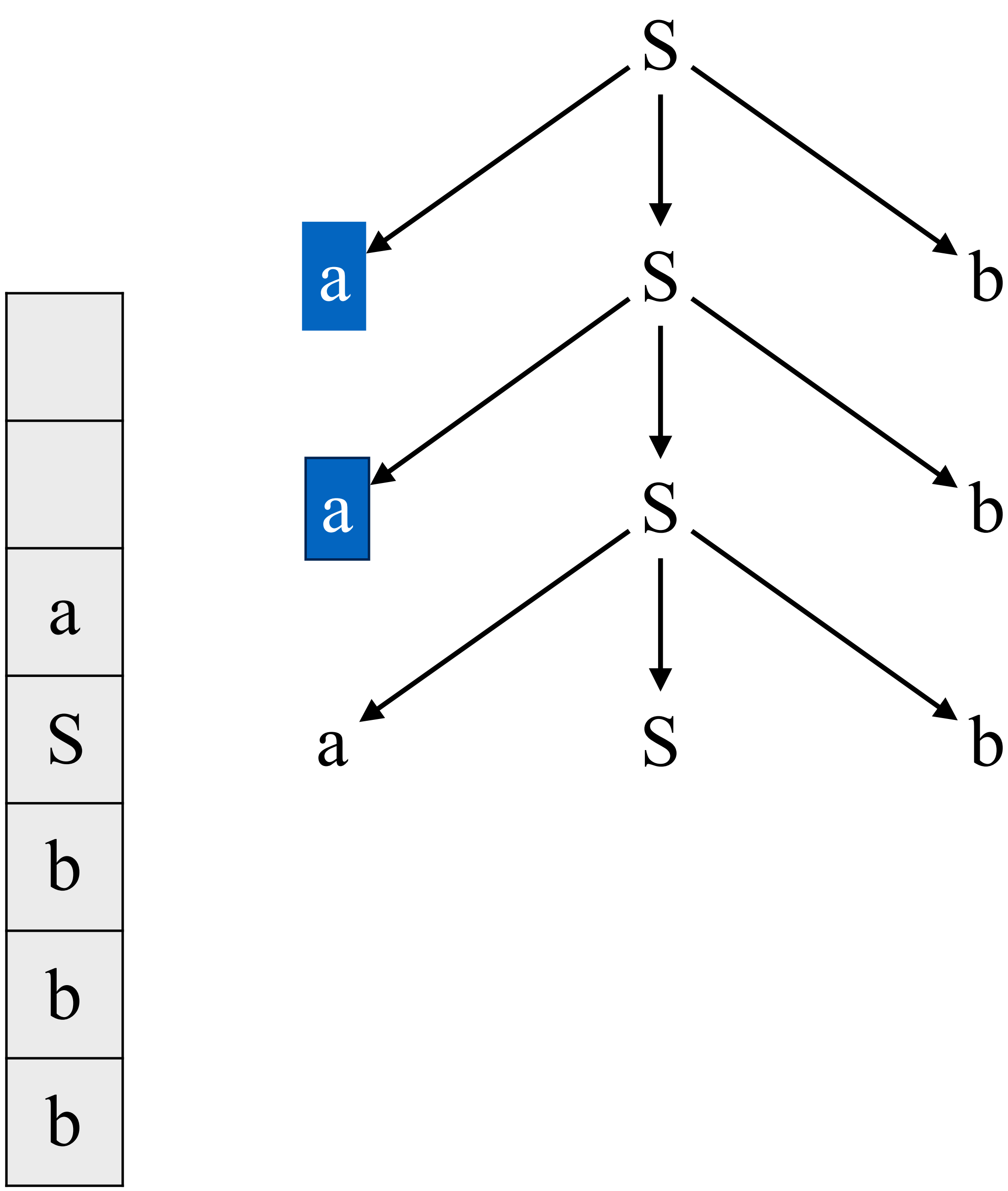
Remaining Input:
a b b b

Sentential Form:
a a S b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



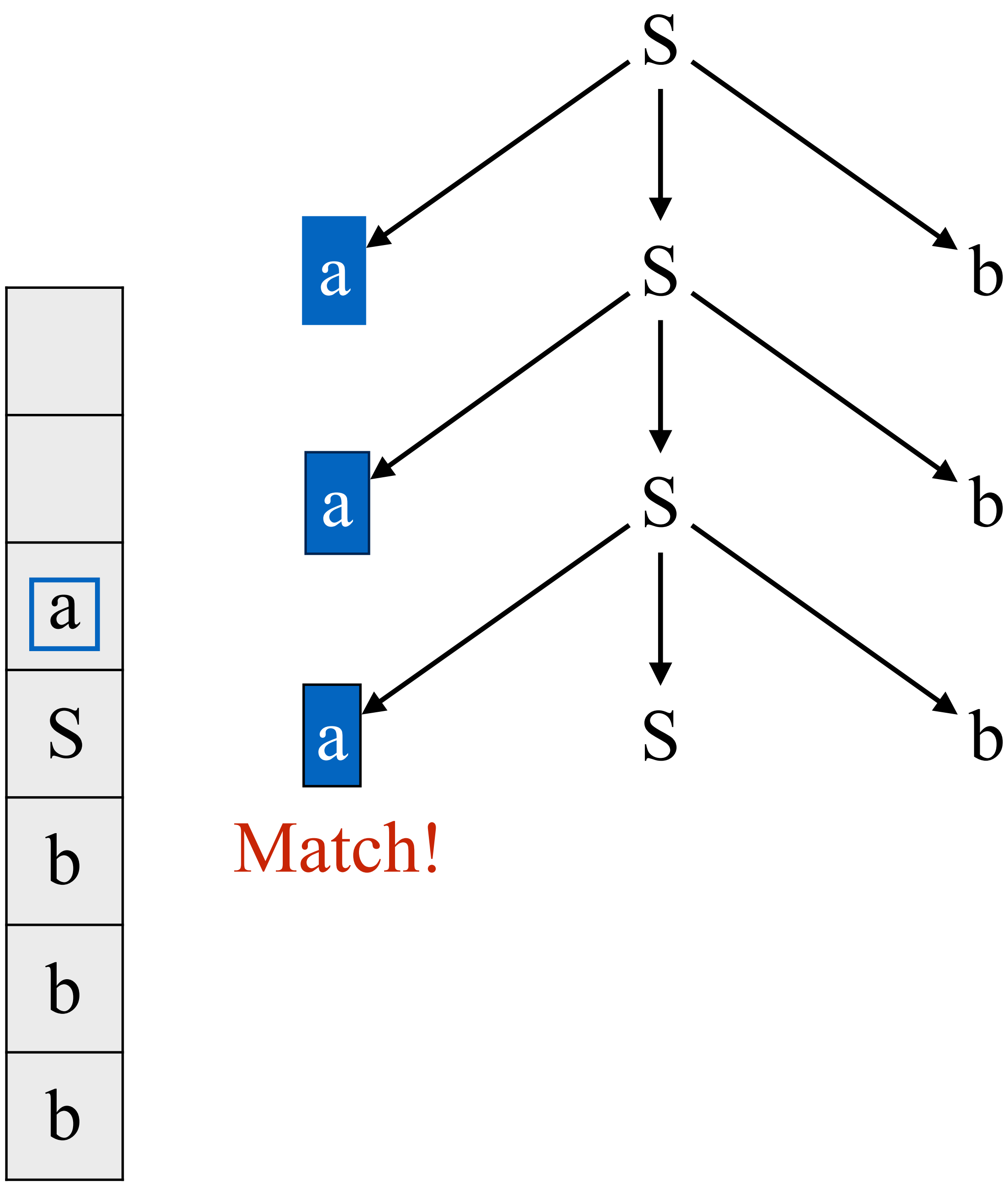
Remaining Input:
a b b b

Sentential Form:
a a a S b b b

Applied Production:
S ::= a S b

LL(1) Parsing Example

S ::= a S b | ε



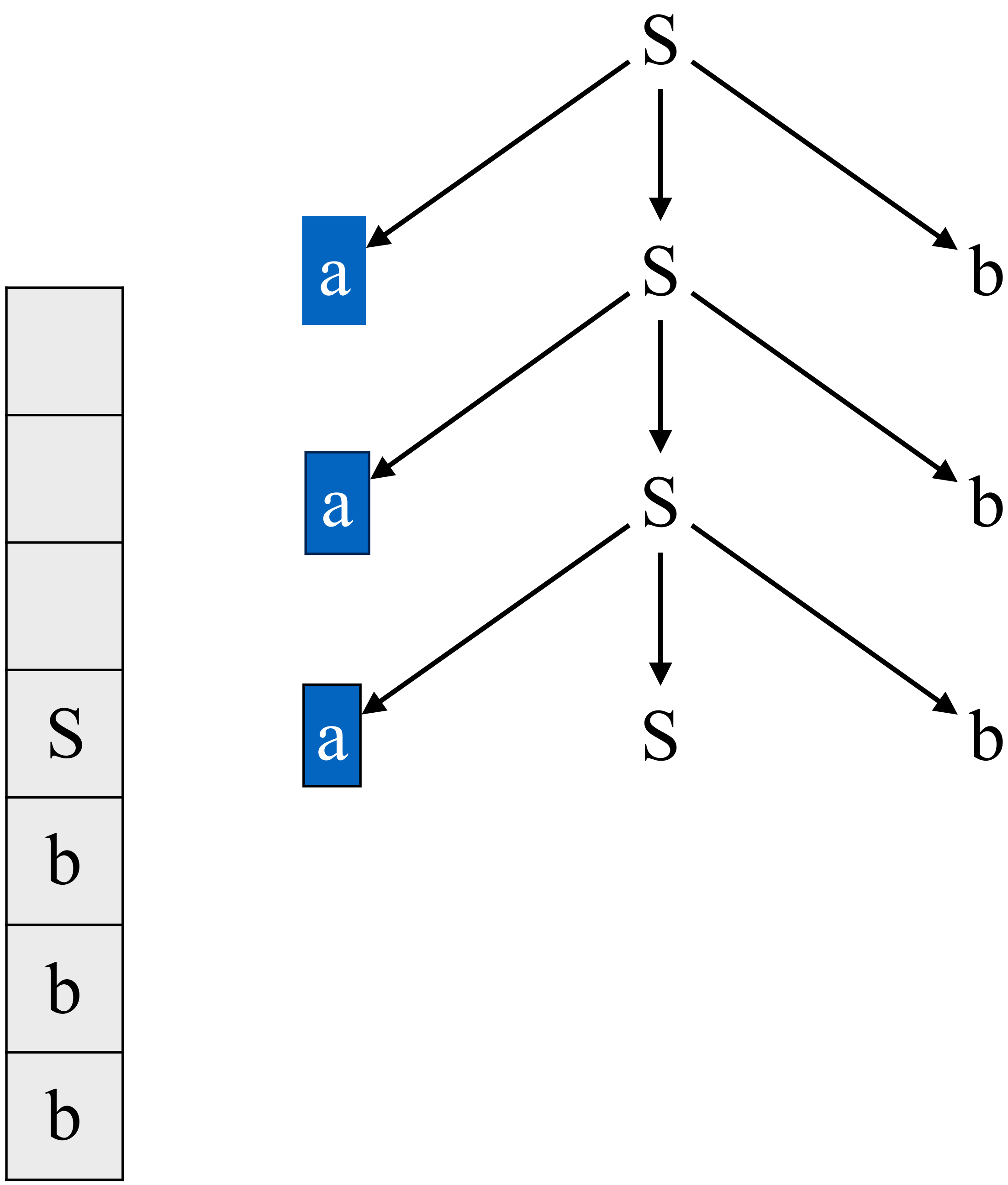
Remaining Input:
a b b b

Sentential Form:
a a a S b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



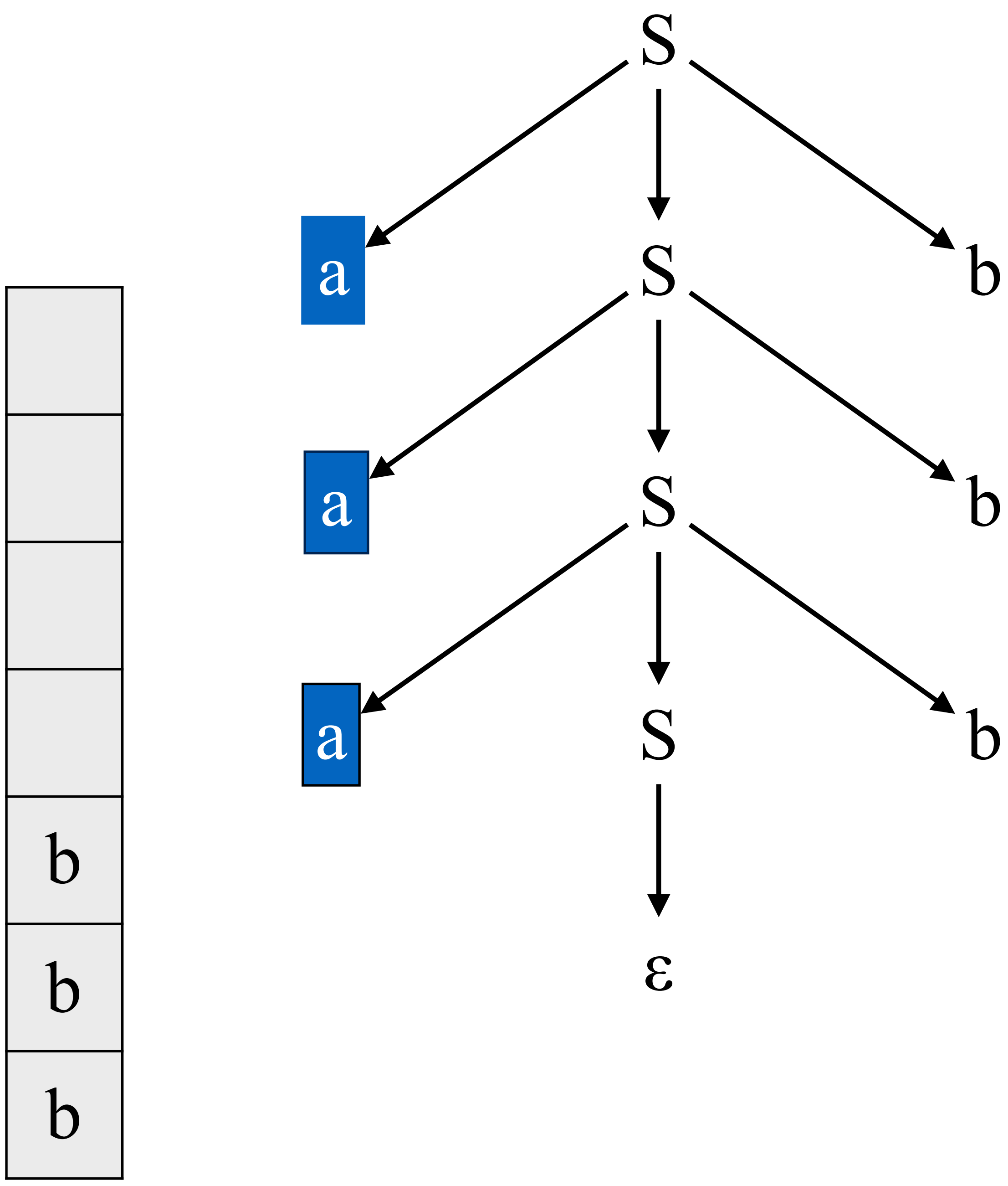
Remaining Input:
b b b

Sentential Form:
a a a S b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



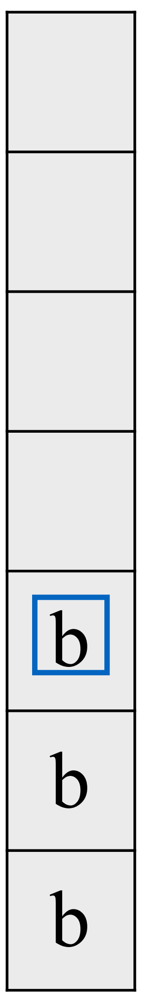
Remaining Input:
b b b

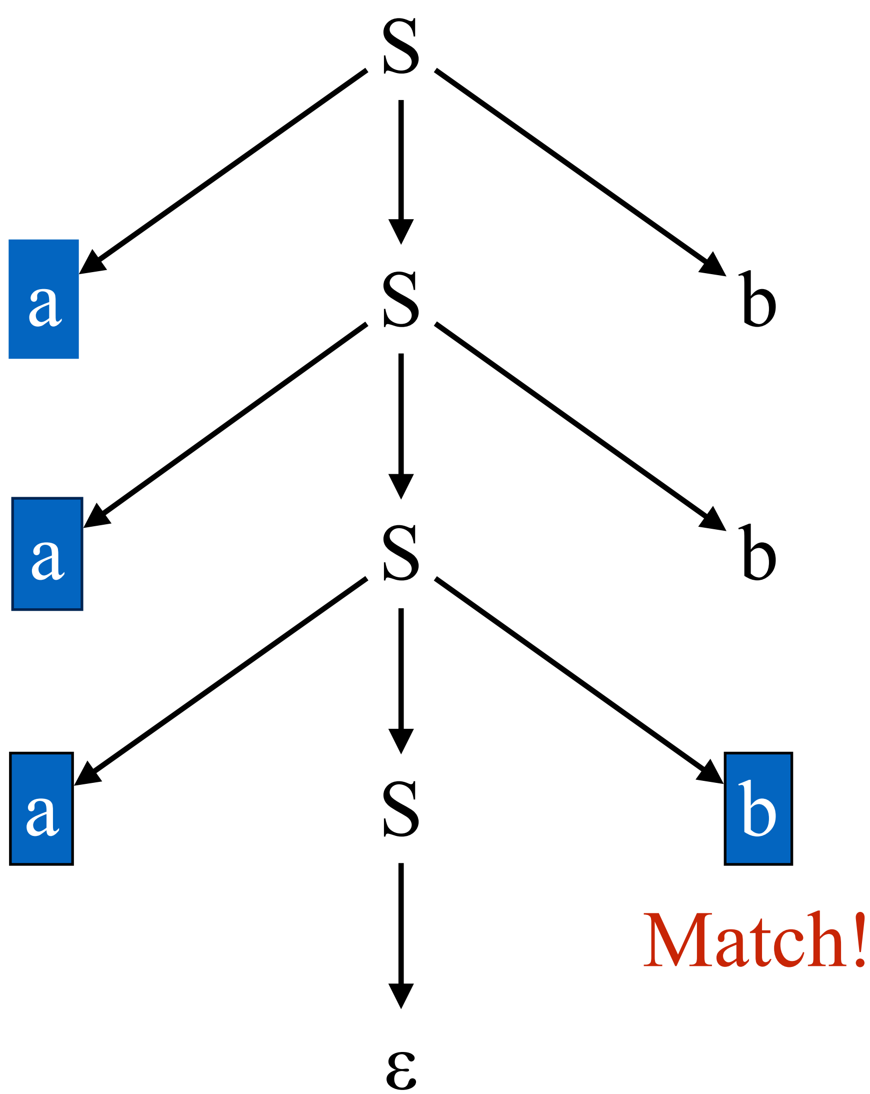
Sentential Form:
a a a b b b

Applied Production:
S ::= ε

LL(1) Parsing Example

S ::= a S b | ε





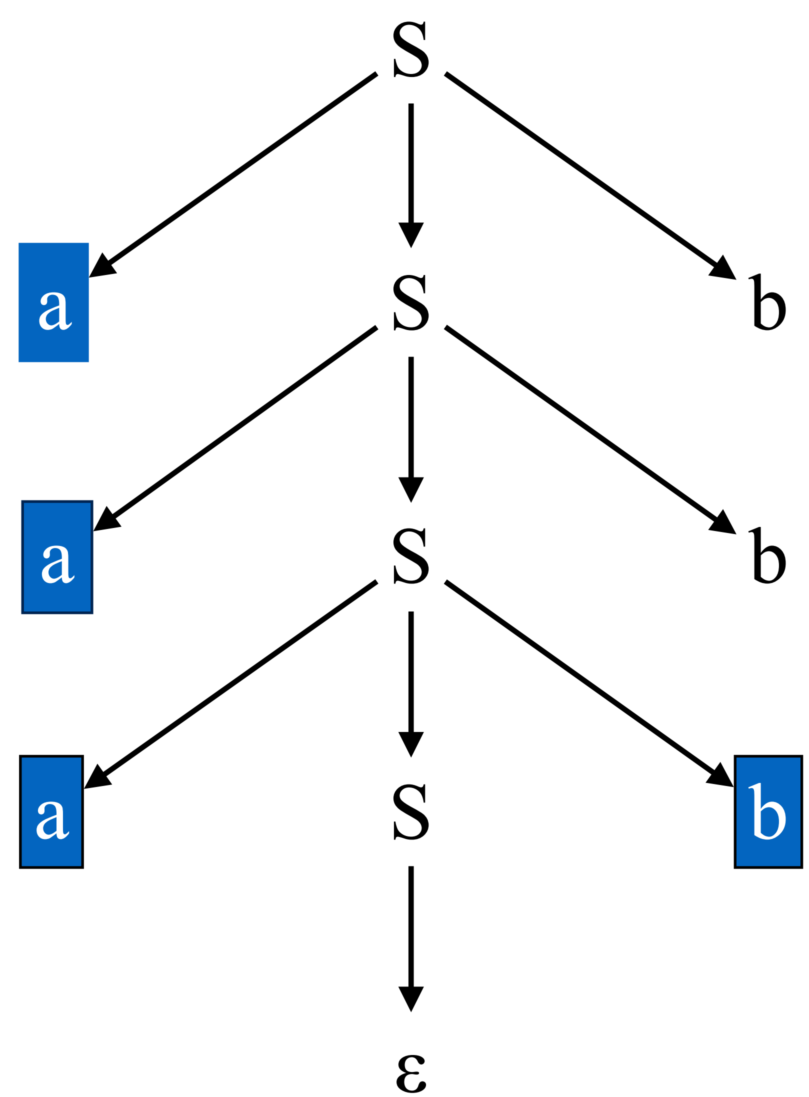
Remaining Input:
b b b

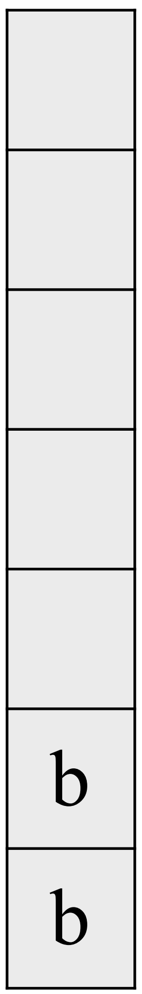
Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε





