CS 314 Principles of Programming Languages

Lecture 9: LL(1) Parsing & Syntax Directed Translation

Zheng (Eddy) Zhang

Rutgers University

February 14, 2018

Class Information

- Homework 3 due this Sunday, 2/18 11:55 pm EST.
- Project 1 will be posted after homework 3 is due.

FIRST and FOLLOW Sets

FOLLOW(A):

For A $\in$ **NT** , define **FOLLOW**(A) as the set of tokens that can occur immediately after A in a valid sentential form.

FOLLOW set is defined over the set of non-terminal symbols (**NT**)

Back to Our Example

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

FOLLOW(S) = { eof , b }

Follow Set Construction

Given a rule *p* in the grammar:

$$A \rightarrow B_1 B_2 \ldots B_i B_{i+1} \ldots B_k$$

If B_i is a non-terminal, FOLLOW(B_i) includes

- FIRST($B_{i+1} \ldots B_k$) - {ε} U FOLLOW(A), if $\varepsilon \in$ FIRST($B_{i+1} \ldots B_k$)
- FIRST($B_{i+1} \ldots B_k$) otherwise

Relationship between FOLLOW sets and FIRST sets of different symbols

Follow Set Construction

To Build FOLLOW(X) for non-terminal X:

- Place EOF in FOLLOW(<start>)
- For each X as a non-terminal, initialize FOLLOW(X) to ∅

Iterate until no more terminals can be added to any FOLLOW(X):

For each rule *p* in the grammar

If p is of the form A ::= αBβ, then

if ε ∈ *FIRST*(β)

Place {FIRST(β) - ε, FOLLOW(A)} in FOLLOW(B)

else

Place {FIRST(β)} in FOLLOW(B)

If p is of the form A ::= αB, then

Place FOLLOW(A) in FOLLOW(B)

End iterate

Pseudocode for Computing *FOLLOW* Sets

```
for each A ∈ NT
    FOLLOW(A) ← ∅
FOLLOW(S) ← { EOF }

while (FOLLOW sets are still changing) do
    for each p ∈ P, of the form A → B1B2…Bk do
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then                          // domain checking
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ε ∈ FIRST(Bi)                     // add right context
                    TRAILER ← TRAILER ∪ (FIRST(Bi) - { ε })
                else TRAILER ← FIRST(Bi)  // no ε => truncate the right context
            else TRAILER ← { Bi }                    // Bi ∈ T => only 1 symbol
```

Don't add ε

To build *FOLLOW* sets, we need *FIRST* sets

Computing *FOLLOW* Sets

For a production $A \rightarrow B_1 B_2 \ldots B_k$:

- The pseudocode works its way backward through the production: $B_k, B_{k-1}, \ldots B_1$
- It builds the ***FOLLOW*** sets for the rhs symbols, $B_1, B_2, \ldots B_k$, not A
- If ε does not belong to ***FIRST***(B_{i+1}), ***FOLLOW***(B_i) is just ***FIRST***(B_{i+1})

To handle ε, the algorithm in the previous slide keeps track of the first token in the trailing right context as it works its way back through rhs: $B_k, B_{k-1}, \ldots B_1$

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*
Goal	**EOF**
List	∅
Pair	∅

Initial Values:

- Goal, List and Pair are set to ∅
- Goal is then set to { **EOF** }

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st
Goal	**EOF**	**EOF**
List	∅	
Pair	∅	

Iteration 1:

If we visit the rules in order 1, 2, 3, 4

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

```
1  Goal ::= List
2  List  ::= Pair List
3            |  ε
4  Pair ::= LP List RP
```

Iteration 1:

Symbol	*Initial*	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**
Pair	∅	

If we visit the rules in order 1, 2, 3, 4

Goal ::= List

Add FOLLOW(Goal) to FOLLOW(List)

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**
Pair	∅	**EOF**, LP

Iteration 1:

If we visit the rules in order 1, 2, 3, 4

List ::= Pair List

- Add FIRST(List) to FOLLOW(Pair)
- Add FOLLOW(List) to FOLLOW(Pair)

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**, RP
Pair	∅	**EOF**, LP

Iteration 1:

If we visit the rules in order 1, 2, 3, 4

Pair ::= LP List RP

- Add FIRST(RP) to FOLLOW(List)

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st
Goal	**EOF**	**EOF**
List	∅	**EOF**, RP
Pair	∅	**EOF**, LP

Iteration 1:

If we visit the rules in order 1, 2, 3, 4

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st	2nd
Goal	**EOF**	**EOF**	**EOF**
List	∅	**EOF**, RP	**EOF**, RP
Pair	∅	**EOF**, LP	**EOF**, LP

Iteration 2:

If we visit the rules in order 1, 2, 3, 4

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st	2nd
Goal	**EOF**	**EOF**	**EOF**
List	∅	**EOF**, RP	**EOF**, RP
Pair	∅	**EOF**, LP	**EOF**, LP

Iteration 2:

If we visit the rules in order 1, 2, 3, 4

Goal ::= List

~~Add FOLLOW(Goal) to FOLLOW(List)~~

Symbol	*FIRST* Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st	2nd
Goal	**EOF**	**EOF**	**EOF**
List	∅	**EOF**, RP	**EOF**, RP
Pair	∅	**EOF**, LP	**EOF**, LP, RP

Iteration 2:

If we visit the rules in order 1, 2, 3, 4

List ::= Pair List

- ~~Add FIRST(List) to FOLLOW(Pair)~~
- Add FOLLOW(List) to FOLLOW(Pair)

Symbol	*FIRST* Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st	2nd
Goal	**EOF**	**EOF**	**EOF**
List	∅	**EOF**, RP	**EOF**, RP
Pair	∅	**EOF**, LP	**EOF**, RP, LP

Iteration 2:

If we visit the rules in order 1, 2, 3, 4

Pair ::= LP List RP

- ~~Add FIRST(RP) to FOLLOW(List)~~

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

An Example

Consider the simplest parentheses grammar

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Symbol	*Initial*	1st	2nd
Goal	**EOF**	**EOF**	**EOF**
List	∅	**EOF**, RP	**EOF**, RP
Pair	∅	**EOF**, LP	**EOF**, RP, LP

Iteration 2:

- Production 1 adds nothing new
- Production 2 adds RP to ***FOLLOW***(Pair)

from ***FOLLOW***(List), ε ∈ ***FIRST***(List)

- Production 3 does nothing
- Production 4 adds nothing new

Symbol	***FIRST*** Set
Goal	LP, ε
List	LP, ε
Pair	LP
LP	LP
RP	RP
EOF	EOF

Iteration 3 produces the same result ⇒ reached a fixed point

Building Top-down Parsers

Building the *FIRST+* set

- Need a ***FIRST+ set*** for every rule

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

- *FIRST* (δ) - { ε } U Follow (A), if $\varepsilon \in$ *FIRST*(δ)
- *FIRST* (δ) otherwise

Symbol	*FIRST*	*FOLLOW*
Goal	LP, ε	**EOF**
List	LP, ε	**EOF**, RP
Pair	LP	**EOF**, RP, LP
LP	LP	-
RP	RP	-
EOF	EOF	-

1	Goal ::= List
2	List ::= Pair List
3	List ::= ε
4	Pair ::= LP List RP

Rule	*FIRST+*
1	**EOF, LP**
2	**LP**
3	**EOF**, RP
4	LP

Building Top-down Parsers

Building the *FIRST+* set

- Need a ***FIRST+ set*** for every rule

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

- *FIRST* (δ) - { ε } U Follow (A), if $\varepsilon \in$ *FIRST*(δ)
- *FIRST* (δ) otherwise

Symbol	*FIRST*	*FOLLOW*
Goal	LP, ε	**EOF**
List	LP, ε	**EOF**, RP
Pair	LP	**EOF**, RP, LP
LP	LP	-
RP	RP	-
EOF	EOF	-

1	Goal ::= List
2	List ::= Pair List
3	List ::= ε
4	Pair ::= LP List RP

Rule	*FIRST+*	
1	**EOF, LP**	
2	**LP**	← FIRST(Pair List)
3	**EOF, RP**	← FOLLOW(List)
4	LP	

Building Top-down Parsers

Building the complete parse table

- Need a row for every **NT** and a column for every **T**
- Need an interpreter for the table (skeleton parser)

1	Goal ::= List
2	List ::= Pair List
3	List ::= ε
4	Pair ::= LP List RP

Rule	*FIRST+*
1	**EOF, LP**
2	**LP**
3	**EOF**, RP
4	LP

	LP	*RP*	*EOF*
Goal	**1**		**1**
List			
Pair			

Building Top-down Parsers

Building the complete parse table

- Need a row for every **NT** and a column for every **T**
- Need an interpreter for the table (skeleton parser)

