

Parsing

A.k.a. **Syntax Analysis**

- Recognize *sentences* in a language.
- Discover the structure of a document/program.
- Construct (implicitly or explicitly) a tree (called as a *parse tree*) to represent the structure.
- The parse tree is used to guide translation.

Grammars

The syntactic structure of a language is defined using *grammars*.

- Grammars (like regular expressions) specify a set of strings over an alphabet.
- Efficient *recognizers* (automata) can be constructed to determine whether a string is in the language.
- Language heirarchy:
 - Finite Languages (FL)
Enumeration
 - Regular Languages (RL $\supset$ FL)
Regular Expressions
 - **Context-Free Languages (CFL $\supset$ RL)**
Context-Free Grammars

Regular Languages

Languages represented by regular expressions $\equiv$ Languages recognized by finite automata

Examples:

- ✓ $\{a, b, c\}$
- ✓ $\{\epsilon, a, b, aa, ab, ba, bb, \dots\}$
- ✓ $\{(ab)^n \mid n \geq 0\}$
- ✗ $\{a^n b^n \mid n \geq 0\}$

Context-Free Grammars

- **Terminal Symbols:** Tokens
- **Nonterminal Symbols:** set of strings made up of tokens
- **Productions:** Rules for constructing the set of strings associated with nonterminal symbols.

Example: $Stmt \longrightarrow \text{while } Expr \text{ do } Stmt$

- **Start symbol:** a nonterminal symbol that represents the set of all strings in the language.

Grammars

Notation where recursion is explicit.

Examples:

- $\{\epsilon, a, b, aa, ab, ba, bb, \dots\} = \mathcal{L}((a \mid b)^*)$:

E	$\longrightarrow$	a	Notational shorthand:
E	$\longrightarrow$	b	$E \longrightarrow a \mid b$
S	$\longrightarrow$	ϵ	$S \longrightarrow \epsilon \mid ES$
S	$\longrightarrow$	ES	

- $\{a^n b^n \mid n \geq 0\}$:

$$S \longrightarrow \epsilon$$

$$S \longrightarrow aSb$$

- $\{w \mid \text{no. of } a\text{'s in } w = \text{no. of } b\text{'s in } w\}$:

Not expressible in CFG .

The First Useful Example

$$E \longrightarrow E + E$$

$$E \longrightarrow E - E$$

$$E \longrightarrow E * E$$

$$E \longrightarrow E / E$$

$$E \longrightarrow (E)$$

$$E \longrightarrow \text{id}$$

$$\mathcal{L}(E) = \{\text{id}, \text{id} + \text{id}, \text{id} - \text{id}, \dots, \text{id} + (\text{id} * \text{id}) - \text{id}, \dots\}$$

Context-Free Grammars: Notations

Production: rule with *nonterminal* symbol on left hand side, and a (possibly empty) sequence of terminal or nonterminal symbols on the right hand side.

Notations:

- **Terminals:** lower case letters, digits, punctuation
- **Nonterminals:** Upper case letters
- **Arbitrary Terminals/Nonterminals:** X, Y, Z
- **Strings of Terminals:** u, v, w
- **Strings of Terminals/Nonterminals:** α, β, γ
- **Start Symbol:** S

Derivations

Grammar:	$E \longrightarrow E + E$ $E \longrightarrow \text{id}$
-----------------	--

E derives $\text{id} + \text{id}$:

$$\begin{aligned}
 E &\Longrightarrow E + E \\
 &\Longrightarrow E + \text{id} \\
 &\Longrightarrow \text{id} + \text{id}
 \end{aligned}$$

- $\alpha A \beta \Longrightarrow \alpha \gamma \beta$ iff $A \longrightarrow \gamma$ is a production in the grammar.
- $\alpha \xRightarrow{*} \beta$ if α derives β in zero or more steps.