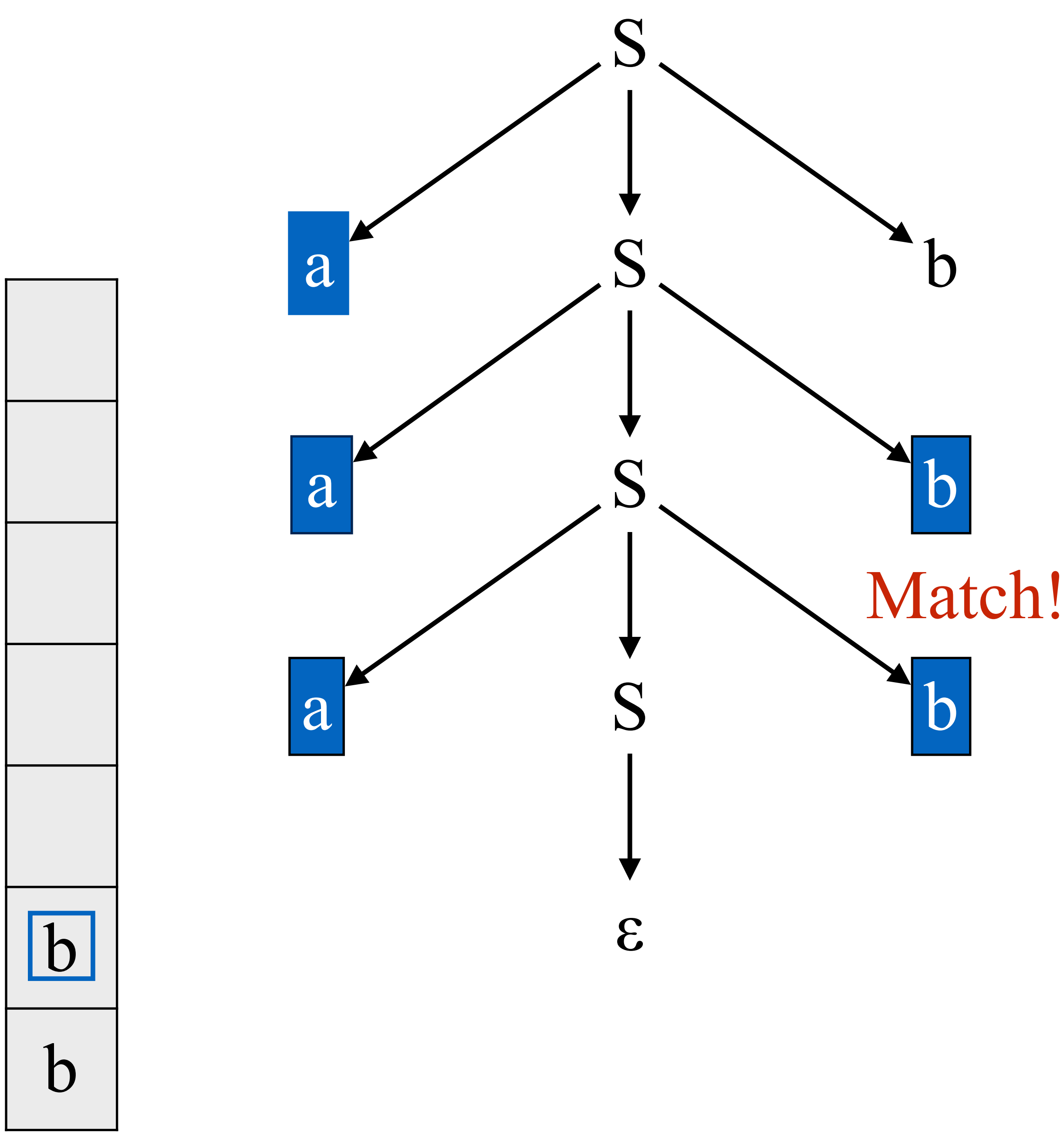
Remaining Input:
b b

Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



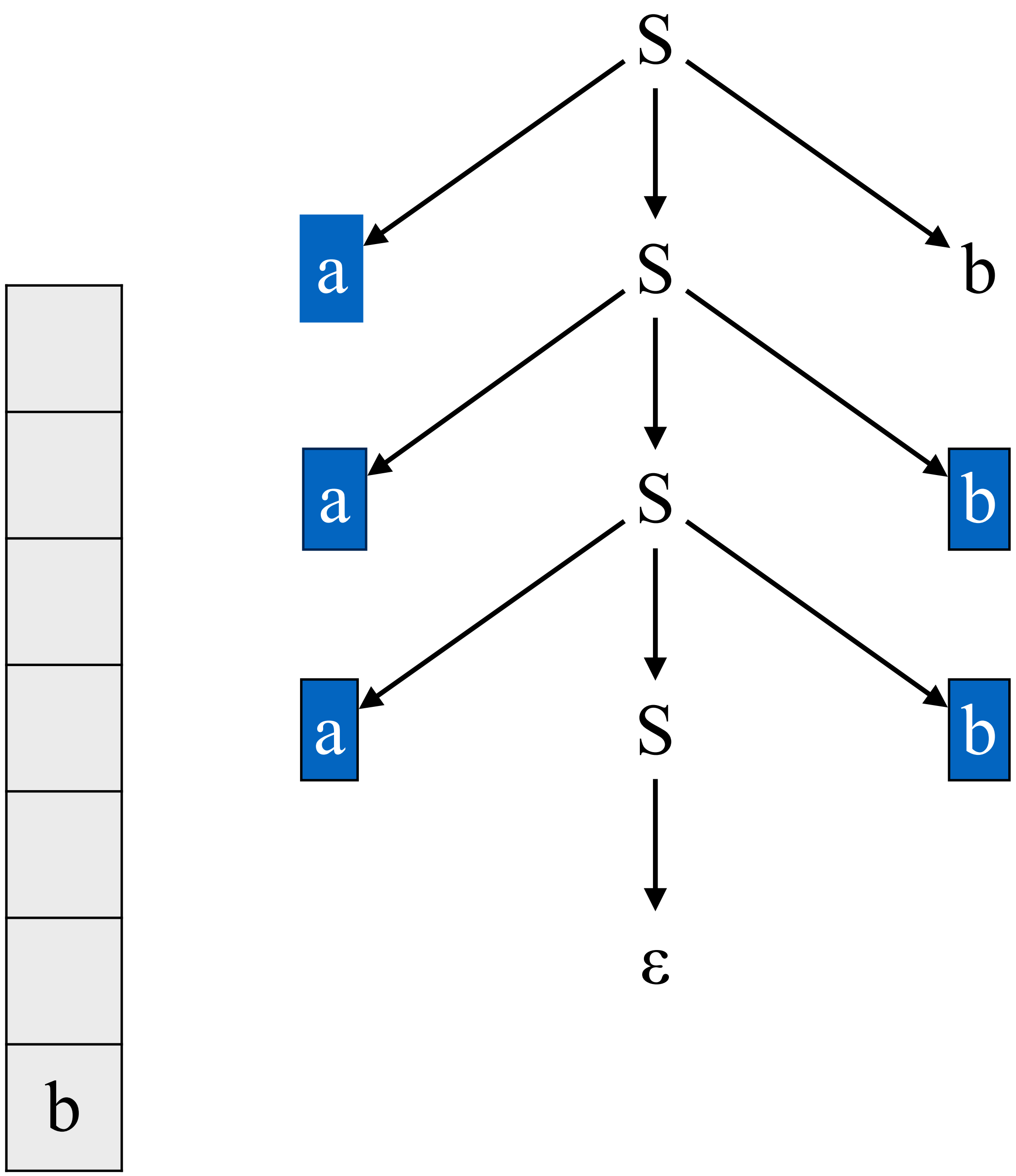
Remaining Input:
$\boxed{b}$ b

Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



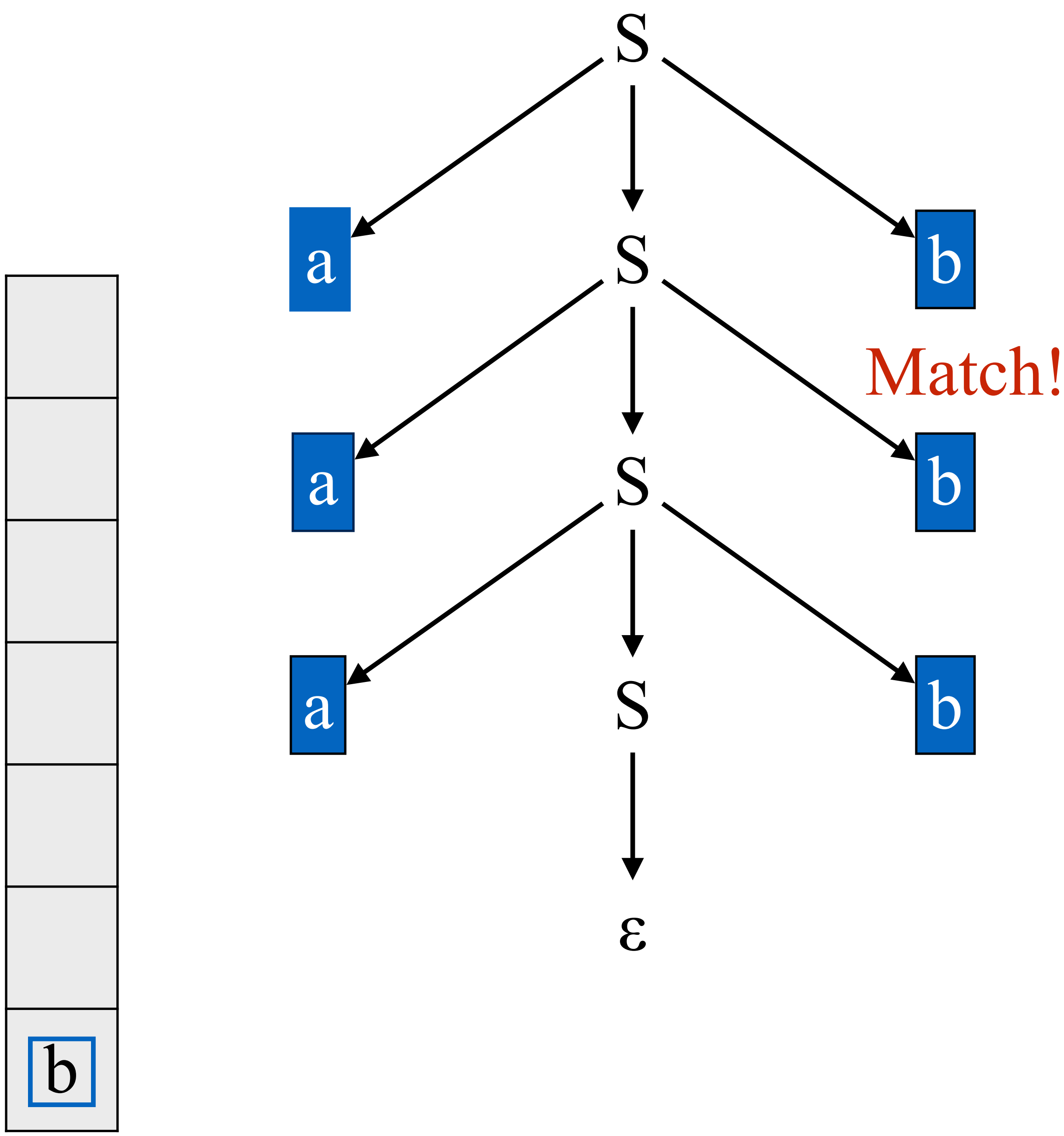
Remaining Input:
b

Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



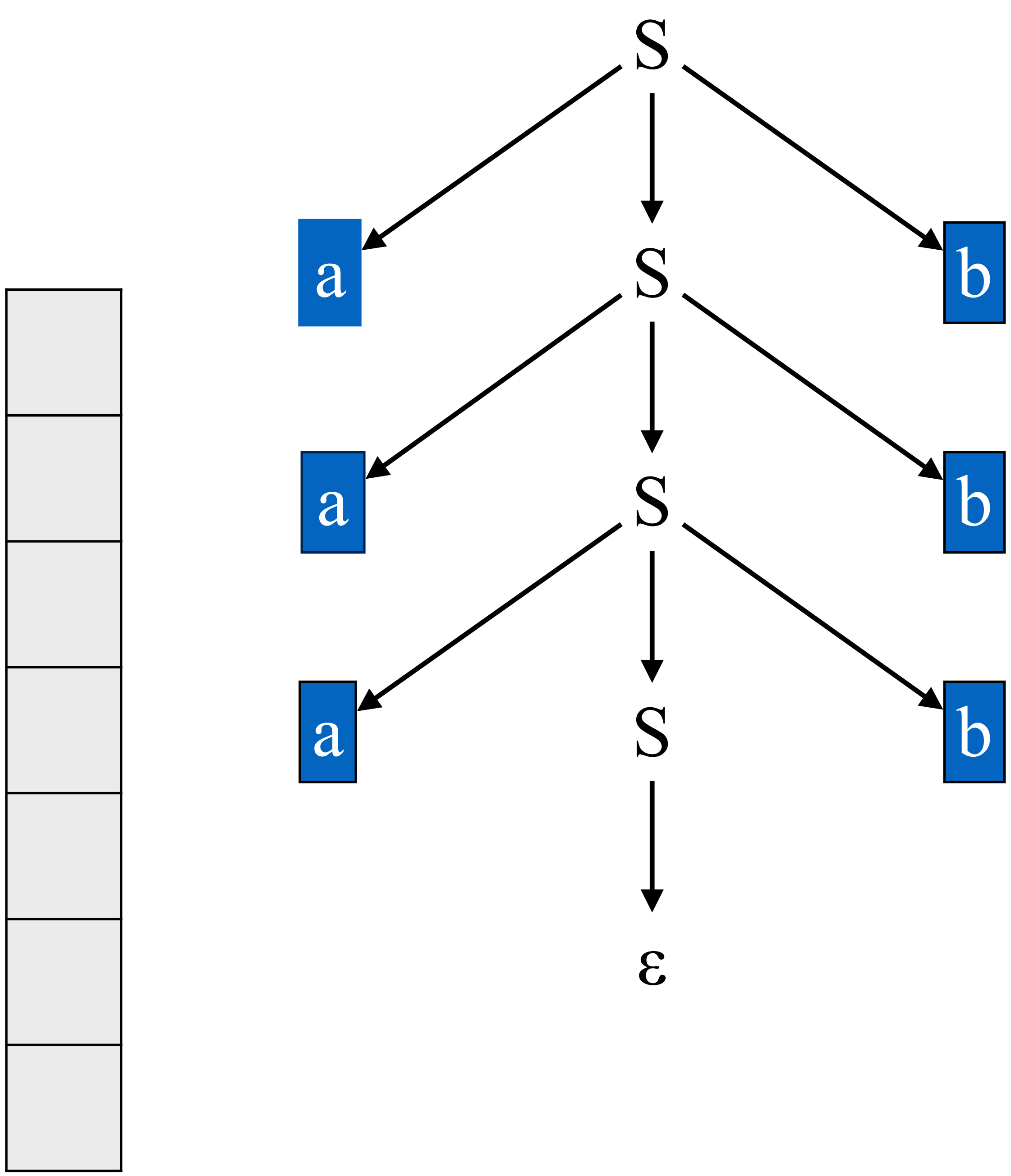
Remaining Input:
b

Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε



Remaining Input:

Sentential Form:
a a a b b b

Applied Production:

Another LL(1) Parsing Example

Consider this example grammar:

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

How to parse the following input string?

A, B, C;