1	Goal ::= List
2	List ::= Pair List
3	List ::= ε
4	Pair ::= LP List RP

Rule	*FIRST+*
1	**EOF, RP**
2	**LP**
3	**EOF**, RP
4	LP

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair			

Building Top-down Parsers

Building the complete parse table

- Need a row for every **NT** and a column for every **T**
- Need an interpreter for the table (skeleton parser)

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

Rule	*FIRST+*
1	**EOF, RP**
2	**LP**
3	**EOF**, RP
4	LP

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair	4		

Building Top-down Parsers

Building the complete parse table

- Need a row for every **NT** and a column for every **T**
- Need an interpreter for the table (skeleton parser)

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= <u>LP</u> List <u>RP</u>

Rule	*FIRST+*
1	**EOF, RP**
2	**LP**
3	**EOF**, RP
4	LP

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair	4		

Review: 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()
```

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair	4		

M is the parse table

LL(1) Parsing Example

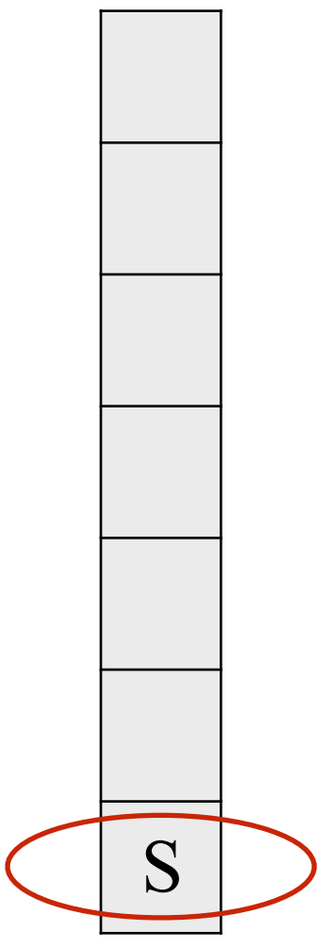
S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error

Remaining Input:

a a a b b b

S



Sentential Form:

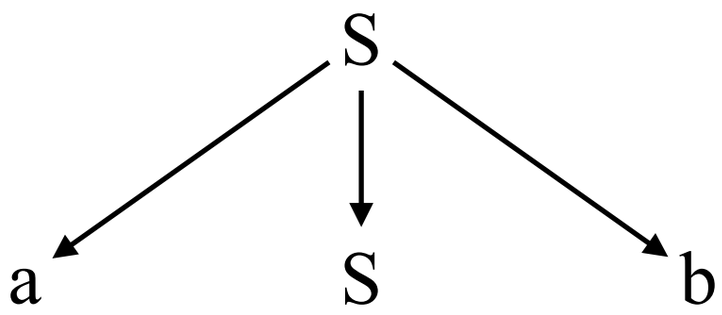
S

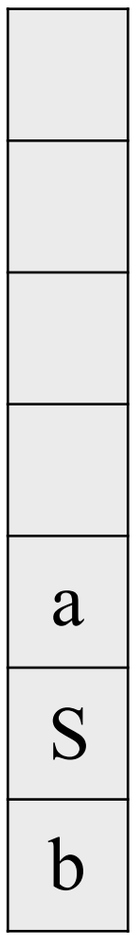
Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error