Example: $E \xRightarrow{*} \text{id} + \text{id}$

- **Sentence:** A sequence of terminal symbols w such that $S \xRightarrow{+} w$ (where S is the start symbol)
- **Sentential Form:** A sequence of terminal/nonterminal symbols α such that $S \xRightarrow{*} \alpha$

Derivations

Grammar: $E \rightarrow E + E$
 $E \rightarrow \text{id}$

- **Leftmost derivation:** Leftmost nonterminal is replaced first:

$$\begin{aligned} E &\Rightarrow \underline{E} + E \\ &\Rightarrow \text{id} + \underline{E} \\ &\Rightarrow \text{id} + \text{id} \end{aligned}$$

Written as $E \xRightarrow{*}_{lm} \text{id} + \text{id}$

- **Rightmost derivation:** Rightmost nonterminal is replaced first:

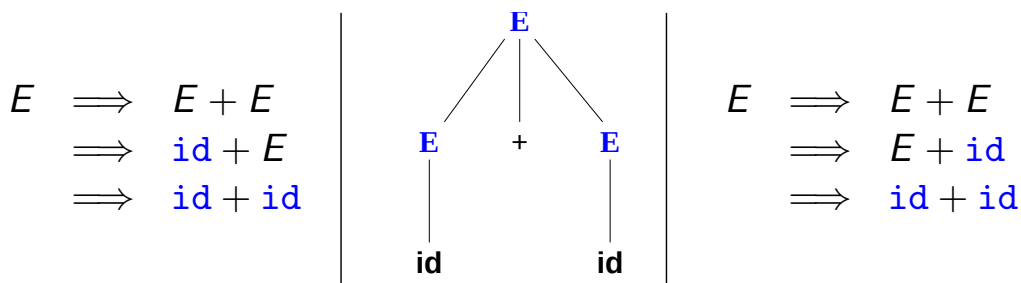
$$\begin{aligned} E &\Rightarrow E + \underline{E} \\ &\Rightarrow E + \text{id} \\ &\Rightarrow \text{id} + \text{id} \end{aligned}$$

Written as $E \xRightarrow{*}_{rm} \text{id} + \text{id}$

Parse Trees

A Parse Tree is a graphical representation of a derivation

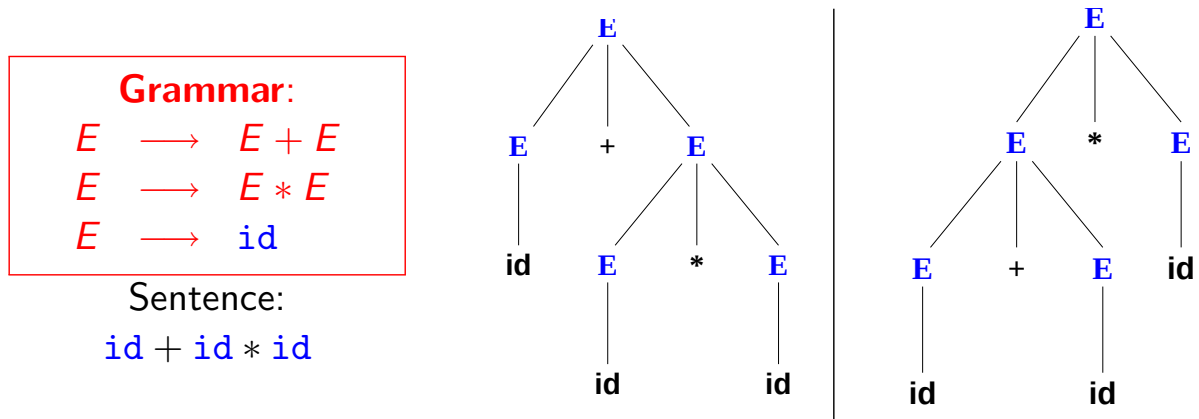
Grammar: $E \rightarrow E + E$
 $E \rightarrow \text{id}$



A **Parse Tree** succinctly captures the structure of a sentence.