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

id_list

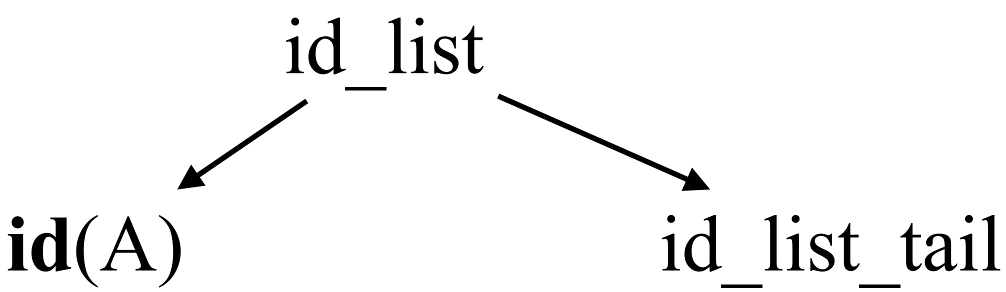
Remaining Input:
A, B, C;

Sentential Form:
id_list

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**



Remaining Input:
A, B, C;

Sentential Form:
id(A) id_list_tail

Applied Production:
id_list ::= **id** id_list_tail

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**

id_list

id(A) id_list_tail

Match!

Remaining Input:
A , B , C ;

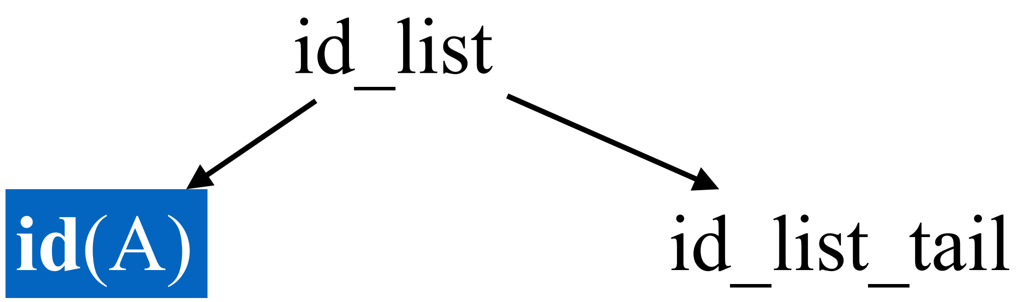
Sentential Form:
id(A) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;

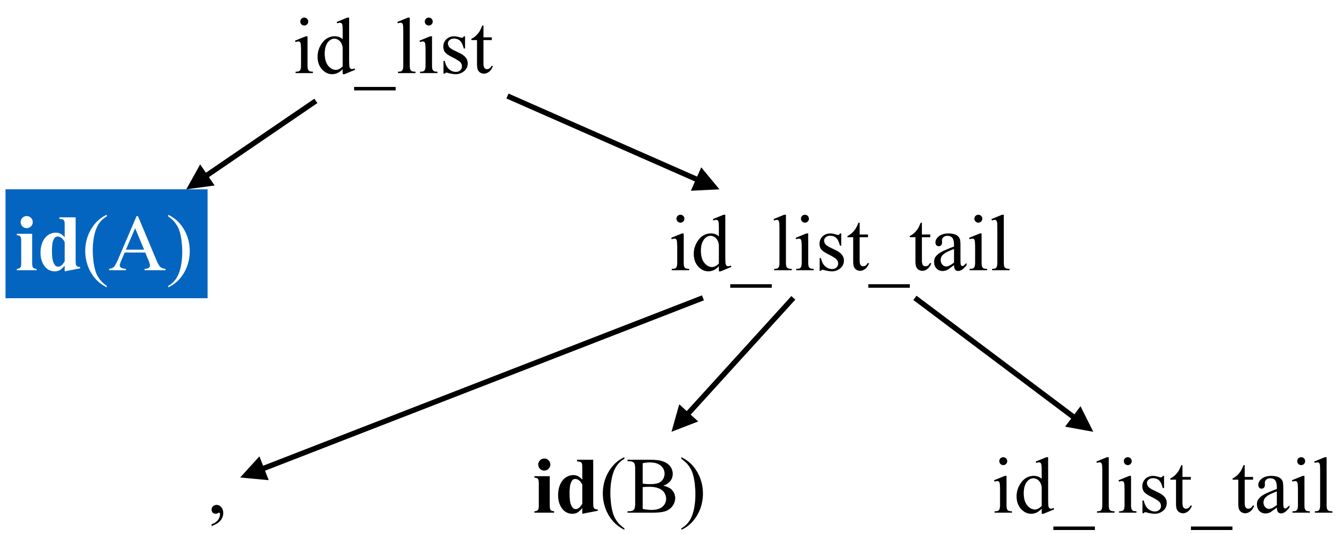


Sentential Form:
id(A) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**



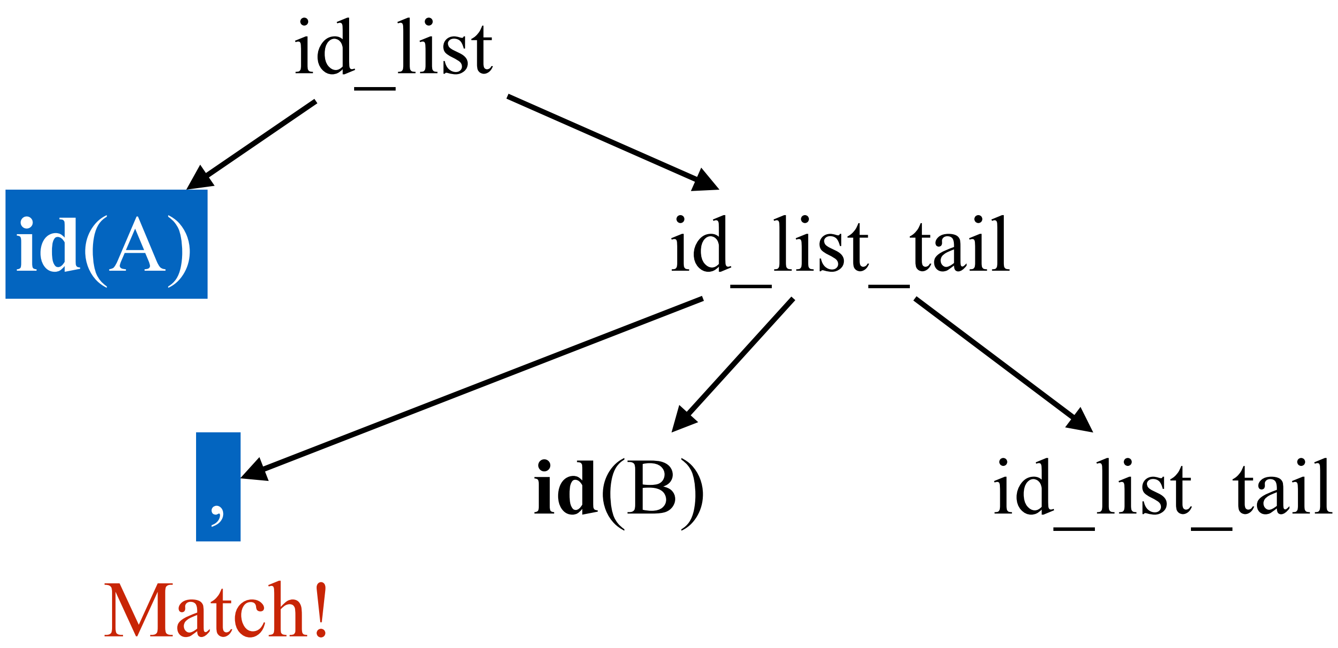
Remaining Input:
, B , C ;