Remaining Input:
a a a b b b

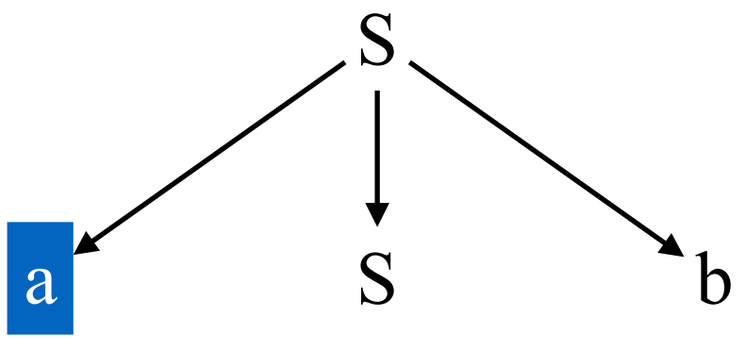
Sentential Form:
a S b

Applied Production:
S ::= a S b

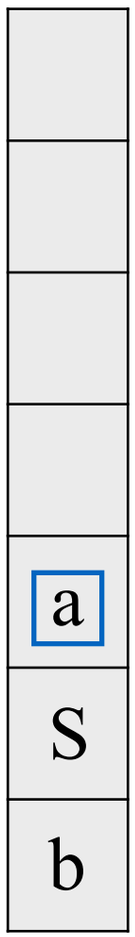
LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Match!



Remaining Input:

a a a b b b

Sentential Form:

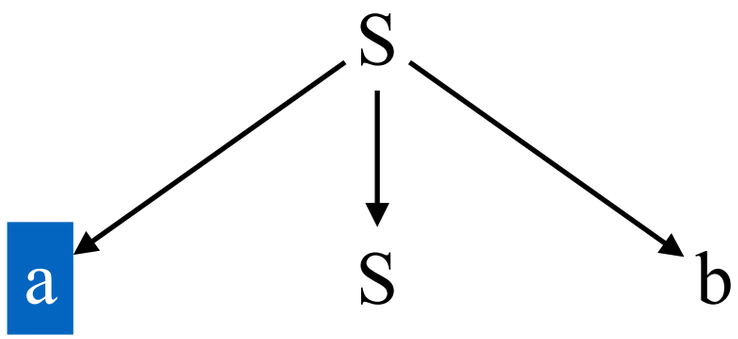
a S b

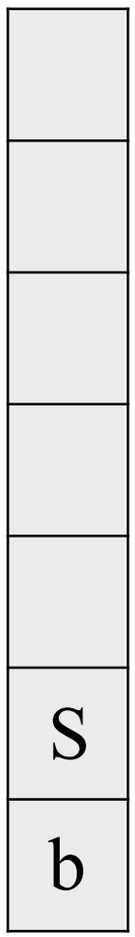
Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error





Remaining Input:
a a b b b

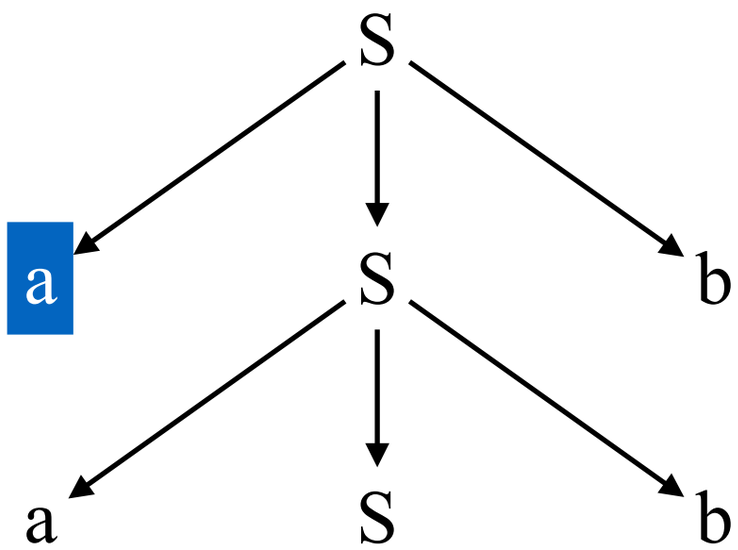
Sentential Form:
a S b

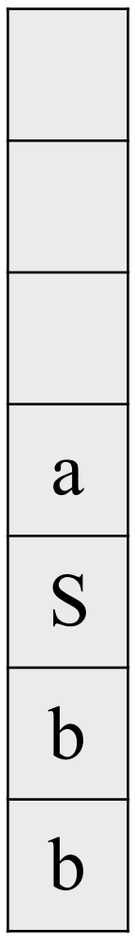
Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error





Remaining Input:
a a b b b

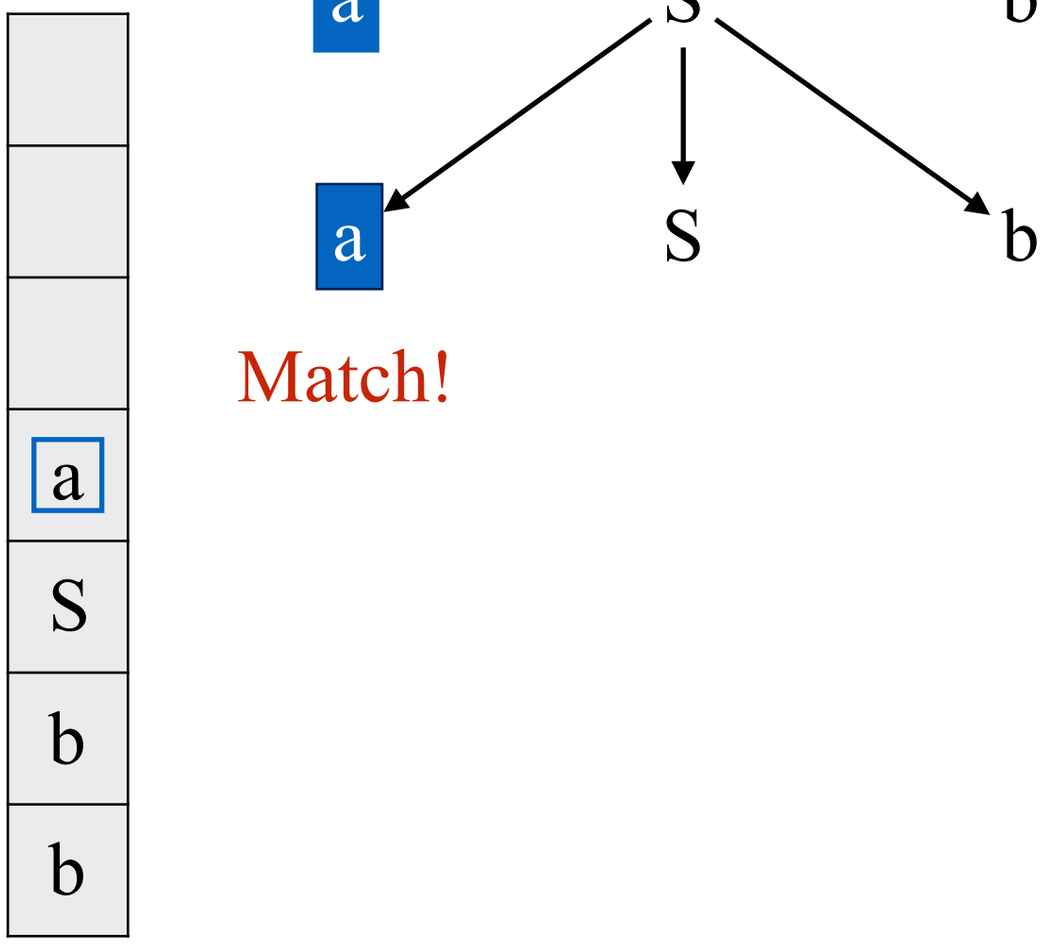