Ambiguity

A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

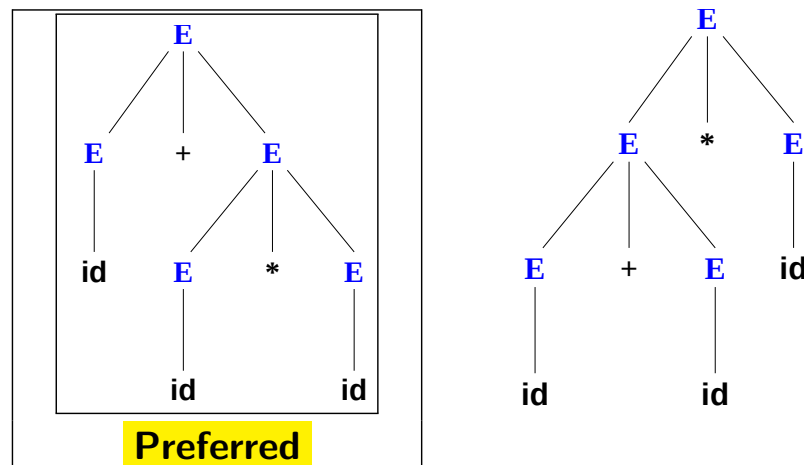


Disambiguation

Express Preference for one parse tree over others.

Example: $id + id * id$

The usual precedence of $*$ over $+$ means:



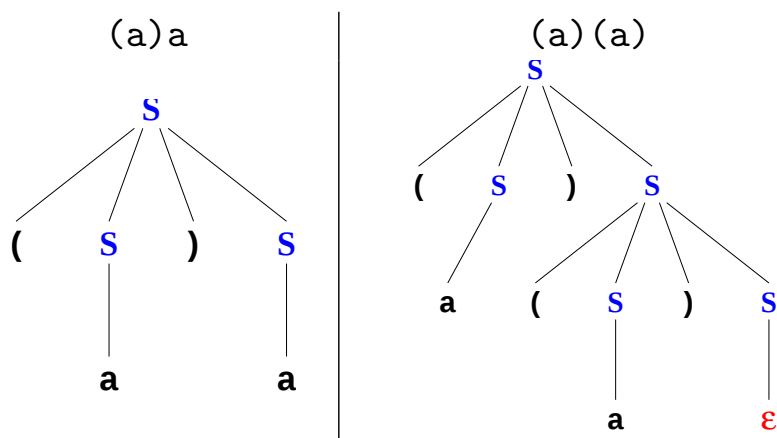
Parsing

Construct a parse tree for a given string.

$$S \longrightarrow (S)S$$

$$S \longrightarrow a$$

$$S \longrightarrow \epsilon$$



Recursive-Descent Parsing

A Procedure for Parsing

Grammar: $S \longrightarrow a$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A:
      consume(TOKEN_A);
      return;
    default:
      /* Parse Error */
  }
}

```

A Procedure for Parsing (Contd.)

Grammar:	$S \rightarrow a$
	$S \rightarrow \epsilon$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A: /* Production 1 */
      consume(TOKEN_A);
      return;
    case TOKEN_EOF: /* Production 2 */
      return;
    default:
      /* Parse Error */
  }
}

```

A Procedure for Parsing (Contd.)

Grammar:	$S \rightarrow (S)S$
	$S \rightarrow a$
	$S \rightarrow \epsilon$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_OPEN_PAREN: /* Production 1 */
      consume(TOKEN_OPEN_PAREN);
      parse_S();
      consume(TOKEN_CLOSE_PAREN);
      parse_S();
      return;
  }
  /* Continued on next page */
}

```

A Procedure for Parsing (contd.)

Grammar:	$S \longrightarrow (S)S$
	$S \longrightarrow a$
	$S \longrightarrow \epsilon$