Sentential Form:
id(A) , **id**(B) id_list_tail

Applied Production:
id_list_tail ::= **, id** id_list_tail

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**



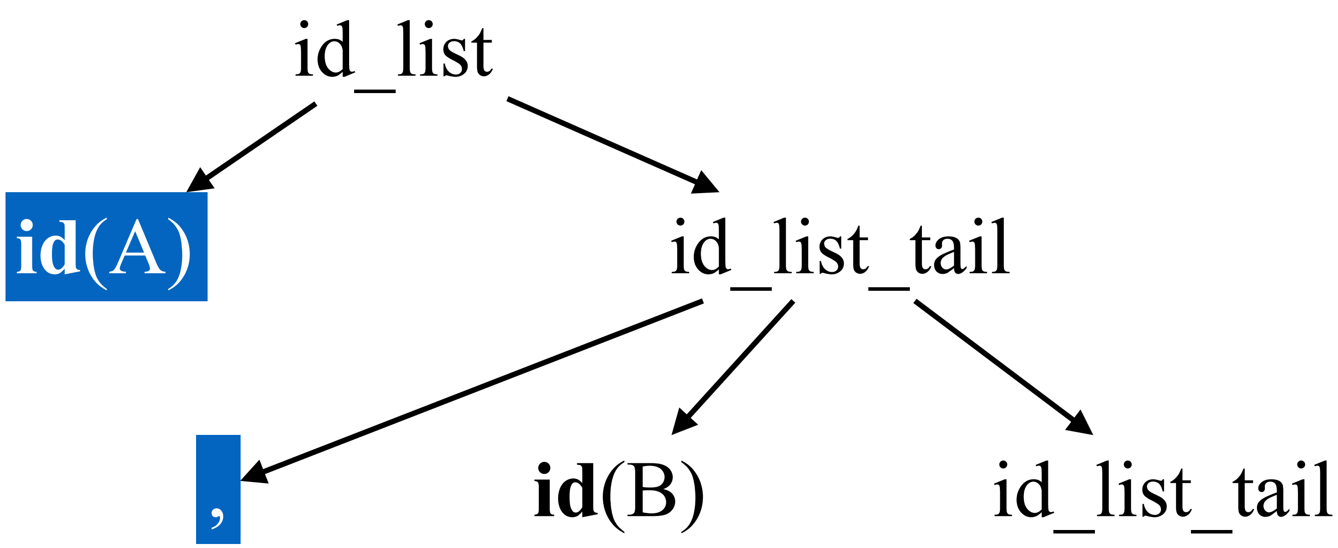
Remaining Input:
, B , C ;

Sentential Form:
id(A) , **id**(B) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**



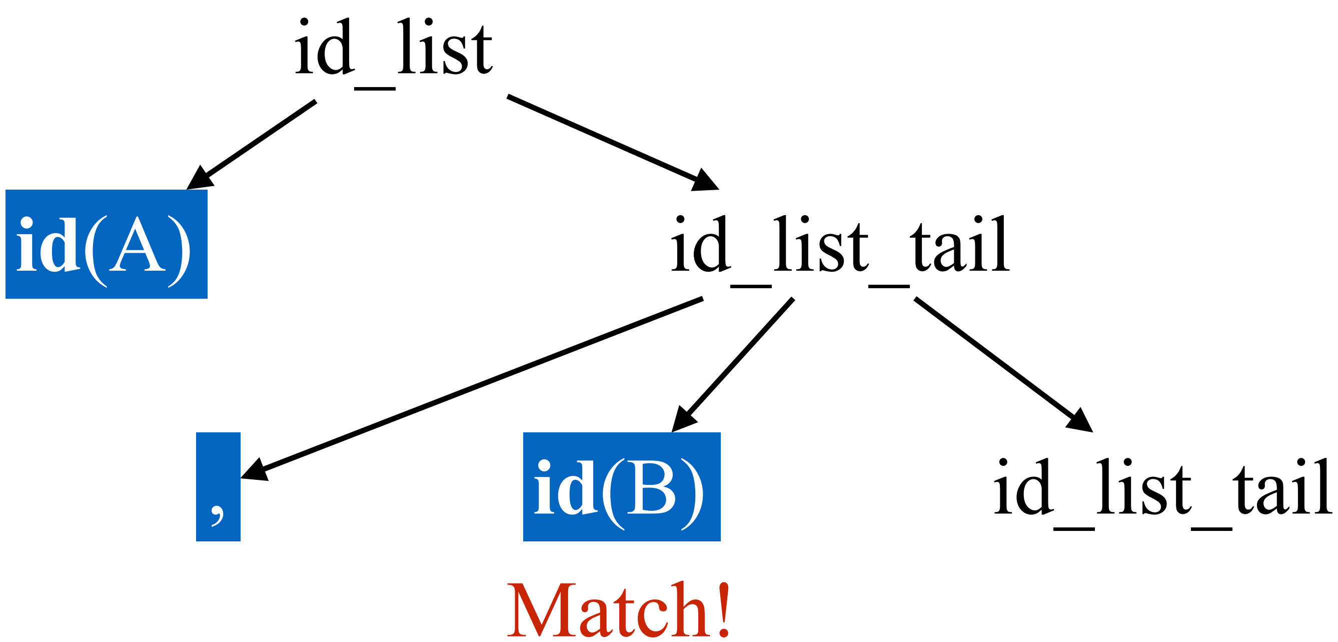
Remaining Input:
B , C ;

Sentential Form:
id(A) , **id**(B) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**



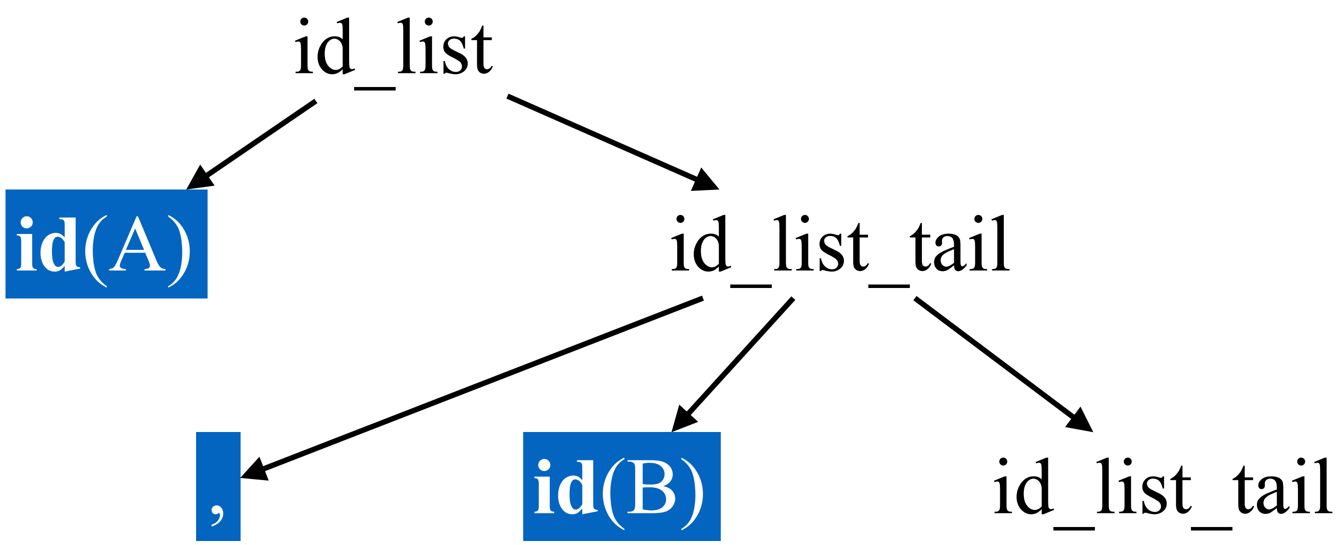
Remaining Input:
B , C ;

Sentential Form:
id(A) , **id**(B) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**



Remaining Input:
, C ;

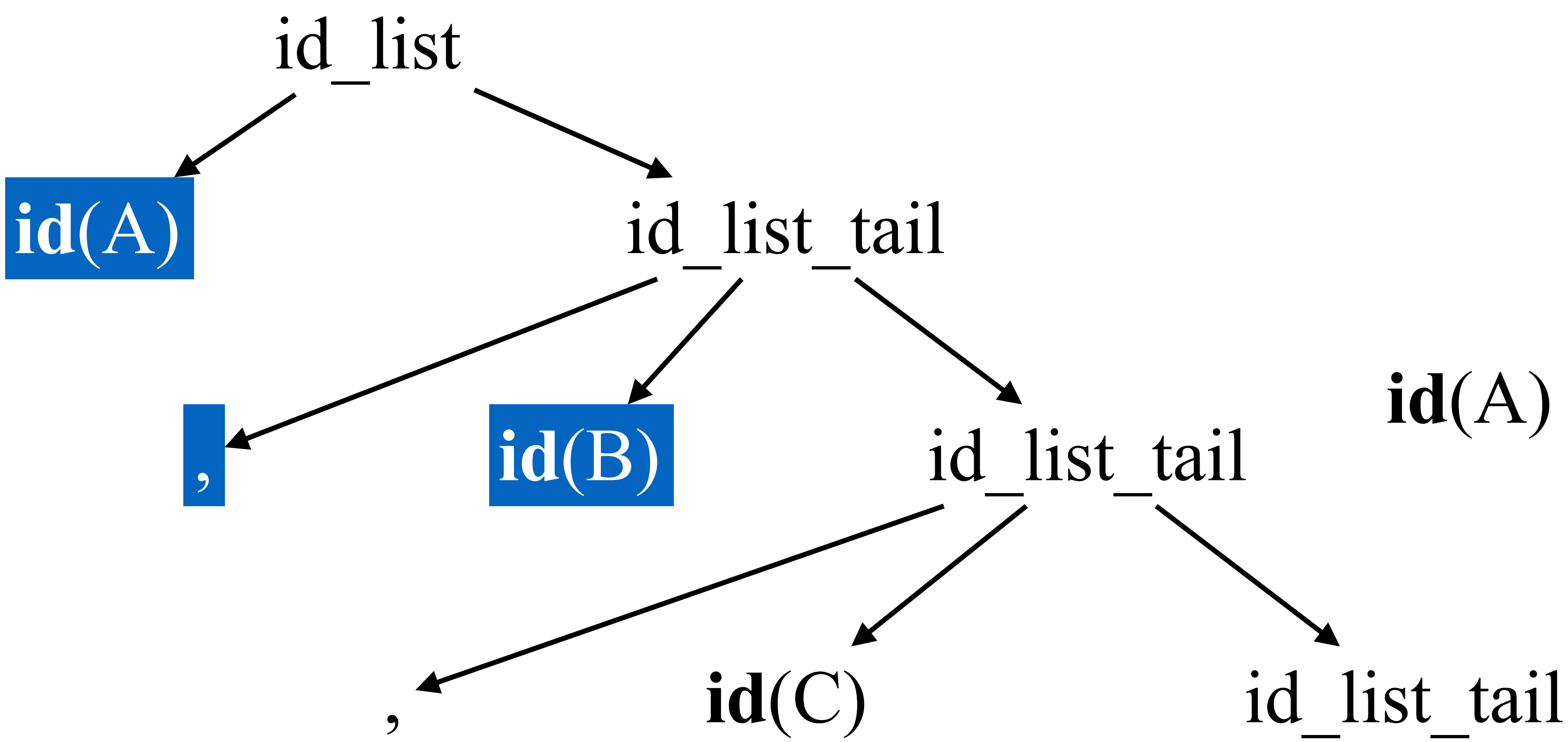
Sentential Form:
id(A) , **id**(B) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, C ;