Sentential Form:
a a S b b

Applied Production:
S ::= a S b

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Match!

Remaining Input:

a a b b b

Sentential Form:

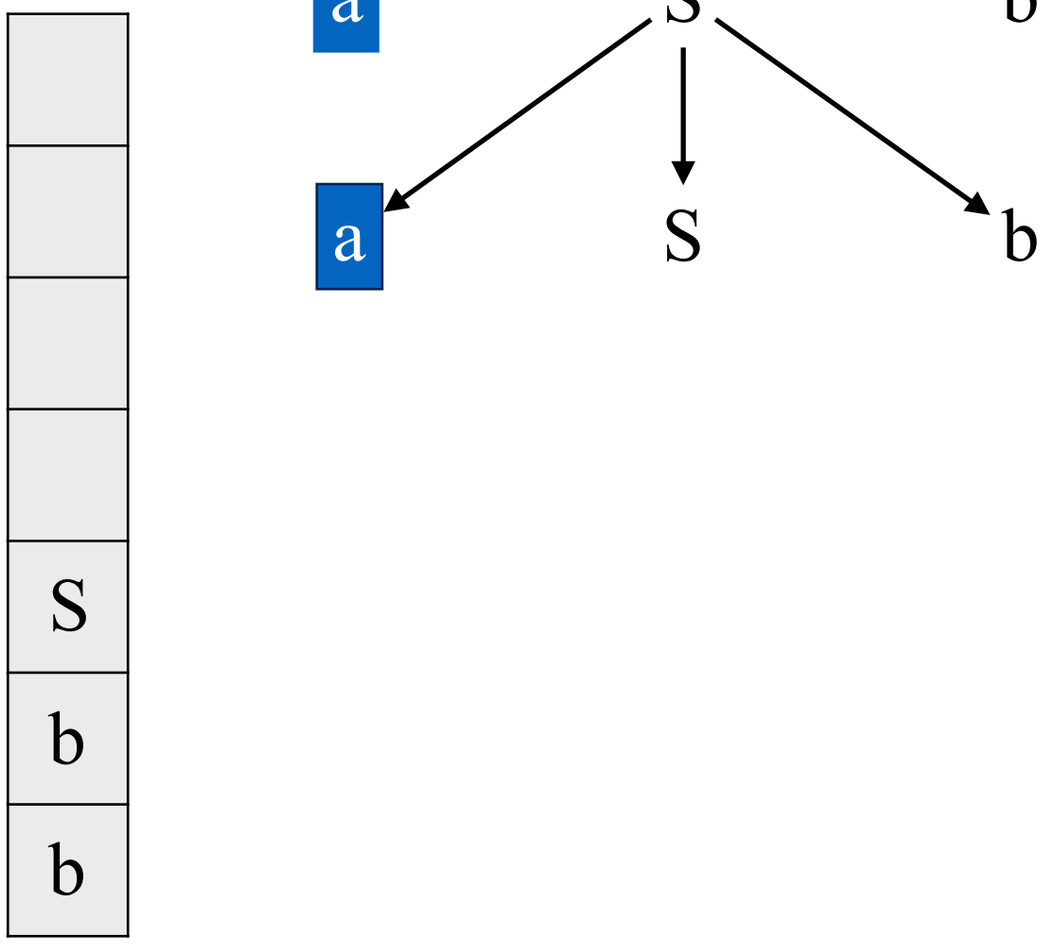
a a S b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
a b b b

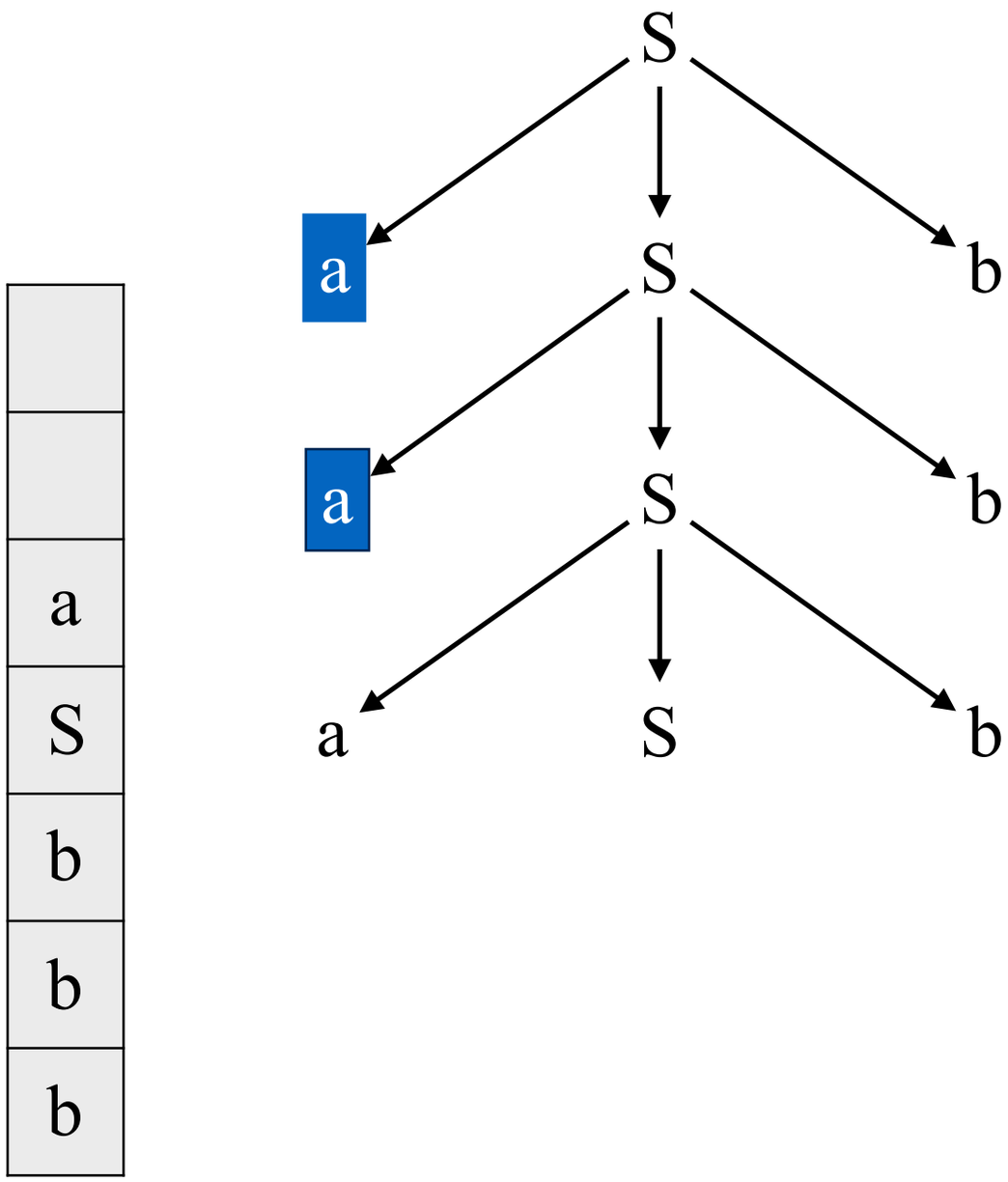
Sentential Form:
a a S b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
a b b b

Sentential Form:
a a a S b b b

Applied Production:
S ::= a S b

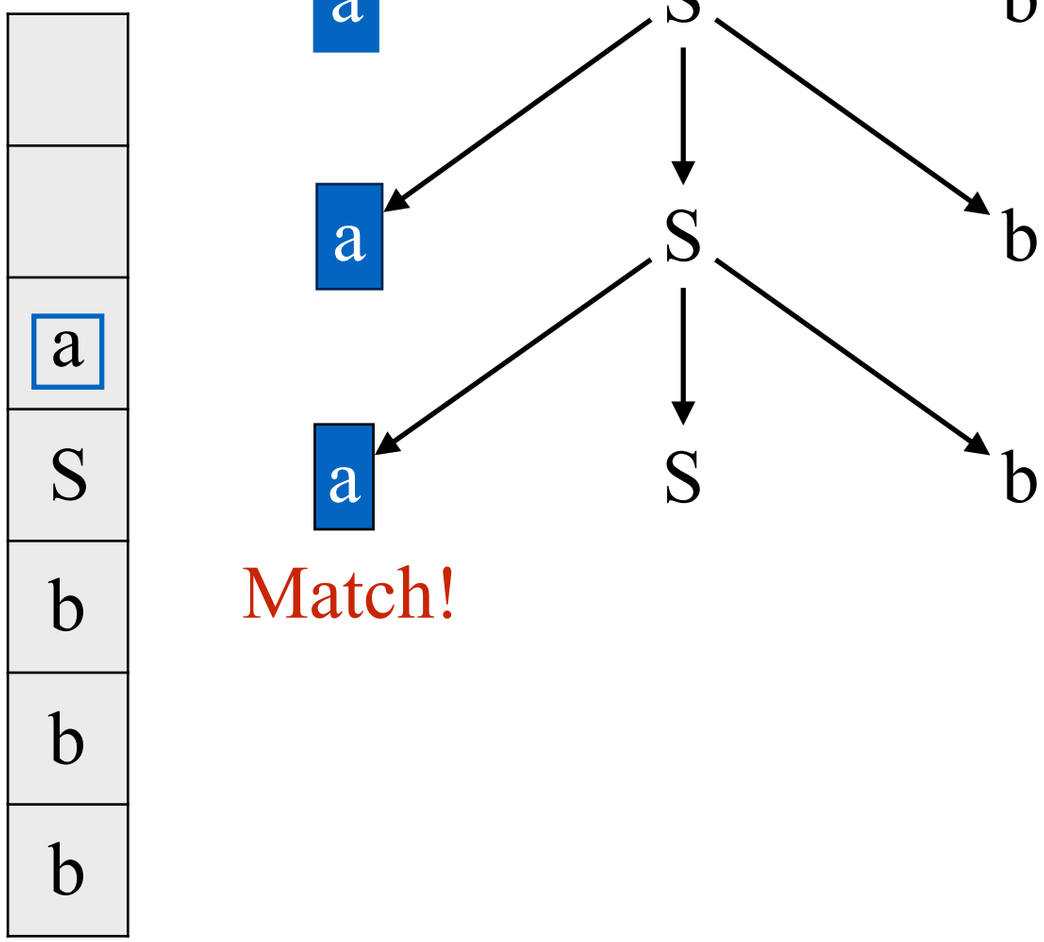
LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error

Remaining Input:

a b b b



Match!

Sentential Form:

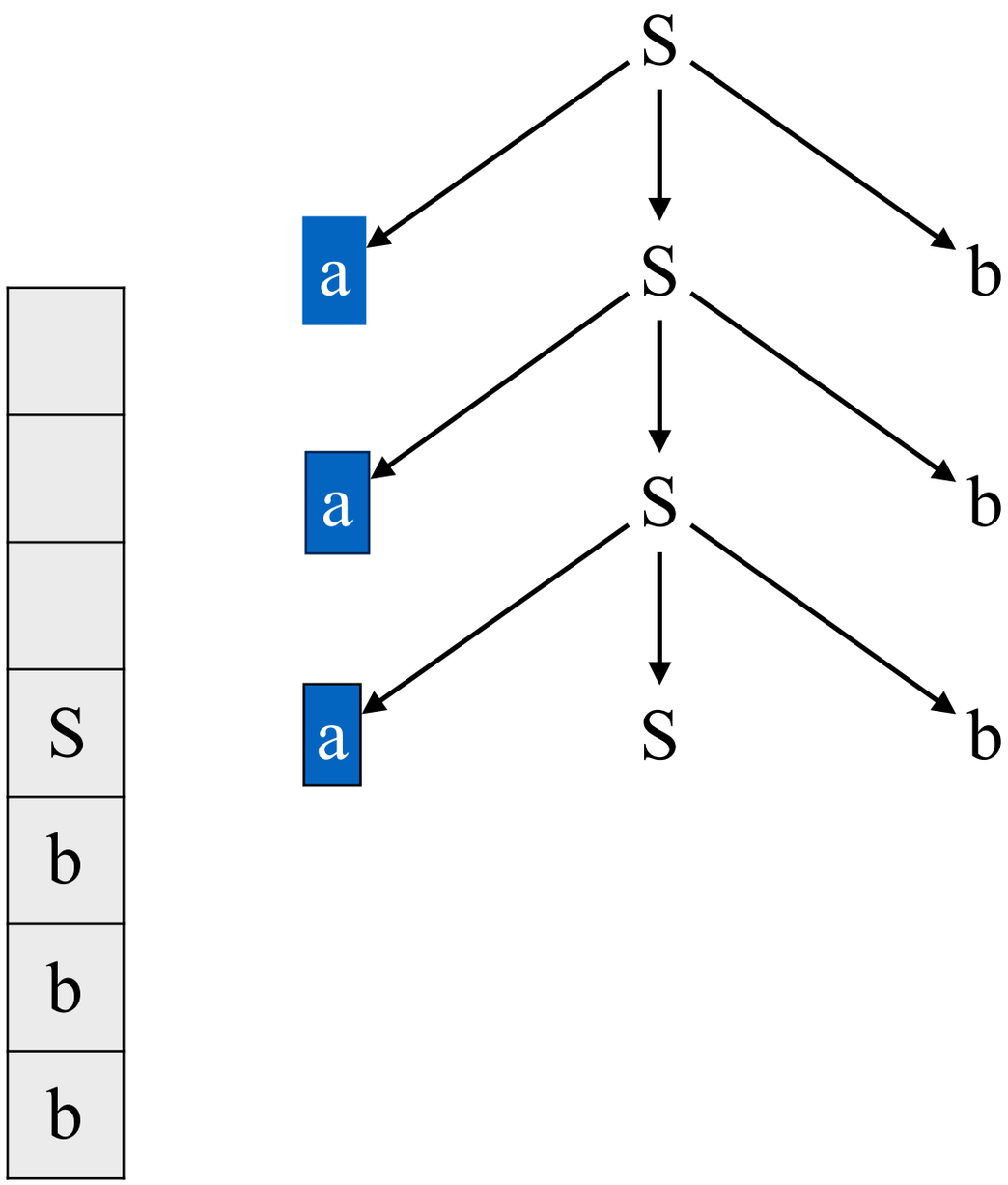