```

case TOKEN_A: /* Production 2 */
    consume(TOKEN_A);
    return;
case TOKEN_CLOSE_PAREN:
case TOKEN_EOF: /* Production 3 */
    return;
default:
    /* Parse Error */

```

Predictive Parsing: Restrictions

May not be able to choose a unique production

$$\begin{aligned}
 S &\longrightarrow a B d \\
 B &\longrightarrow b \\
 B &\longrightarrow bc
 \end{aligned}$$

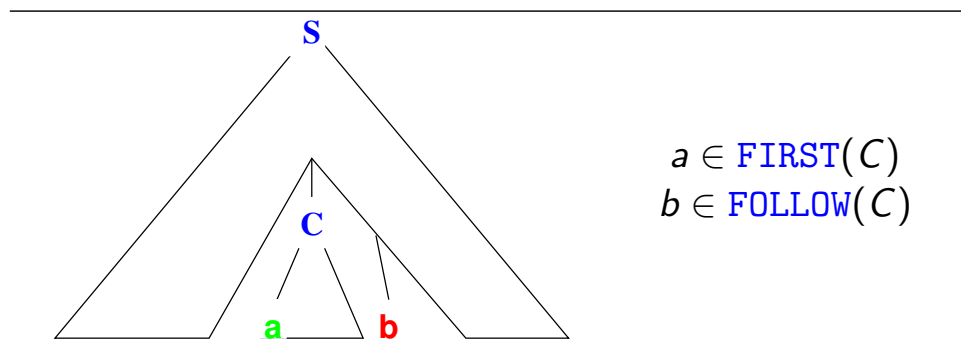
In general, we may need a backtracking parser:

Recursive Descent Parsing

FIRST and FOLLOW

Grammar: $S \rightarrow (S)S \mid a \mid \epsilon$

- **FIRST**(X) = First character of any string that can be derived from X
 $\text{FIRST}(S) = \{ (, a, \epsilon \}$.
- **FOLLOW**(A) = First character that, in any derivation of a string in the language, appears immediately after A .
 $\text{FOLLOW}(S) = \{), \text{EOF} \}$



FIRST and FOLLOW

Grammar: $A \rightarrow a$ $S \rightarrow A S B$
 $B \rightarrow b$ $S \rightarrow \epsilon$

$\text{FIRST}(X)$: First terminal in some α such that
 $X \xRightarrow{*} \alpha$.
 $\text{FOLLOW}(A)$: First terminal in some β such that
 $S \xRightarrow{*} \alpha A \beta$.

$\text{FIRST}(S) = \{ a, \epsilon \}$	$\text{FOLLOW}(S) = \{ b, \text{EOF} \}$
$\text{FIRST}(A) = \{ a \}$	$\text{FOLLOW}(A) = \{ a, b \}$
$\text{FIRST}(B) = \{ b \}$	$\text{FOLLOW}(B) = \{ b, \text{EOF} \}$

Definition of FIRST

Grammar:

$$\begin{array}{ll} A \longrightarrow a & S \longrightarrow A S B \\ B \longrightarrow b & S \longrightarrow \epsilon \end{array}$$

$FIRST(\alpha)$ is the smallest set such that

$\alpha =$	Property of $FIRST(\alpha)$
a , a terminal	$a \in FIRST(\alpha)$
A , a nonterminal	$A \longrightarrow \epsilon \in G \implies \epsilon \in FIRST(\alpha)$ $A \longrightarrow \beta \in G, \beta \neq \epsilon \implies FIRST(\beta) \subseteq FIRST(\alpha)$
$X_1, X_2, \dots, X_k$, a string of terminals and nonterminals	$FIRST(X_1) - \{\epsilon\} \subseteq FIRST(\alpha)$ $FIRST(X_i) \subseteq FIRST(\alpha)$ if $\forall j < i \quad \epsilon \in FIRST(X_j)$ $\epsilon \in FIRST(\alpha)$ if $\forall j < k \quad \epsilon \in FIRST(X_j)$

$FIRST(A) \supseteq FIRST(a) \supseteq \{a\}$ $FIRST(B) \supseteq FIRST(b) \supseteq \{b\}$
 $FIRST(S) \supseteq \{\epsilon\}$, and $FIRST(S) \supseteq FIRST(\{ASB\}) \supseteq FIRST(A) \supseteq \{a\}$

Definition of FOLLOW

Grammar:

$$\begin{array}{ll} A \longrightarrow a & S \longrightarrow A S B \\ B \longrightarrow b & S \longrightarrow \epsilon \end{array}$$

$FOLLOW(A)$ is the smallest set such that

A	Property of $FOLLOW(A)$
$= S$, the start symbol	$EOF \in FOLLOW(S)$ Book notation: $\$ \in FOLLOW(S)$
$B \longrightarrow \alpha A \beta \in G$	$FIRST(\beta) - \{\epsilon\} \subseteq FOLLOW(A)$
$B \longrightarrow \alpha A$, or $B \longrightarrow \alpha A \beta, \epsilon \in FIRST(\beta)$	$FOLLOW(B) \subseteq FOLLOW(A)$

$FOLLOW(S) \supseteq \{EOF\}$, and $FOLLOW(S) \supseteq FIRST(B) - \epsilon \supseteq \{b\}$
 $FOLLOW(A) \supseteq FIRST(SB) - \epsilon \supseteq \{a, b\}$
 $FOLLOW(B) \supseteq FOLLOW(S) \supseteq \{b, EOF\}$

Parsing Table

Grammar:

$A \rightarrow a$	$S \rightarrow A S B$
$B \rightarrow b$	$S \rightarrow \epsilon$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A: /* Production 3 */
      parse_A();
      parse_S();
      parse_B();
      return;
    case TOKEN_B:
    case TOKEN_EOF: /* Production 4 */
      return;
  }
}