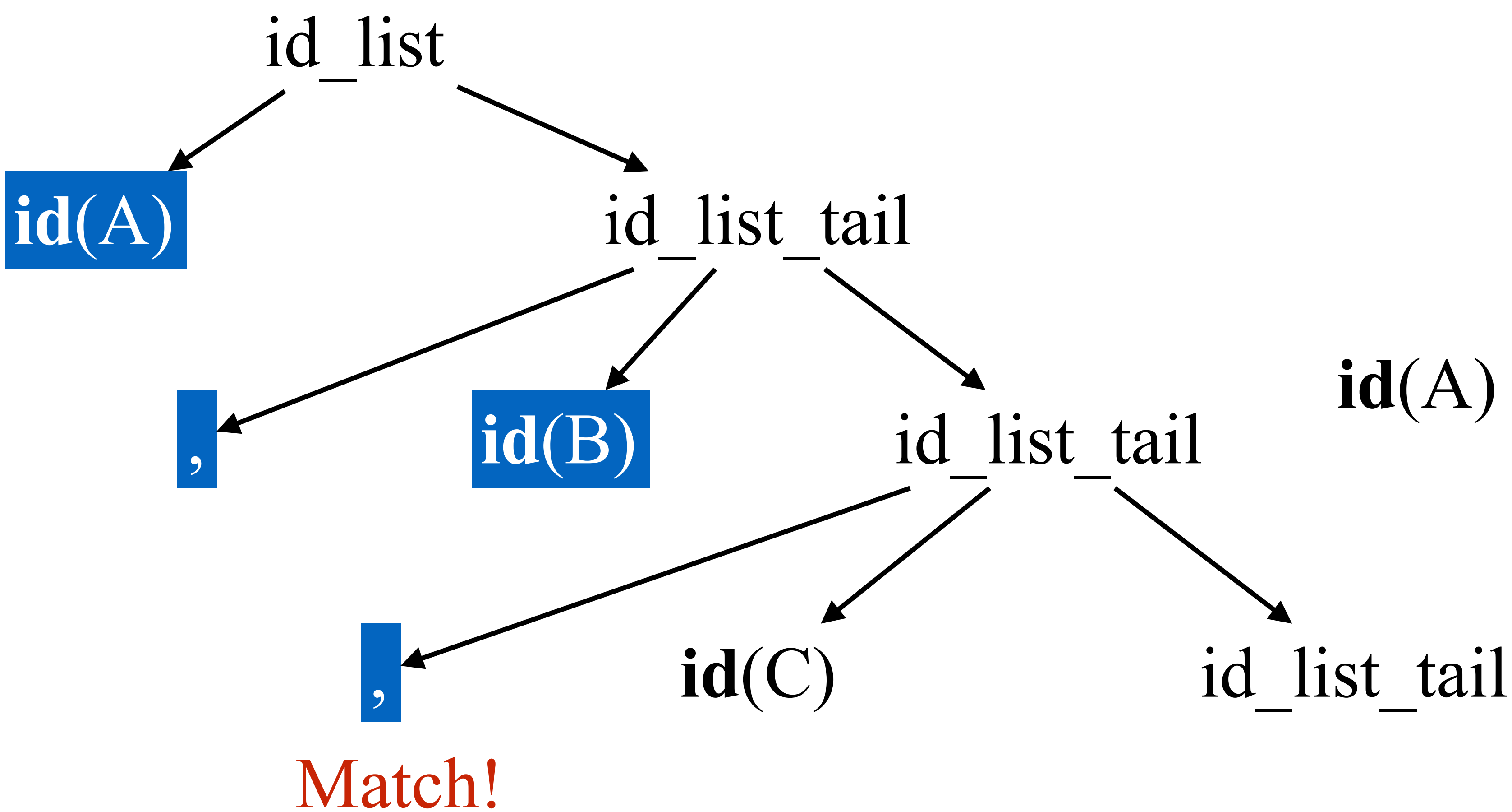
Sentential Form:
id(A) , **id**(B) , **id**(C) id_list_tail

Applied Production:
id_list_tail ::= **, id** id_list_tail

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
$\boxed{,}$ C ;



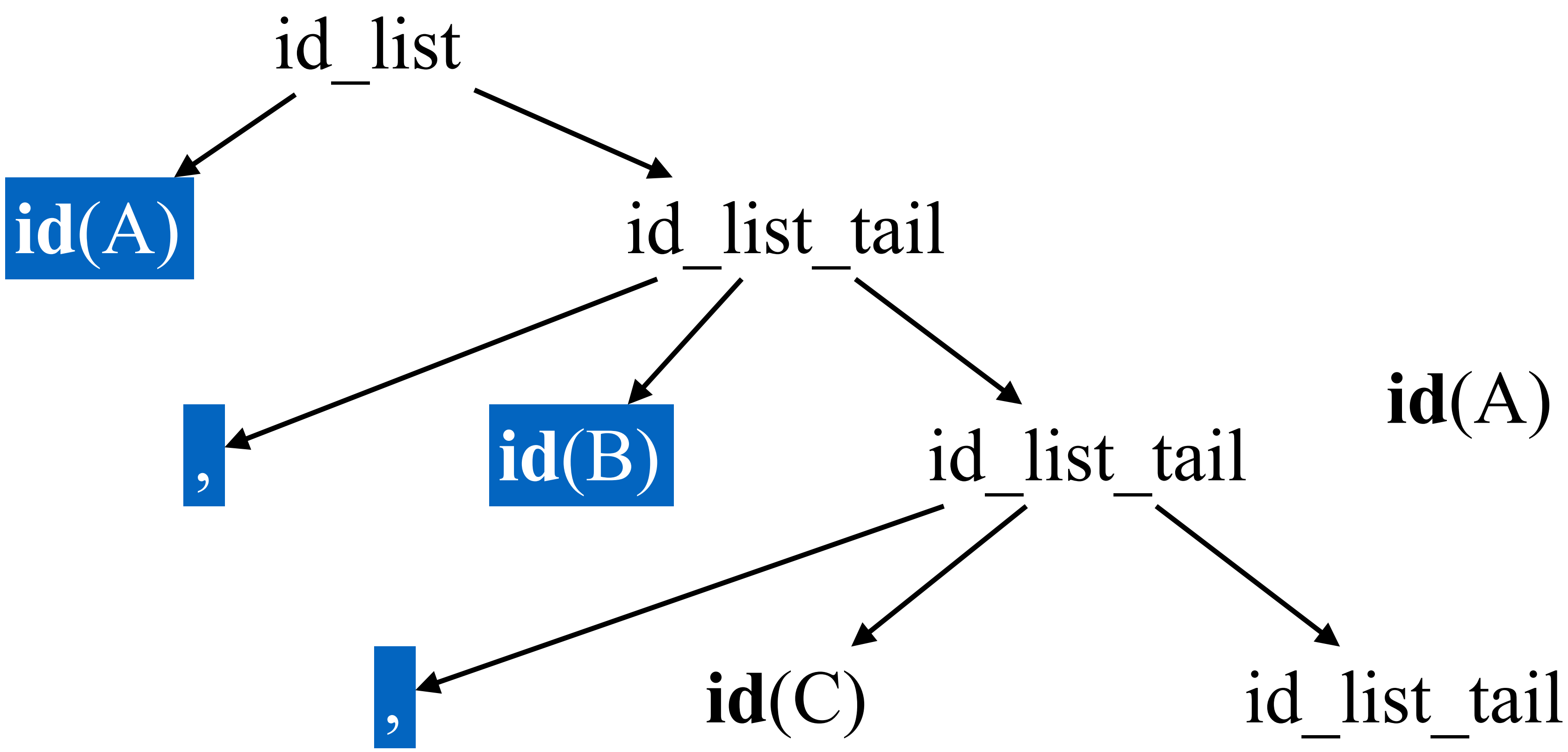
Sentential Form:
id(A) , **id**(B) , **id**(C) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
C ;



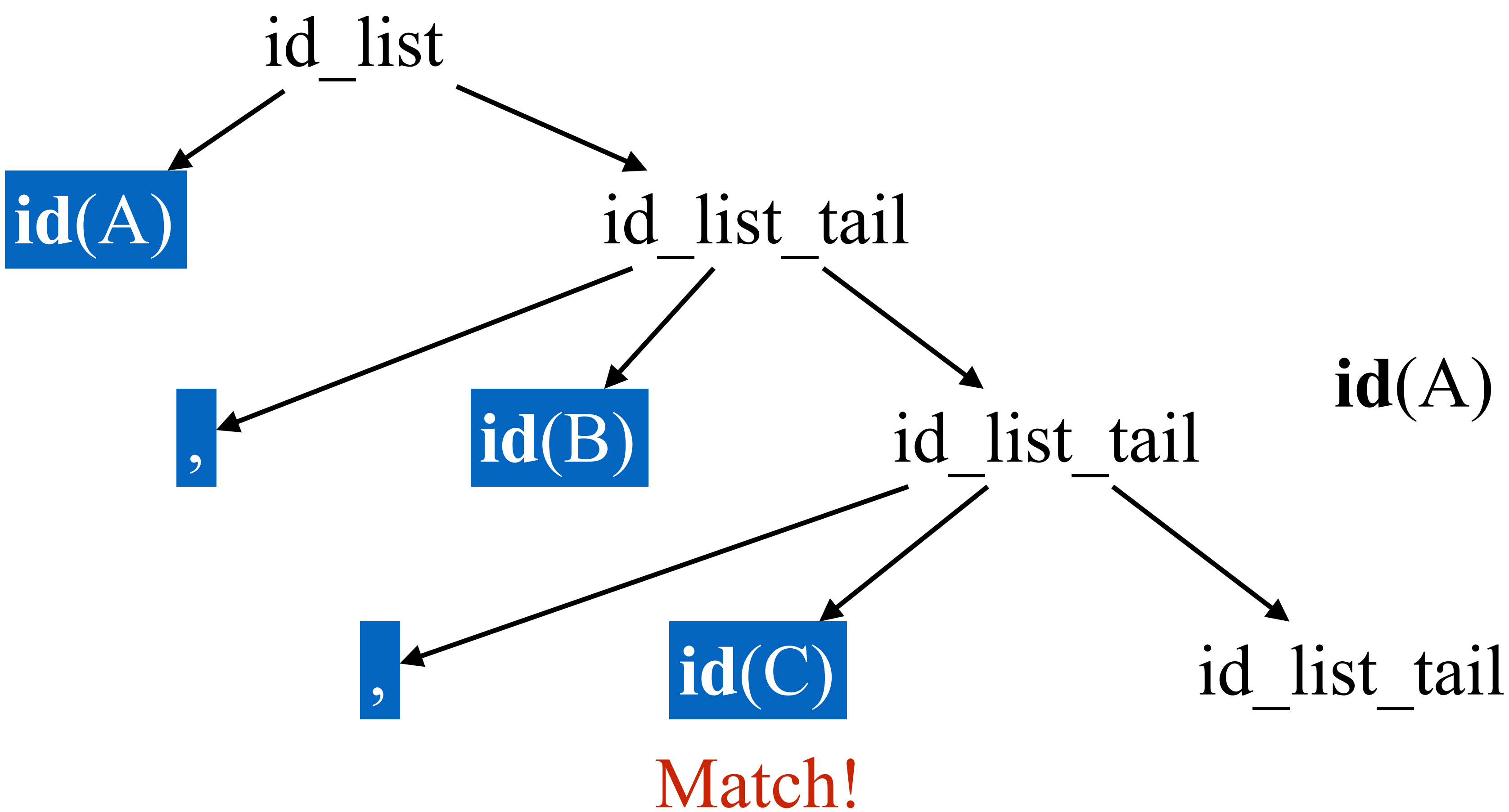
Sentential Form:
id(A) , **id**(B) , **id**(C) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
$\boxed{\text{C}}$;

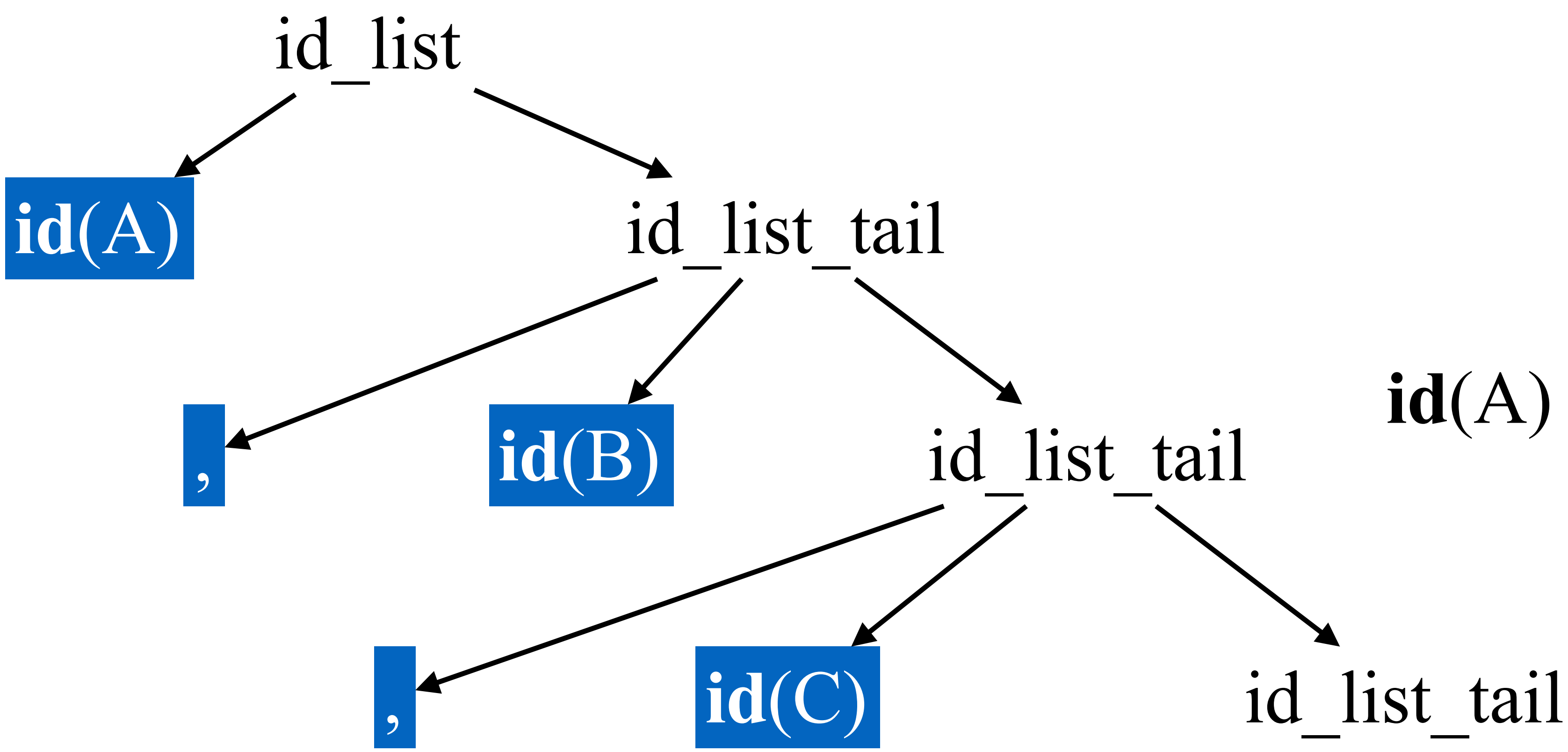


Sentential Form:
id(A) , **id**(B) , **id**(C) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**



Remaining Input:
;