a a a S b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
b b b

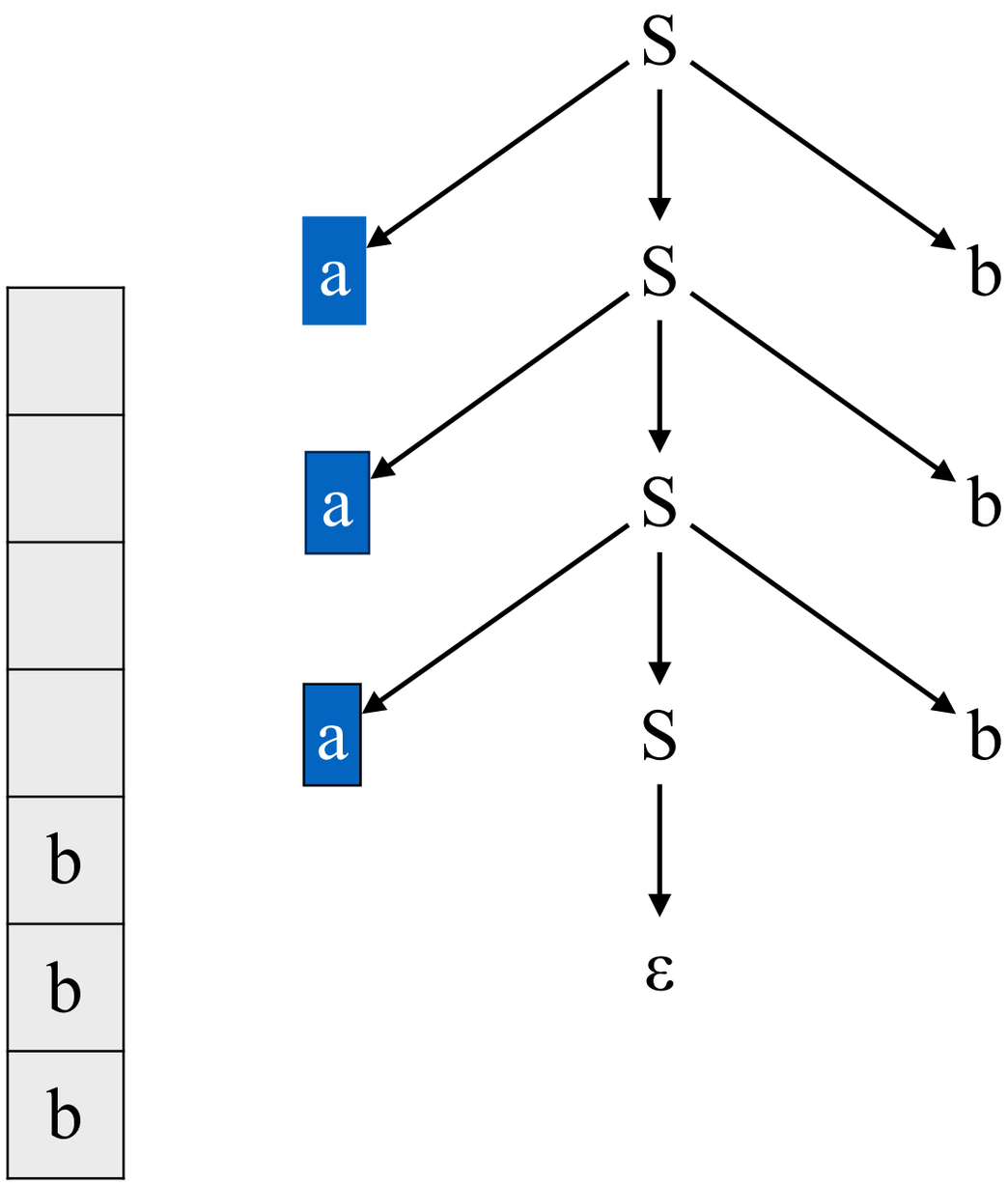
Sentential Form:
a a a S b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
b b b

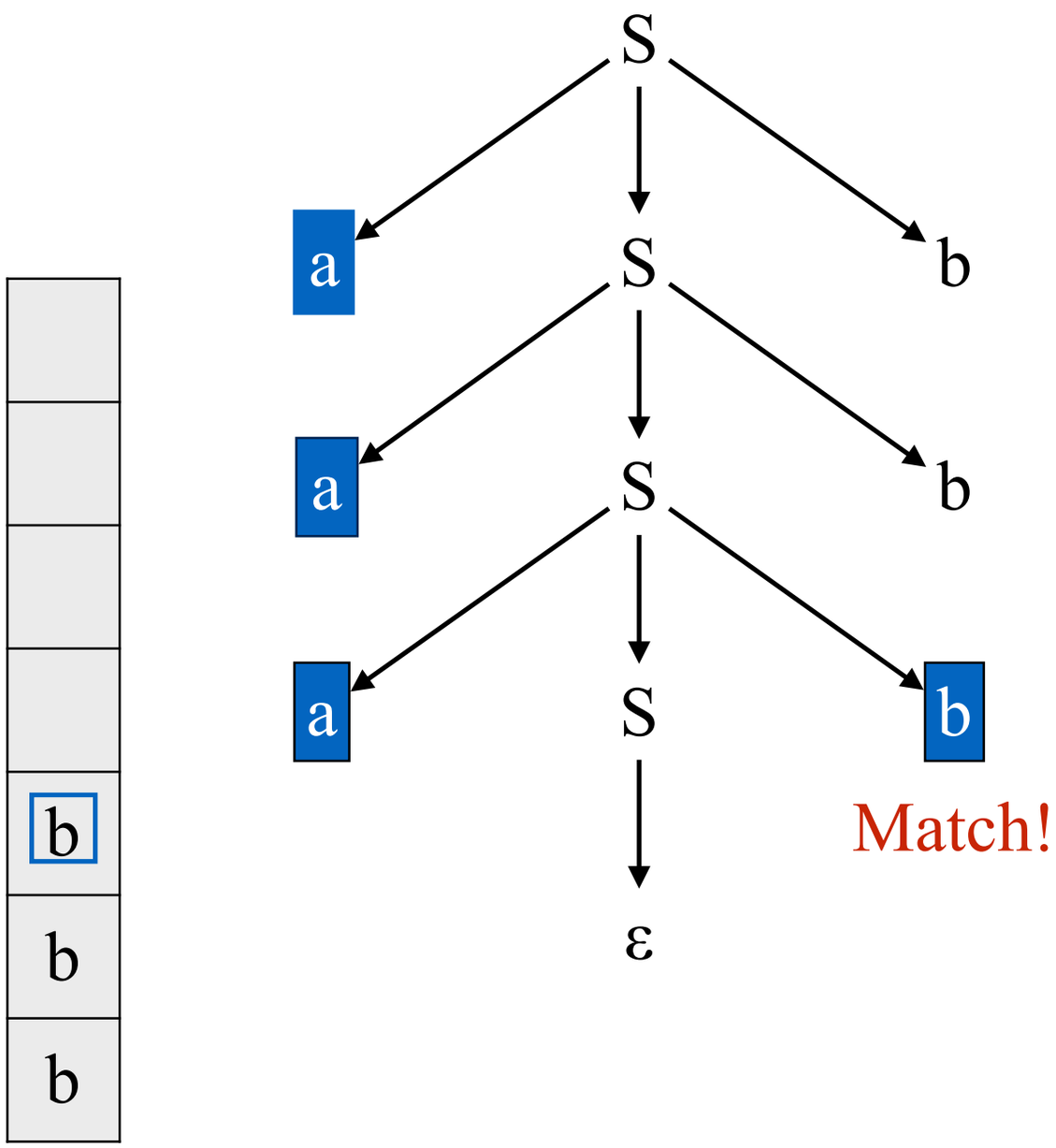
Sentential Form:
a a a b b b

Applied Production:
S ::= ε

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:

b b b

Sentential Form:

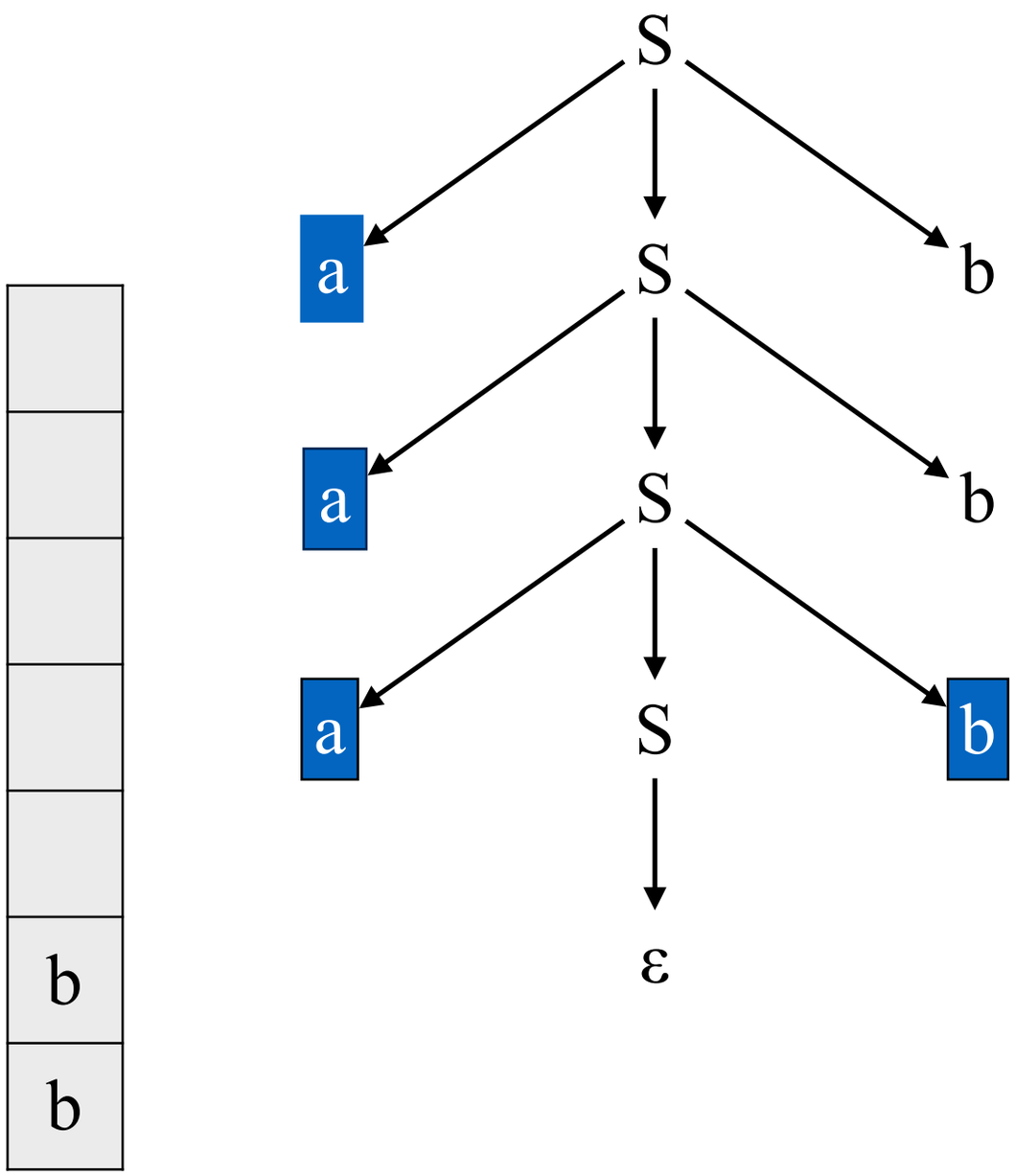
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
b b

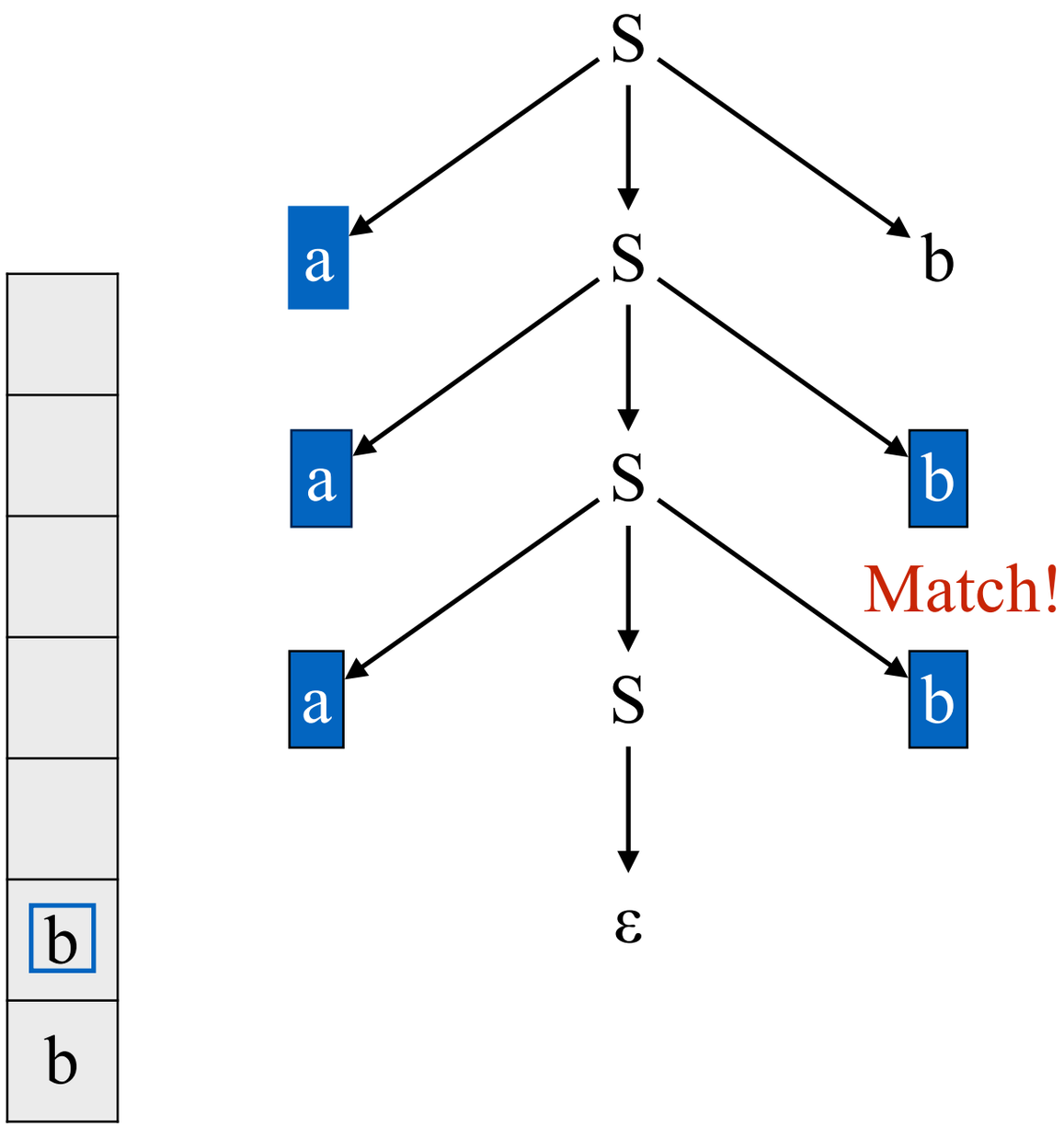
Sentential Form:
a a a b b b

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:

b b

Sentential Form:

a a a b b b

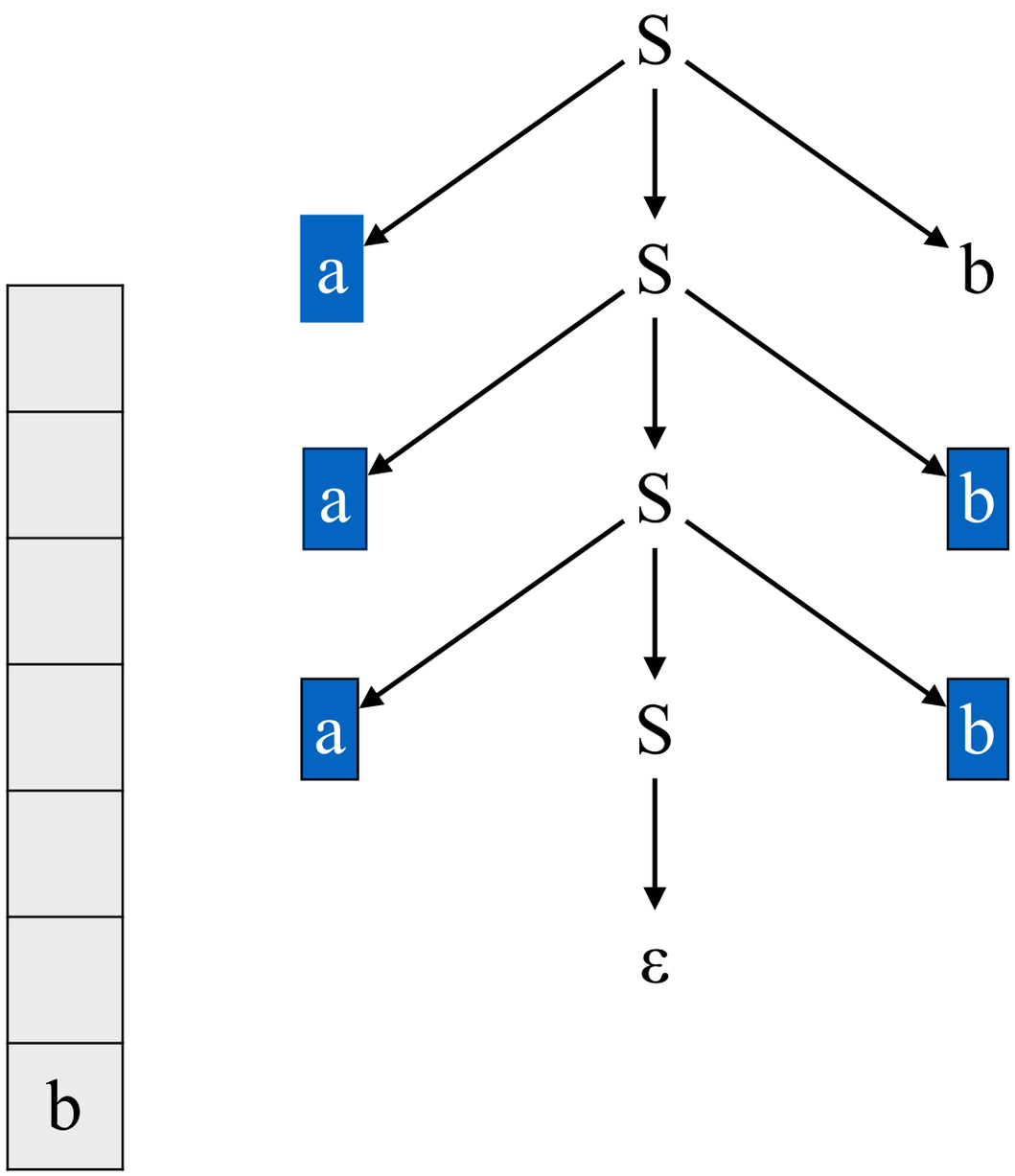
Match!

Applied Production:

LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error



Remaining Input:
b

Sentential Form:
a a a b b b

Applied Production:

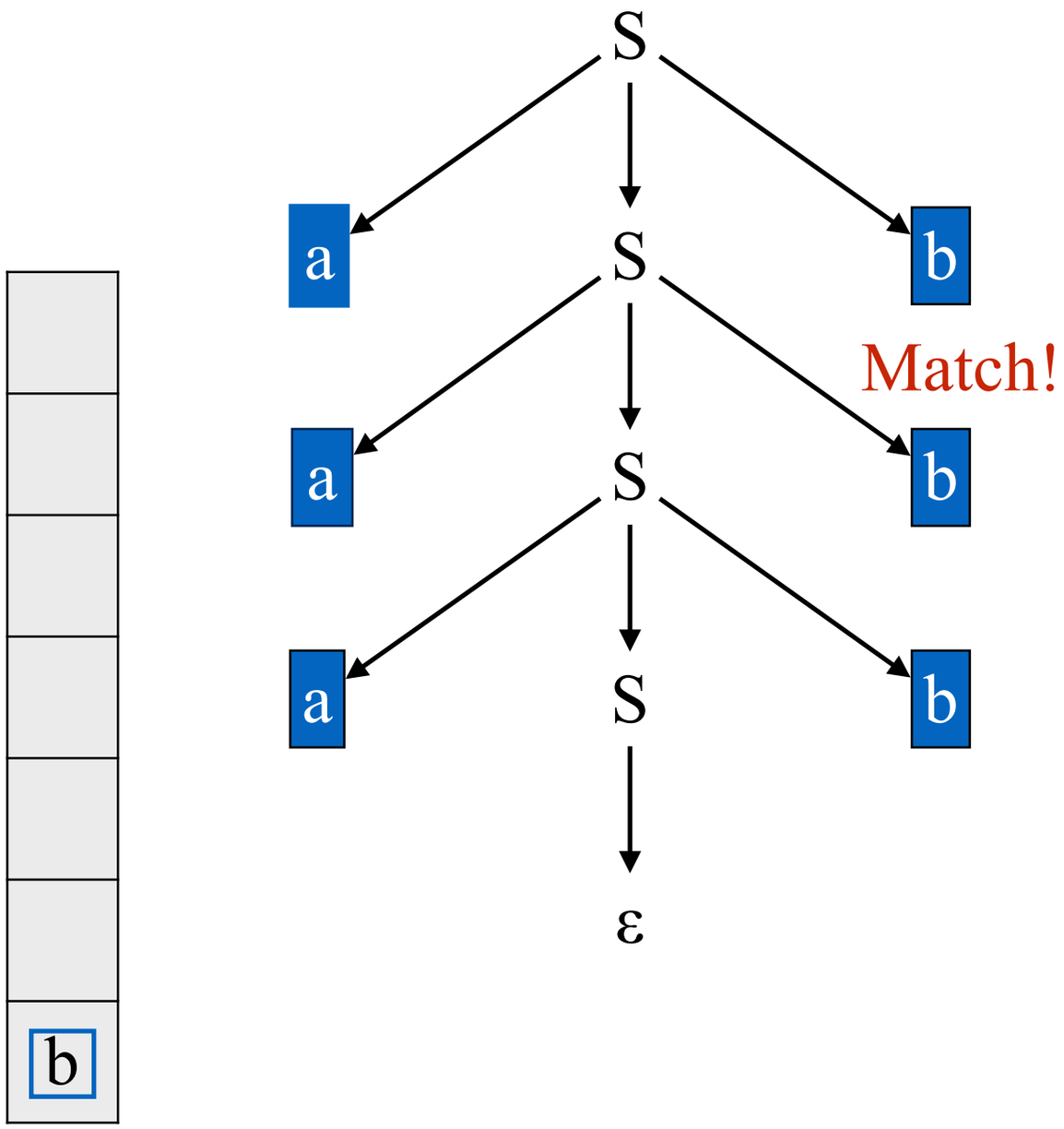
LL(1) Parsing Example

S ::= a S b | ε

	a	b	eof	other
S	S ::=aSb	ε	ε	error

Remaining Input:

b



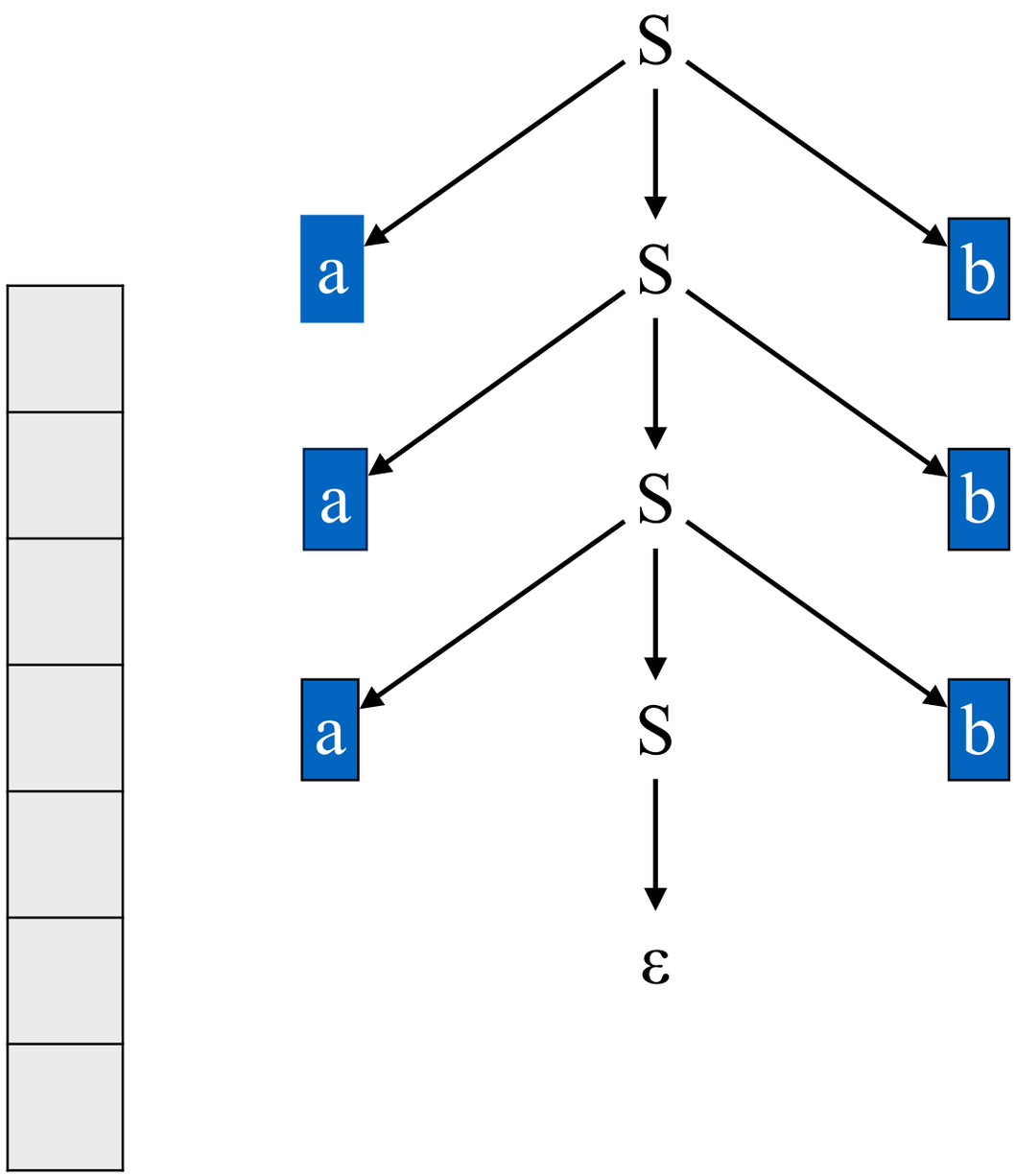
Sentential Form:

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:

Recursive Descent Parsing

Recursive descent parser for LL(1)

- 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:the *token variable* in previous code example) 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

"token" is a global variable that stores the first terminal in the remaining input