```

Parsing Table:

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
S	$S \rightarrow A S B$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Table-driven Parsing

Grammar:

$A \rightarrow a$	$S \rightarrow A S B$
$B \rightarrow b$	$S \rightarrow \epsilon$

Parsing Table:

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
S	$S \rightarrow A S B$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
A	$A \rightarrow a$		
B		$B \rightarrow b$	

Nonrecursive Parsing

Instead of recursion,

use an explicit *stack* along with the parsing table.

Data objects:

- **Parsing Table:** $M(A, a)$, a two-dimensional array, dimensions indexed by nonterminal symbols (A) and terminal symbols (a).
- A **Stack** of terminal/nonterminal symbols
- **Input stream** of tokens

The above data structures manipulated using a *table-driven parsing program*.

Table-driven Parsing: a Sketch

stack initialized to *EOF*.

```
while (! stack.isEmpty()) {
    X = stack.top();
    if (X is a terminal symbol)
        consume(X);
    else /* X is a nonterminal */
        if (M[X, input_token] = X → Y1, Y2, ..., Yk) {
            stack.pop();
            for i = k downto 1 do
                stack.push(Yi);
        }
    else
        /* Syntax Error */
}
```

Constructing Parsing Table

Grammar:

$$\begin{array}{ll} A \longrightarrow a & S \longrightarrow A S B \\ B \longrightarrow b & S \longrightarrow \epsilon \end{array}$$

$$\begin{array}{ll} \text{First}(S) = \{ a, \epsilon \} & \text{Follow}(S) = \{ b, \text{EOF} \} \\ \text{First}(A) = \{ a \} & \text{Follow}(A) = \{ a, b \} \\ \text{First}(B) = \{ b \} & \text{Follow}(B) = \{ b, \text{EOF} \} \end{array}$$

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
<i>S</i>	$S \longrightarrow A S B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
<i>A</i>	$A \longrightarrow a$		
<i>B</i>		$B \longrightarrow b$	

A Procedure to Construct Parsing Tables

FIRST(*X*): First terminal in some α such that
 $X \xRightarrow{*} \alpha$.
FOLLOW(*A*): First terminal in some β such that
 $S \xRightarrow{*} \alpha A \beta$.

```

Algorithm table_construct(G) {
  for each  $A \longrightarrow \alpha \in G$  {
    for each  $a \in \text{FIRST}(\alpha)$  such that  $a \neq \epsilon$ 
      add  $A \longrightarrow \alpha$  to  $M[A, a]$ ;
    if  $\epsilon \in \text{FIRST}(\alpha)$ 
      for each  $b \in \text{FOLLOW}(A)$ 
        add  $A \longrightarrow \alpha$  to  $M[A, b]$ ;
  }}

```

Parsing Table: Another Example

Grammar:

$$\begin{array}{lcl} S & \longrightarrow & (S)S \\ S & \longrightarrow & a \\ S & \longrightarrow & \epsilon \end{array}$$

$\text{FIRST}(S) = \{ '(', a, \epsilon \}$
 $\text{FOLLOW}(S) = \{ ')', \text{EOF} \}$

INPUT SYMBOL			
a	(	)	EOF
$S \longrightarrow a$	$S \longrightarrow (S)S$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$

LL(1) Grammar: When the grammar's recursive descent parsing table has no conflicts (i.e. each cell has at most one entry).