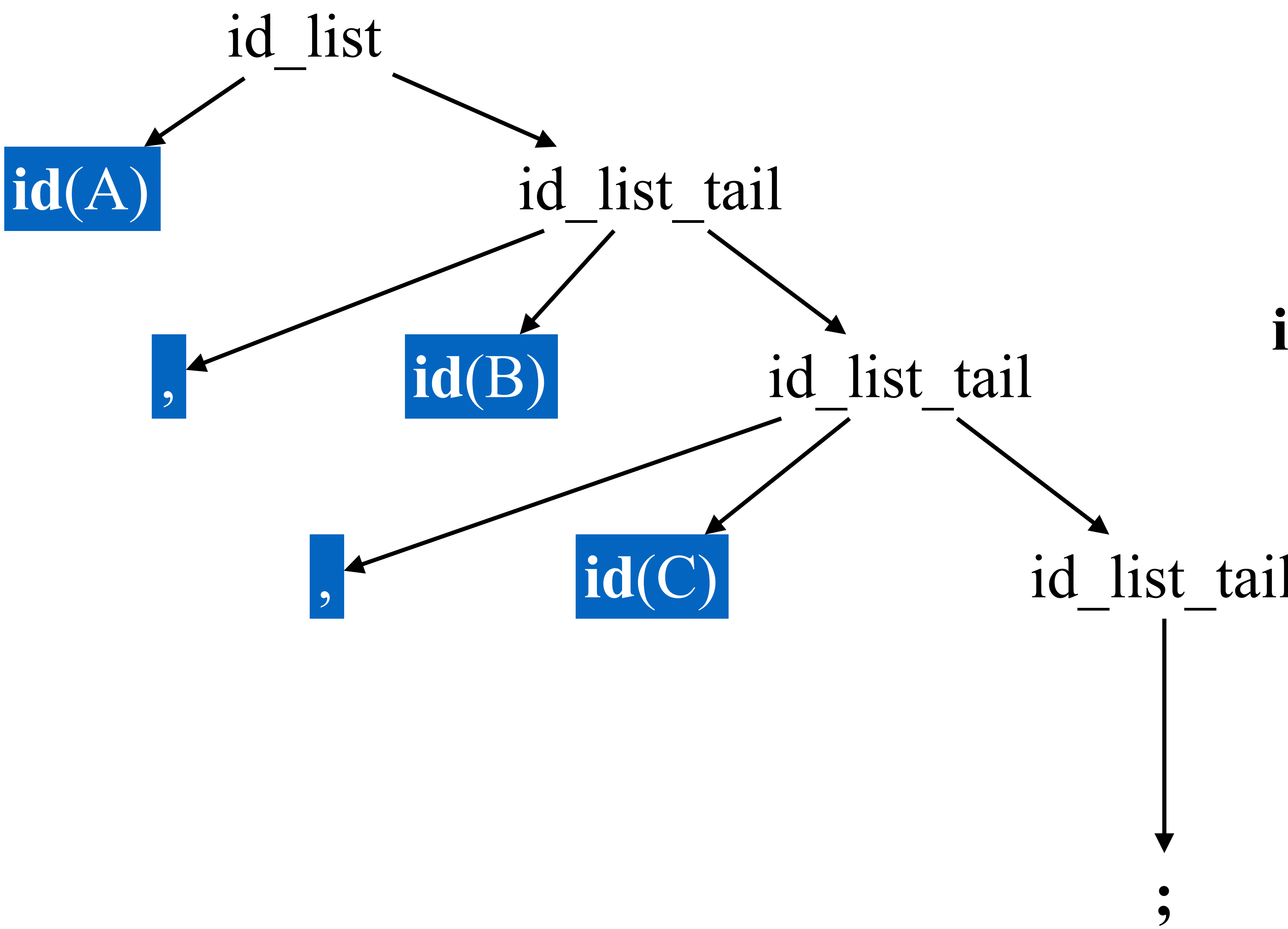
Sentential Form:
id(A) , **id**(B) , **id**(C) id_list_tail

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
;



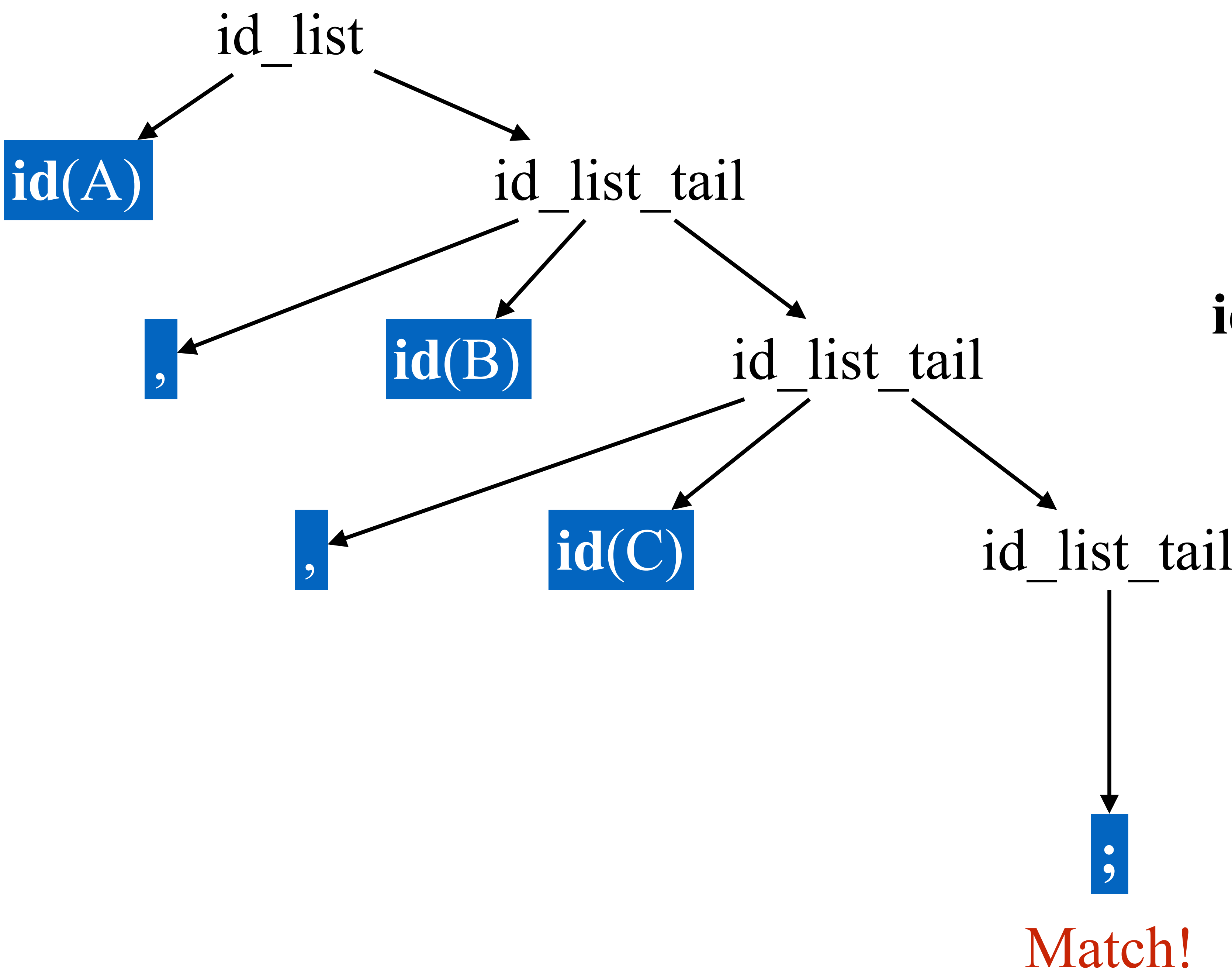
Sentential Form:
id(A) , **id**(B) , **id**(C) ;

Applied Production:
id_list_tail ::= **;**

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
;

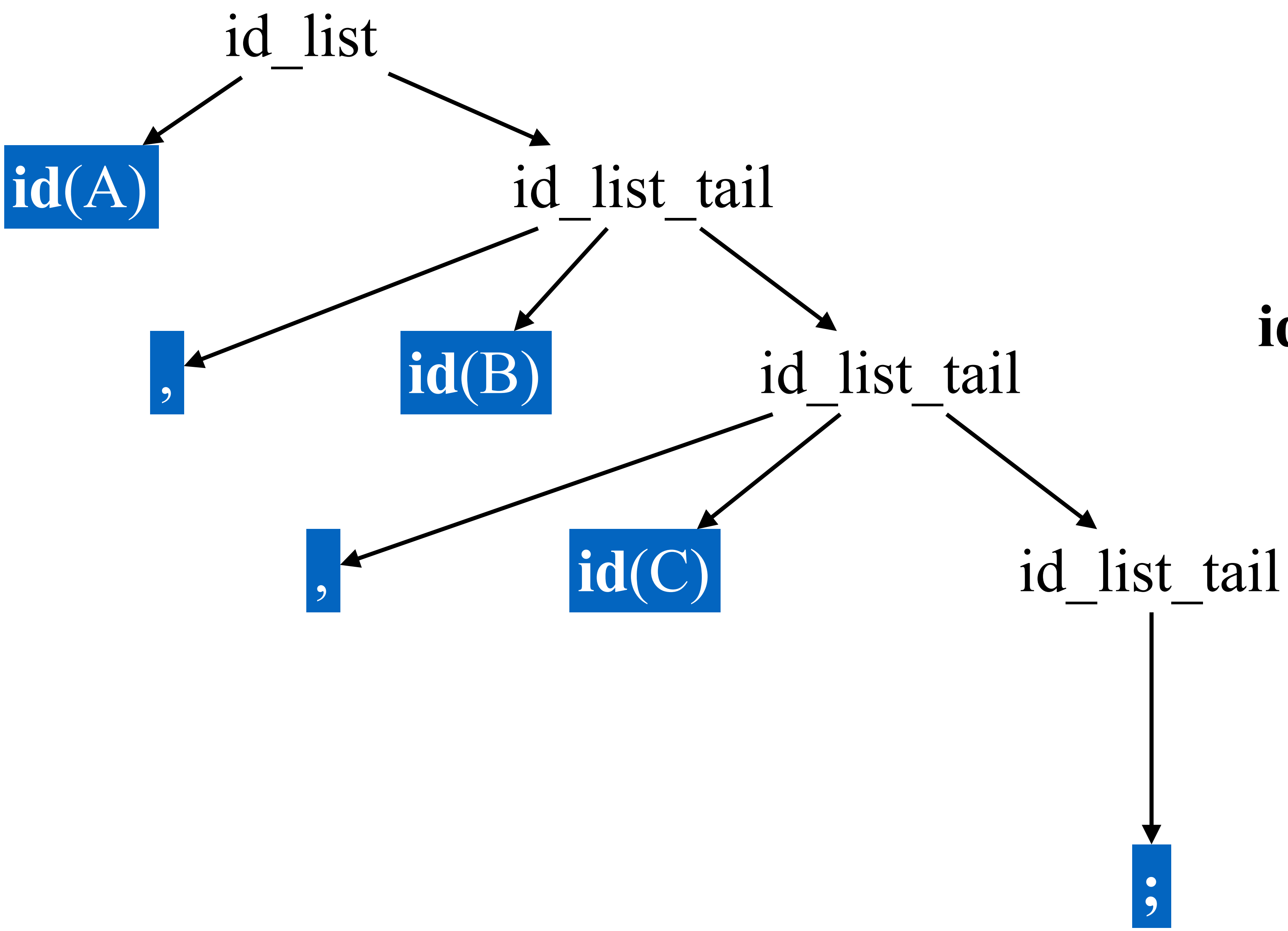


Sentential Form:
id(A) , **id**(B) , **id**(C) ;

Applied Production:

Another LL(1) Parsing Example

id_list ::= **id** id_list_tail
id_list_tail ::= **,** **id** id_list_tail
id_list_tail ::= **;**



Remaining Input:

Sentential Form:
id(A) , **id**(B) , **id**(C) ;

Applied Production:

Predictive Parsing

Basic idea:

For any two productions $A ::= \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For some rhs $\alpha \in G$, define **FIRST**(α) as the set of tokens that appear as the first symbol in some string derived from α.

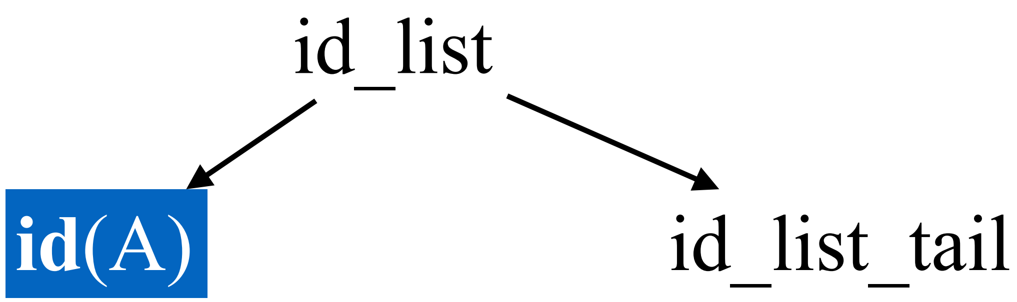
That is

$x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \mathbf{x}\gamma$ for some γ, and

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;

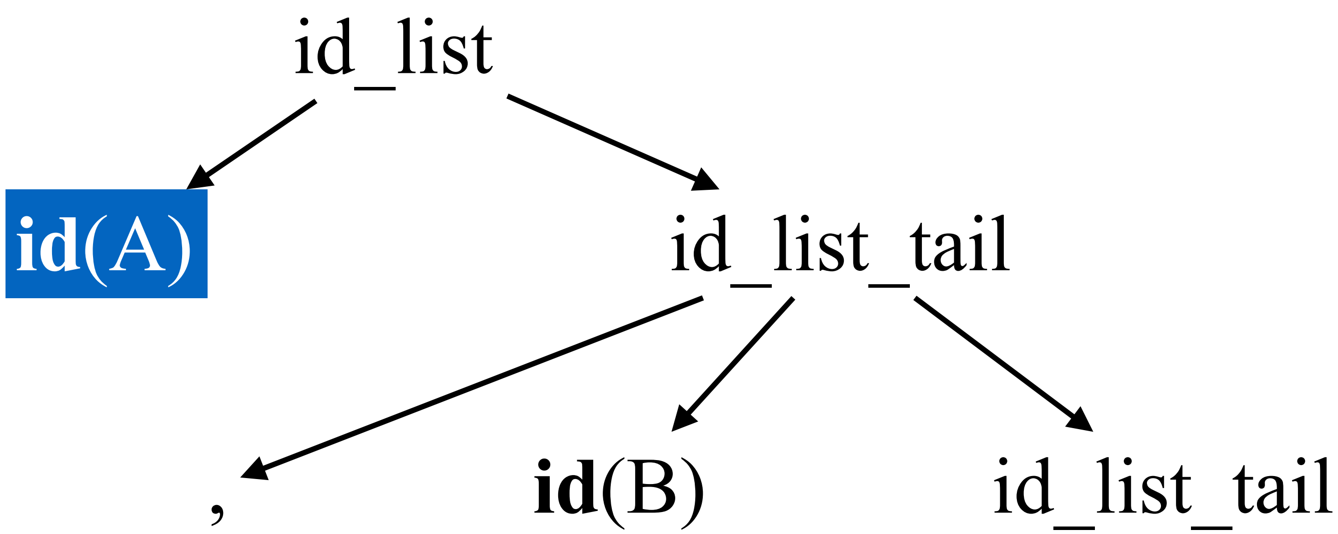


Applied Production:

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;