```
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;
          }
   }
```

"token" is a global variable that stores the first terminal in the remaining input

Recursive Descent Parsing (pseudo code)

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair	4		

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

```
main:  {
       token := next_token( );
       if ( List( ) and token == EOF) print “accept” else print “error”;
}
```

Recursive Descent Parsing (pseudo code)

	LP	*RP*	*EOF*
Goal	**1**		**1**
List	2	3	3
Pair	4		

1	Goal ::= List
2	List ::= Pair List
3	| ε
4	Pair ::= LP List RP

```
bool List( ):  {
  switch token {
      case LP:
                  call Pair( );
                  call List( );
                 break;
      case RP:
      case EOF:  return true;
                   break;
      default: return false;
  }
}
```

```
bool Pair( ):  {
  switch token {
    case LP:  token := next_token( );
                  call List( );
                  if ( token == RP ) {
                      token := next_token( );
                      return true;
                  }
                  else
                      return false;
                  break;
     default: return false;
    }
}
```

Another Example: Classic Expression Grammar

0 Goal ::= Expr
1 Expo ::= Term Expr’
2 Expr' ::= + Term Expr’
3 | - Term Expr’
4 | ε
5 Term ::= Factor Term’
6 Term’ ::= * Factor Term’
7 | / Factor Term’
8 | ε
9 Factor ::= (Expr)
10 | num
11 | id

FIRST$^+$(A ::= β) is identical to ***FIRST***(β) except for productions 4 and 8
FIRST$^+$(Expr' ::= ε) is {), eof}
FIRST$^+$(Term’ ::= ε) is {+,-,),eof

Symbol	*FIRST*	*FOLLOW*
num	num	∅
id	id	∅
+	+	∅
-	-	∅
*	*	∅
/	/	∅
(	(	∅
)	)	∅
eof	eof	∅
ε	ε	∅
Goal	(, id, num	eof
Expr	(, id, num	), eof
Expr'	+, -, ε	), eof
Term	(, id, num	+, -,), eof
Term’	*, /, ε	+, -,), eof
Factor	(, id, num	+, -, *,), eof

More Examples: Classic Expression Grammar

```
0    Goal ::= Expr
1    Expr ::= Term Expr’
2    Expr' ::= + Term Expr’
3            |  - Term Expr’
4            |  ε
5    Term ::= Factor Term’
6    Term’ ::=  * Factor Term’
7            |   /  Factor Term’
8            |  ε
9    Factor ::= ( Expr )
10           | num
11           | id
```

Prod’n	*FIRST*$^+$
0	(, id, num
1	(, id, num
2	+
3	-
4	), eof
5	(, id, num
6	*
7	/
8	+, -,), eof
9	(
10	num
11	id

Syntax Directed Translation

Examples:

- Interpreter
- Code generator
- Type checker
- Performance estimator

Use hand-written recursive descent LL(1) parser

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
void expr( ):  // return value of the expression
        int val1, val2; // two values
        switch token {
                case +:       token := next_token( );
                              expr( );
                case 0..9:    digit( );
                …
        }

void digit( ):  // return value of constant
        switch token {
                case 1: token := next_token( );
                case 2: token := next_token( );
               …
        }
```

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

```
int expr( ):  // return value of the expression
        int val1, val2; // two values
        switch token {
                case +:    token := next_token( );
                           val1 =  expr( ) ;
                           val2 = expr( );
                           return val1 + val2;
                case 0..9:
                           return digit( );
                …
        }

int digit( ):  // return value of constant
        switch token {
                case 1: token := next_token( ); return 1;
                case 2: token := next_token( ); return 2;
               …
        }
```

What happens when you parse expression
“ + **2** + **1 2**”

The parsing produces:
?

Example: Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
int expr:  // return target register of operation
        int target_reg; // "fresh" register
        int reg1, reg2; // other registers
        switch token {
        case +:    token := next_token ( );
                   target_reg = next_register ( );
                   reg1 = expr ( );
                   reg2 = expr ( );
                   CodeGen(ADD, reg1, reg2, target_reg);
                   return target_reg;
        case 0..9:
                   return digit ( );
        …
}
```

ADD rReg1, rReg2 => rTarget

Example: Code Generator (cont.)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
int digit:  // return target register of operation
        int target_reg;  // "fresh" register
        switch token {
                case 1:     token := next_token ( );
                            target_reg = next_register ( );
                            CodeGen(LOADI, 1, target_reg);
                            return target_reg;
          case 2:     token := next_token ( );
                            target_reg = next_register ( );
                            CodeGen(LOADI, 2, target_reg);
                            return target_reg;
        …
}
```

LOADI 1 => rTarget

Example: Code Generator (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

Assumption:

First call to next_register() will return 1

The parsing produces:

```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
ADD  r2, r3 => r1
```

Example: Type Checker

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
string expr:  // returns type expression
        string  type1, type2; // other type expressions
        switch token {
                case +:    token := next_token ( )
                           type1 = expr ( );
                           type2 = expr ( );
                           if (type1 == “int” and type2 == “int”){
                                return “int”;
                           } else{
                                return “error”;
                           }
        case 0..9:
                           return digit ( );
        …
}
```

Example: Type Checker (cont.)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

```
string digit:  // returns type expression
        switch token {
                case 1:     token := next_token ( );
                            return "int";
                case 2:     token := next_token ( );
                            return "int";
        …
        }
```

Example: Type Checker (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

The parsing produces:

“int”

Example: Basic Performance Predictor

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
int expr:  // returns cycles needed to compute expression
        int cyc1, cyc2; // subexpression cycles
        switch token {
                case +:    token := next_token ( )
                           cyc1 = expr( );
                           cyc2 = expr( );
                           return cyc1 + cyc2 + 2; // ADD takes 2 cycles
        case 0..9:
                           return digit ( );
        …
}
```

Example: Basic Performance Predictor (cont.)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

```
int digit:  // returns cycles
        switch token {
                case 1:     token := next_token ( );
                            return 1; // LOADI takes 1 cycle
        case 2:     token := next_token ( );
                            return 1; // LOADI takes 1 cycle
…
}
```

Example: Basic Performance Predictor (cont.)

What happens when you parse subprogram

“ + **2** + **1 2**” ?

The parsing produces:

7

Next Lecture

Things to do:

- Start programming in C.
- Read Scott, Chapter 3.1 - 3.3; ALSU 7.1
- Read Scott, Chapter 8.1 - 8.2; ALSU 7.1 - 7.3