Recursive Descent Parsing: Restrictions

Grammar cannot be left-recursive

Example: $E \longrightarrow E + E \mid a$

```

Algorithm parse_E() {
  switch (input_token) {
    case TOKEN_A: /* Production 1 */
      parse_E();
      consume(TOKEN_PLUS);
      parse_E();
      return;
    case TOKEN_A: /* Production 2 */
      consume(TOKEN_A);
      return;
  }
}
```

Removing Left Recursion

$$\begin{aligned} A &\longrightarrow A a \\ A &\longrightarrow b \end{aligned}$$

$$\mathcal{L}(A) = \{b, ba, baa, baaa, baaaa, \dots\}$$

$$\begin{aligned} A &\longrightarrow bA' \\ A' &\longrightarrow aA' \\ A' &\longrightarrow \epsilon \end{aligned}$$

Removing Left Recursion: Another Example

$$\begin{aligned} E &\longrightarrow E + E \\ E &\longrightarrow \text{id} \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} E &\longrightarrow \text{id } E' \\ E' &\longrightarrow + \text{id } E' \\ E' &\longrightarrow \epsilon \end{aligned}$$

Parsing with LL(1) Grammars: Example 1

	a	b	EOF
S	$S \rightarrow A S B$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
A	$A \rightarrow a$		
B		$B \rightarrow b$	

Stack	Input Stream	Rule	Derivation
\$S	a a b b\$	$S \rightarrow A S B$	$S \Rightarrow A S B$
\$B S A	a a b b\$	$A \rightarrow a$	$\Rightarrow a S B$
\$B S a	a a b b\$		
\$B S	a b b\$	$S \rightarrow A S B$	$\Rightarrow a A S B B$
\$B B S A	a b b\$	$A \rightarrow a$	$\Rightarrow a a S B B$
\$B B S a	a b b\$		
\$B B S	b b\$	$S \rightarrow \epsilon$	$\Rightarrow a a B B$
\$B B	b b\$	$B \rightarrow b$	$\Rightarrow a a b B$
\$B b	b b\$		
\$B	b\$	$B \rightarrow b$	$\Rightarrow a a b b$
\$b	b\$		
\$	\$		

Parsing with LL(1) Grammars: Example 2

NONTERMINAL	INPUT SYMBOL		
	id	+	EOF
E	$E \rightarrow id E'$		
E'		$E' \rightarrow + E E'$	$E' \rightarrow \epsilon$

Stack	Input Stream	Rule	Derivation
\$E	id + id\$	$E \rightarrow id E'$	$E \Rightarrow id E'$
\$E' id	id + id\$		
\$E'	+ id\$	$E' \rightarrow + E E'$	$\Rightarrow id + id E'$
\$E' id +	+ id\$		
\$E' id	id\$		
\$E'	\$	$E' \rightarrow \epsilon$	$\Rightarrow id+id$
\$	\$		

LL(1) Derivations

Left to Right Scan of input

Leftmost Derivation

(1) look ahead 1 token at each step

Alternative characterization of LL(1) Grammars:

Whenever $A \longrightarrow \alpha \mid \beta \in G$

① $FIRST(\alpha) \cap FIRST(\beta) = \{ \}$, and

② if $\alpha \xRightarrow{*} \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \{ \}$.

Corollary: No Ambiguous Grammar is LL(1).

Other Parsing Algorithms

- LR-, LALR-, SLR, ...
Table-driven “bottom-up” parsers (builds parse trees from leaves to root).
Parsing time is *linear* in the length of the input string.
- The set of LR(k) grammars includes all LL(k) grammars (but some grammars are not LR(k) for any k).
- Operator precedence parsers
- Chart parsers (used in Natural Language Processing)
- Cocke-Kasami-Younger & Earley parsers: can handle arbitrary context-free grammars
(but the parsers may take quadratic or cubic time).