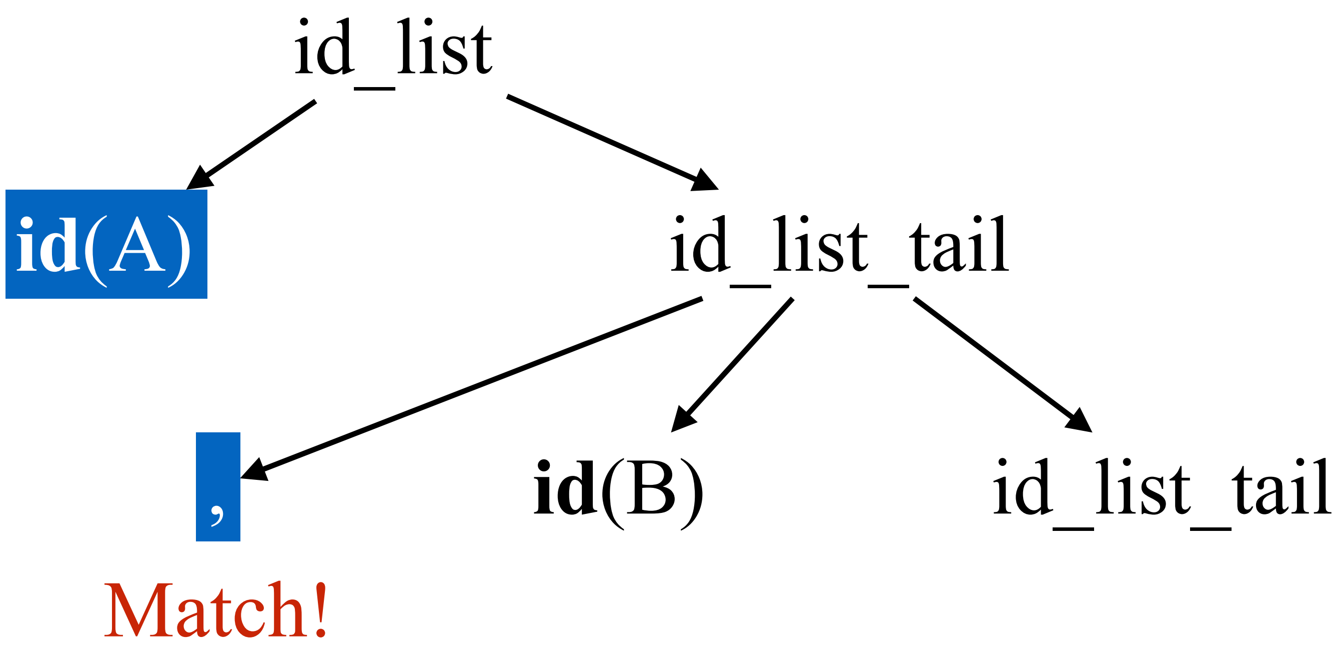


Applied Production:
id_list_tail ::= **, id** id_list_tail

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

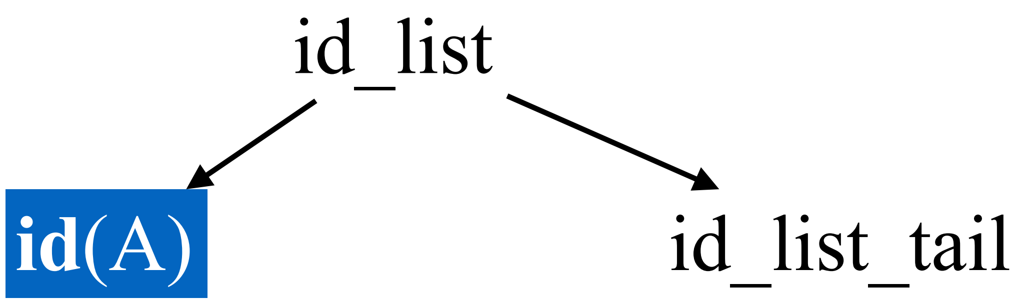
Remaining Input:
, B , C ;



Applied Production:
id_list_tail ::= **, id** id_list_tail

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**



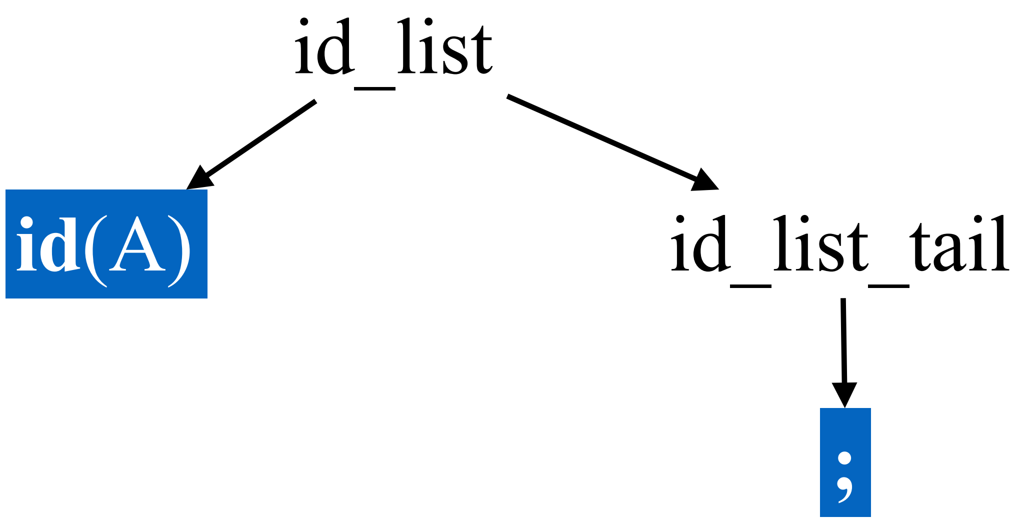
Remaining Input:
$\boxed{,}$ B , C ;

Applied Production:

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;



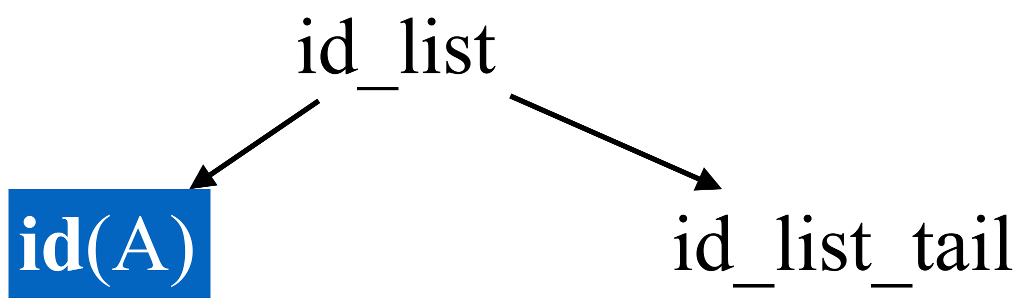
Mismatch!

Applied Production:
~~id_list_tail~~ ::= **;**

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;



FIRST(**, id** id_list_tail) = { , }

FIRST(**;**) = { ; }

Given id_list_tail as the first **non-terminal** to expand in the tree:

If the first token of remaining input is , we choose the rue

id_list_tail ::= **, id** id_list_tail

If the first token of remaining input is ; we choose the rule

id_list_tail ::= **;**

Predictive Parsing

Key Property:

Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$

Revisiting the id_list Example

id_list ::= **id** id_list_tail
id_list_tail ::= **, id** id_list_tail
id_list_tail ::= **;**

Remaining Input:
, B , C ;

id_list

id(A) id_list_tail

FIRST(**, id** id_list_tail) = { , }

FIRST(**;**) = { ; }

FIRST (**, id** id_list_tail ∩ *FIRST* (;) = ∅

Given id_list_tail as the first **non-terminal** to expand in the tree:

If the first token of remaining input is , we choose the rue
id_list_tail ::= **, id** id_list_tail
If the first token of remaining input is ; we choose the rule
id_list_tail ::= **;**

Key Property:

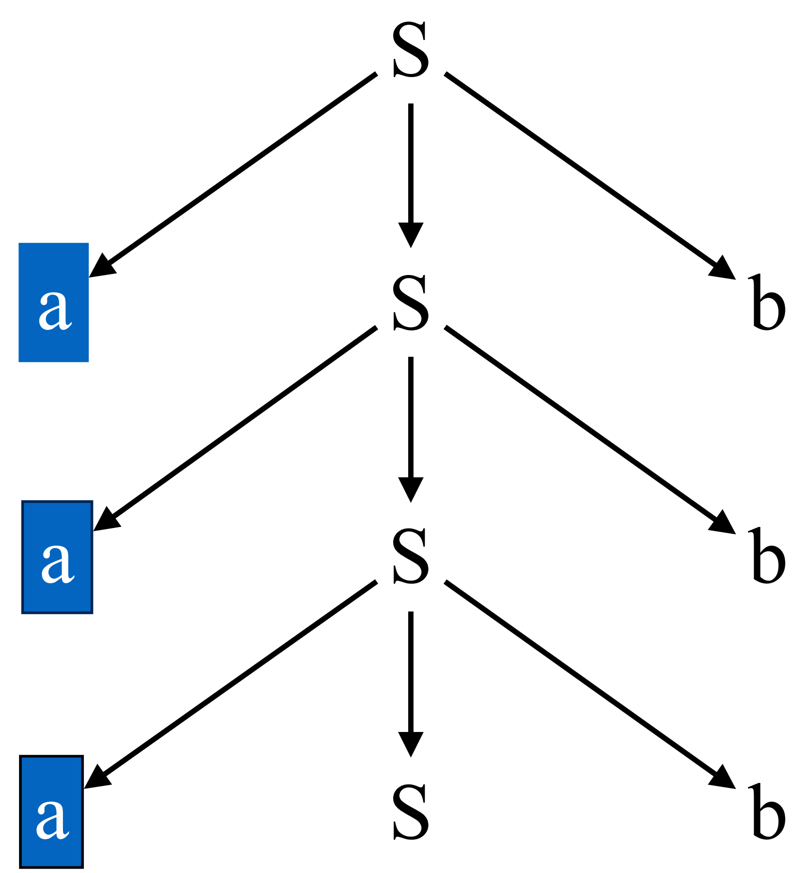
Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$

This rule is intuitive. However, it is **not correct,** because it doesn't handle ε rules. How to handle ε rules?

Revisiting the LL(1) Parsing Example

S ::= a S b | ε

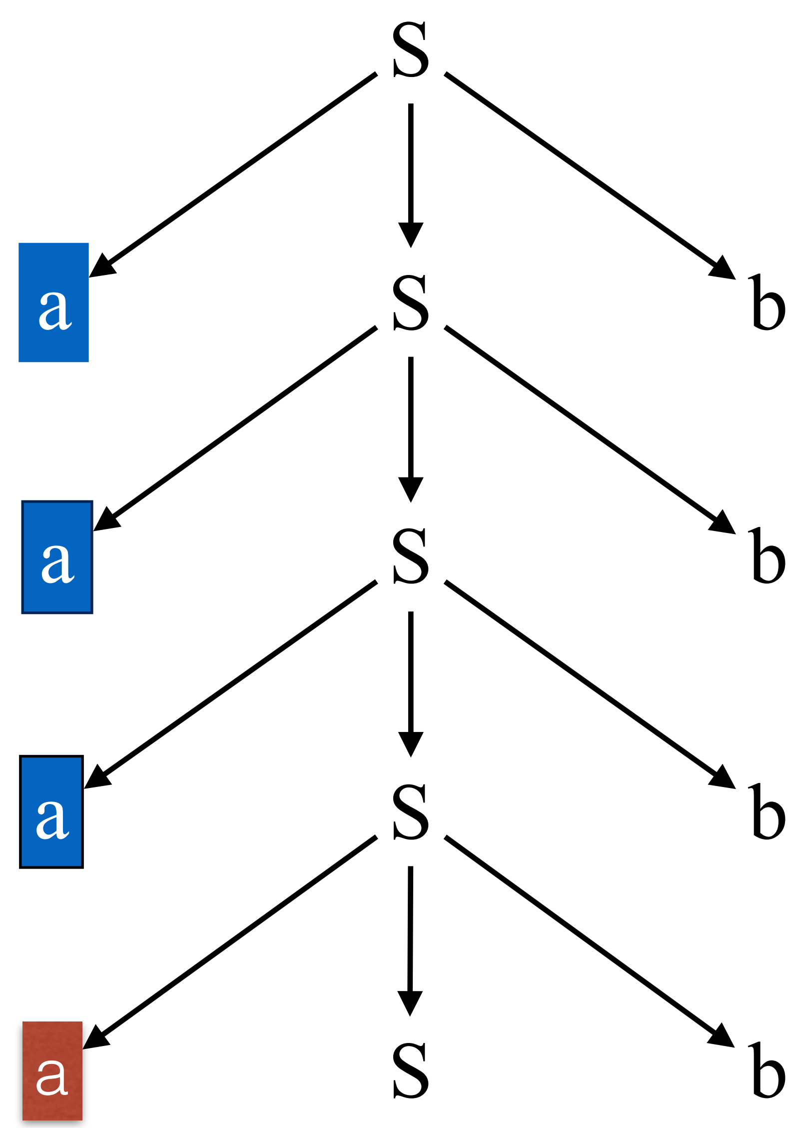


Remaining Input:
b b b

Applied Production:

Revisiting the LL(1) Parsing Example

S ::= a S b | ε



Remaining Input:
b b b

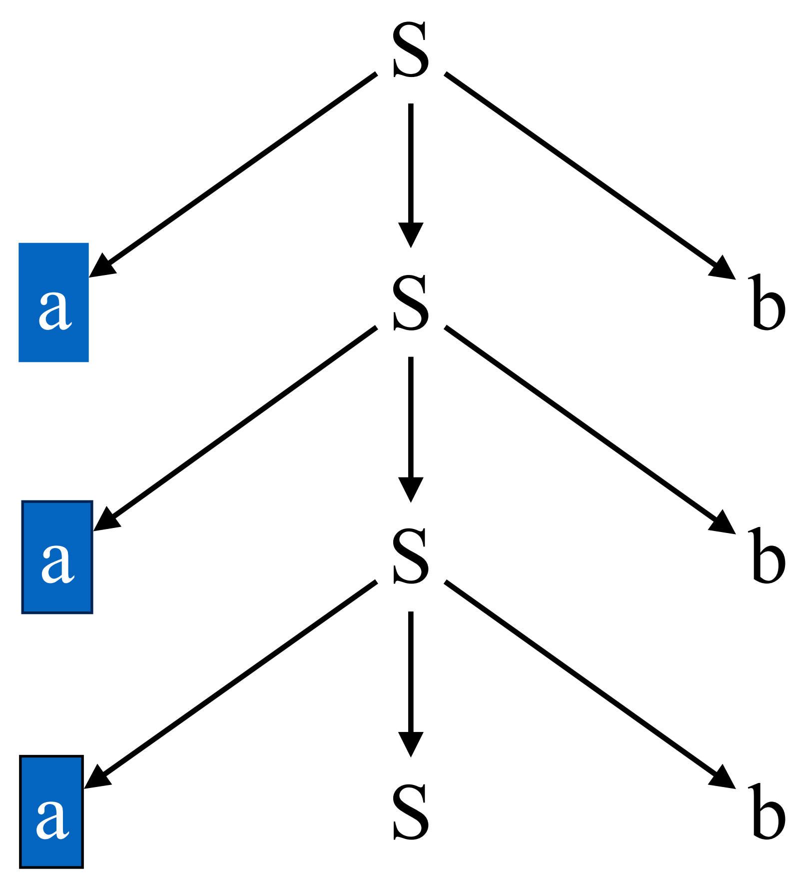
Applied Production:

~~S ::= a S b~~

Mismatch!
It only means S ::= aSb is not the right production rule to use!

Revisiting the LL(1) Parsing Example

S ::= a S b | ε

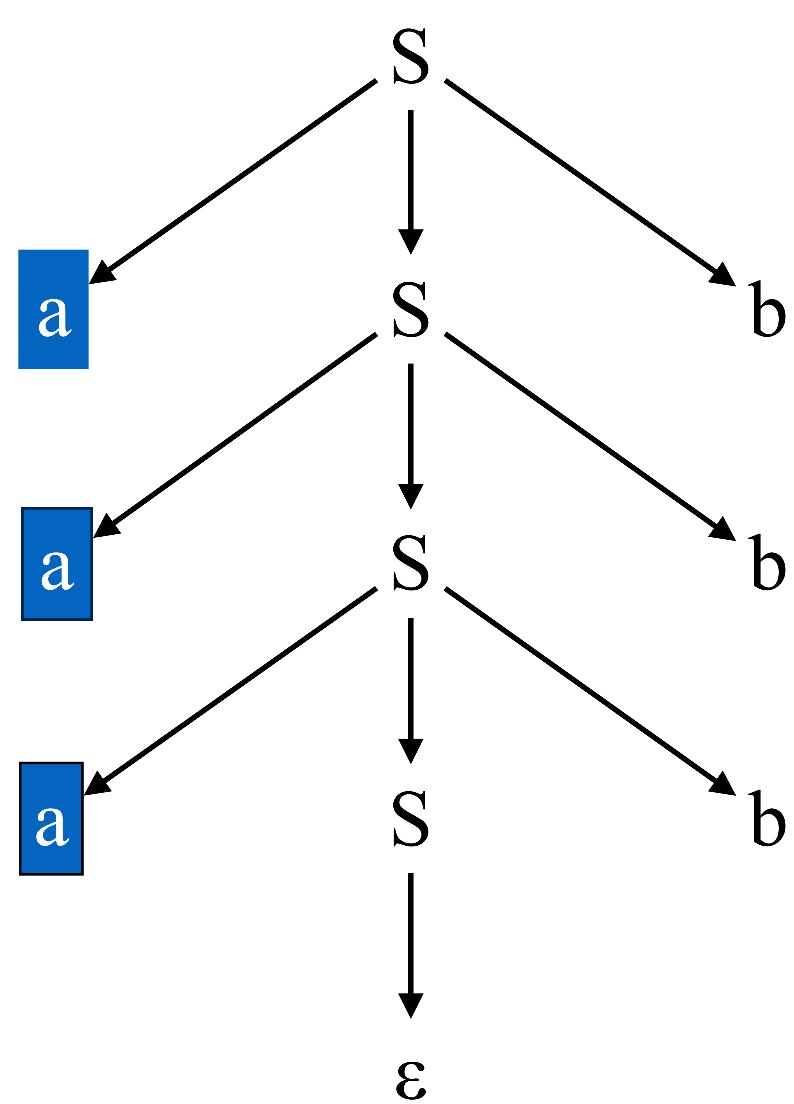


Remaining Input:
b b b

Applied Production:

Revisiting the LL(1) Parsing Example

S ::= a S b | ε



Remaining Input:
b b b

Applied Production:
S ::= ε

S ::= ε turns out to be the right rule later.

However, at this point:
ε does not match "b" either !

Predictive Parsing

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's **FOLLOW** set specifies the tokens that can legally appear after it. A terminal symbol has no **FOLLOW** set.

FIRST and FOLLOW sets can be constructed automatically

Predictive Parsing

Key Property:

Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $\mathit{FIRST}\,(\alpha) \cap \mathit{FIRST}\,(\beta) = \varnothing$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

Key Property:

Whenever two productions A ::= α and A ::= β both appear in the grammar, we would like

- $\mathit{FIRST}(\alpha) \cap \mathit{FIRST}(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \varepsilon$, then $\mathit{FIRST}(\beta) \cap \mathit{FOLLOW}(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \varepsilon$. Note: due to first condition, at most one of α and β can derive ε.

This would allow the parser to make a correct choice with a lookahead of only one symbol!

LL(1) Grammar

Define $FIRST^{+}(A ::= \delta)$ for rule $A ::= \delta$

- $FIRST(\delta) - \{ \varepsilon \} \cup Follow(A)$, if $\varepsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A Grammar is LL(1) iff
($A ::= \alpha$ and $A ::= \beta$) implies

$$FIRST^{+}(A ::= \alpha) \cap FIRST^{+}(A ::= \beta) = \emptyset$$

Back to Our Example

Start ::= S **eof**
S ::= **a** S **b** | ε

FIRST(aSb) = {a}

FIRST(ε) = {ε}

FOLLOW(S) = {eof, b}

Is the grammar LL(1)?

$FIRST^{+}$(S ::= aSb) = {a}

$FIRST^{+}$(S ::= ε) = (*FIRST*(ε) - {ε}) ∪ *FOLLOW*(S) ={eof, b}

Define $FIRST^{+}$(δ) for rule A ::= δ

- *FIRST* (δ) - { ε } U Follow (A), if ε ∈ *FIRST*(δ)
- *FIRST* (δ) otherwise

Table Driven LL(1) Parsing

Example:

S ::= **a** S **b** | ε

LL(1) parse table

	a	b	eof	other
S	aSb	ε	ε	error

How to parse input **a a a b b b** ?

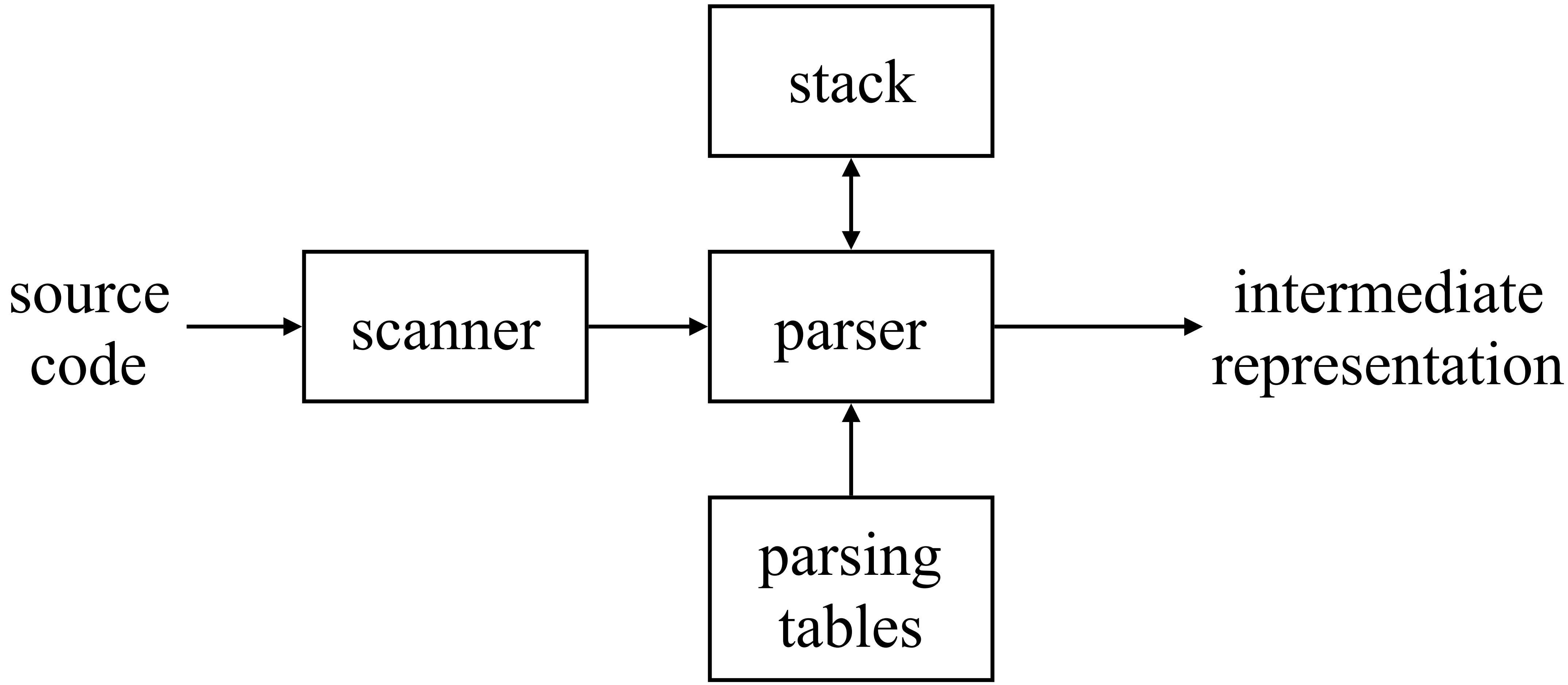
Table Driven LL(1) Parsing

Input: *a string w and a parsing table M for G*

```
push eof
push Start Symbol
token ← next_token()
X ← top-of-stack
repeat
    if X is a terminal then
      if X == token then
          pop X
          token ← next_token()
      else error()
    else /* X is a non-terminal */
          if M[X, token] == X → Y1Y2 . . . Yk then
              pop X
              push Yk, Yk−1, . . . , Y1
          else error()
      X ← top-of-stack
until X = eof
if token != eof then error()
```

Predictive Parsing

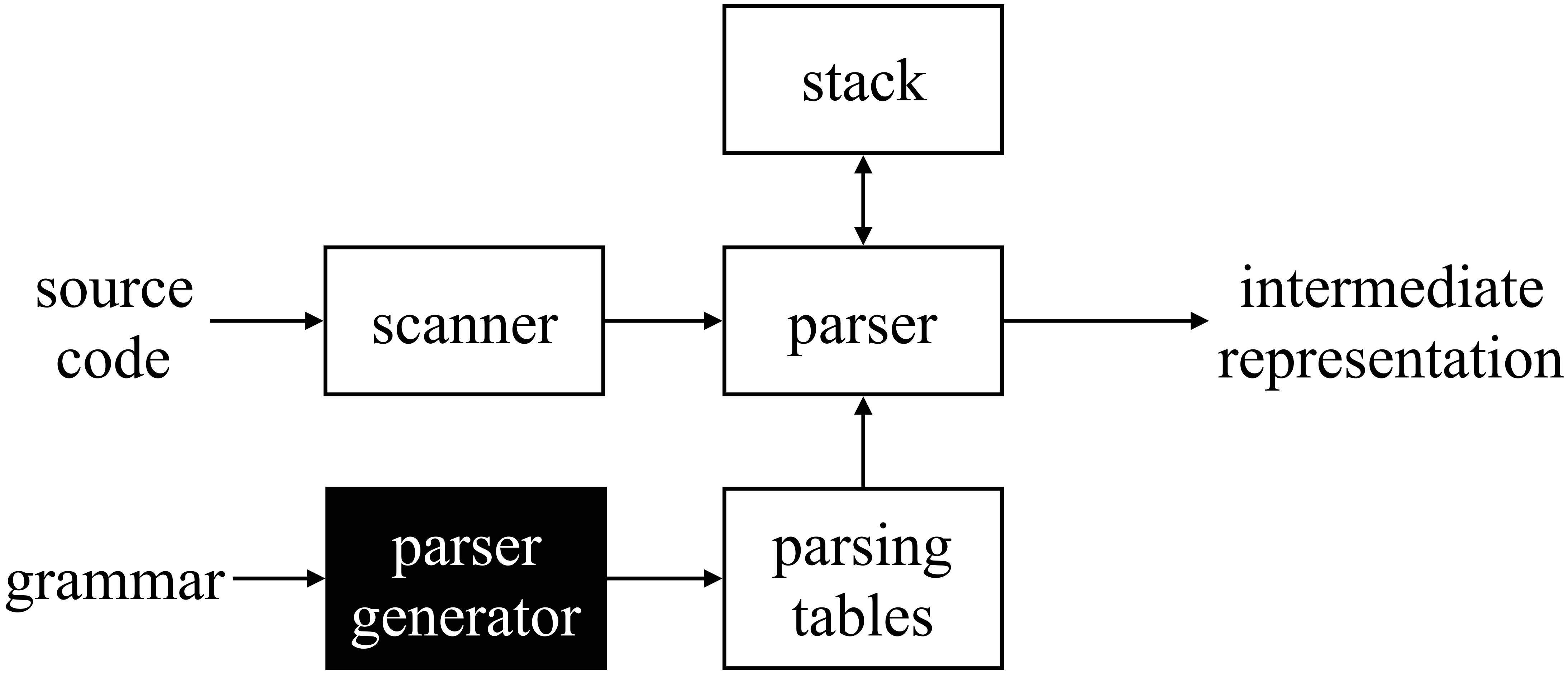
Now, a predictive parser looks like:



Rather than writing code, we build tables.
Building tables can be automated!

Predictive Parsing

Now, a predictive parser looks like:



Rather than writing code, we build tables.
Building tables can be automated!

Recursive Descent Parsing

Now, we can produce a recursive descent parser for our LL(1) grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each **non-terminal** has an associated parsing procedure that can recognize any sequence of tokens generated by that **non-terminal**
- There is a main routine to initialize all globals (e.g: *tokens*) and call the start symbol. On return, check whether token==eof, and whether errors occurred
- Within a parsing procedure, both **non-terminals** and terminals can be matched:
  - non-terminal A: call procedure for A
  - token t: compare t with current input token; if matched, consume input, otherwise, ERROR
- Parsing procedure may contain code that performs some useful "computations" (*syntax directed translation)*

Recursive Descent Parsing (pseudo code)

	a	b	eof	other
S	aSb	ε	ε	error

```
main:  {
        token := next_token( );
        if (S( ) and token == eof) print "accept" else print "error";
}
```

Recursive Descent Parsing (pseudo code)

	a	b	eof	other
S	aSb	ε	ε	error

```
bool S:  {
        switch token {
              case a:  token := next_token( );
                        call S( );
                        if (token == b)  {
                            token := next_token( );
                             return true;
                        }
                        else
                             return false;
                        break;
              case b:
              case eof:  return true;
                          break;
              default: return false;
        }
}
```

Next Lecture

Next Time:

- Review of LL(1) parsing and syntax directed translation
- Read Scott, Chapter 2.3.1 - 2.3.2