

# ДЗ 3

---

Студент: Султанов Артур Радикович (367553), группа Р3313

Вариант: 17

Грамматика:

```
S → abA | acBA | aaC  
A → AAa | Aa | Ab | b  
B → bcB | bbBB | bb  
C → Cc | c
```

## Устранение левой рекурсии

Правило S

```
S → abA | acBA | aaC
```

С правилом S все хорошо, рекурсии нет.

Правило A

```
A → AAa | Aa | Ab | b
```

Разделим на 2 части:

```
A → AAa | Aa | Ab  
A → b
```

Добавим новый нетерминал D, заменим все правила с A в левой части, получим:

```
A → b  
A → bD  
D → a | b | Aa  
D → aD | bD | AaD
```

Упростим:

$$\begin{aligned} A &\rightarrow b \mid bD \\ D &\rightarrow a \mid aD \mid b \mid bD \mid Aa \mid AaD \end{aligned}$$

Сократим:

$$\begin{aligned} A &\rightarrow bD \\ D &\rightarrow aD \mid bD \mid AaD \mid \epsilon \end{aligned}$$

Правило **B**

$$B \rightarrow bcB \mid bbBB \mid bb$$

Левой рекурсии здесь нет

Правило **C**

$$C \rightarrow Cc \mid c$$

Разделим на 2 части:

$$\begin{aligned} C &\rightarrow c \\ C &\rightarrow Cc \end{aligned}$$

Добавим новый нетерминал **E**, заменил все правила с **C** в левой части, получим:

$$\begin{aligned} C &\rightarrow c \\ C &\rightarrow cE \\ E &\rightarrow c \\ E &\rightarrow cE \end{aligned}$$

Упростим:

$$\begin{aligned} C &\rightarrow c \mid cE \\ E &\rightarrow c \mid cE \end{aligned}$$

Сократим:

```
C → cE
E → cE | ε
```

## Итого получаем

```
S → abA | acBA | aaC
A → bD
D → aD | bD | AaD | ε
B → bcB | bbBB | bb
C → cE
E → cE | ε
```

## Левая факторизация

Далее, стоит устранить повторяющиеся начала (состоящие из нетерминалов) правых частей правил. Для этого проведем левую факторизацию.

### S

```
S → abA | acBA | aaC
```

Самый длинный общий префикс - **a**. Он не пустой, выделим новый нетерминал **F** и заменим правила:

```
S → aF
F → bA | cBA | aC
```

Новое правило в факторизации не нуждается.

### A

```
A → bD
```

Не требуется.

### D

```
D → aD | bD | AaD | ε
```

Не требуется.

## B

$$B \rightarrow bcB \mid bbBB \mid bb$$

Самый длинный общий префикс - **b**. Он не пустой, выделим новый нетерминал **G** и заменим правила:

$$\begin{aligned} B &\rightarrow bG \\ G &\rightarrow cB \mid bBB \mid b \end{aligned}$$

Новое правило нуждается в факторизации. Общий префикс - **b**. Не пустой, выделим нетерминал **H** и заменим правила:

$$\begin{aligned} B &\rightarrow bG \\ G &\rightarrow cB \mid bH \\ H &\rightarrow BB \mid \epsilon \end{aligned}$$

## C

$$C \rightarrow cE$$

Не требуется.

## E

$$E \rightarrow cE \mid \epsilon$$

Не требуется.

## Итого получаем

$$\begin{aligned} S &\rightarrow aF \\ F &\rightarrow bA \mid cBA \mid aC \\ A &\rightarrow bD \\ D &\rightarrow aD \mid bD \mid AaD \mid \epsilon \\ B &\rightarrow bG \\ G &\rightarrow cB \mid bH \\ H &\rightarrow BB \mid \epsilon \\ C &\rightarrow cE \\ E &\rightarrow cE \mid \epsilon \end{aligned}$$

## Построение множеств **FIRST**

**FIRST(X)** – это множество терминальных символов, с которых начинаются цепочки, выводимые из **X**.

Нетерминал **S**

$$\text{FIRST}(S) = \{a\}$$

Нетерминал **F**

$$\text{FIRST}(F) = \{a, b, c\}$$

Нетерминал **A**

$$\text{FIRST}(A) = \{b\}$$

Нетерминал **D**

$$\text{FIRST}(D) = \{a, b, \epsilon\}$$

Также один из вариантов начинается с **A**, соответственно добавим сюда **FIRST(A)**, получим:

$$\text{FIRST}(D) = \{a, b, \epsilon\} \cup \text{FIRST}(A)$$

$$\text{FIRST}(D) = \{a, b, \epsilon\}$$

Нетерминал **B**

$$\text{FIRST}(B) = \{b\}$$

Нетерминал **G**

$$\text{FIRST}(G) = \{b, c\}$$

Нетерминал **H**

$$\text{FIRST}(H) = \{\epsilon\} \cup \text{FIRST}(B) = \{\epsilon\} \cup \{b\} = \{b, \epsilon\}$$

Нетерминал **C**

$$\text{FIRST}(C) = \{c\}$$

Нетерминал **E**

$$\text{FIRST}(E) = \{c, \epsilon\}$$

## Построение множеств **FOLLOW**

Множество **FOLLOW(X)** для нетерминала **X** определяется как множество терминальных символов **b**, которые в sentential forms для некоторой грамматики могут располагаться непосредственно справа от **X**;  $S \Rightarrow * \alpha A b \beta$ .

## Нетерминал S

$FOLLOW(S) = \{\$ \}$

## Нетерминал F

$FOLLOW(F) = \{ \} \cup FOLLOW(S) = \{\$ \}$

## Нетерминал A

$FOLLOW(A) = \{ \} \cup FIRST(a) \cup FOLLOW(F) = \{a, \$ \}$  ( $FIRST(a)$  пришло из  $D \rightarrow AaD$ )

## Нетерминал D

$FOLLOW(D) = \{ \} \cup FOLLOW(A) = \{\$, a \}$  (т.к. есть правило  $A \rightarrow bD$  и  $isnullable(\beta) = true$ )

## Нетерминал B

$FOLLOW(B) = \{ \} \cup FOLLOW(G) \cup FIRST(B) = \{b \}$

## Нетерминал G

$FOLLOW(G) = \{ \} \cup FOLLOW(B) = \{b \}$

## Нетерминал H

$FOLLOW(H) = \{ \} \cup FOLLOW(G) = \{b \}$

## Нетерминал C

$FOLLOW(C) = \{ \} \cup FOLLOW(F) = \{\$ \}$

## Нетерминал E

$FOLLOW(E) = \{ \} \cup FOLLOW(C) = \{\$ \}$

## Построение таблицы анализатора

ВНИМАНИЕ: правило  $A \rightarrow bD$  было заменено на  $A \rightarrow dD$  (по согласованию с преподавателем) - во избежание конфликта из  $D$  в  $b$ . Получаем:

```

S → aF
F → bA | cBA | aC
A → dD
D → aD | bD | AaD | ε
B → bG
G → cB | bH
H → BB | ε
C → cE
E → cE | ε

```

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	\$\$\$
<i>S</i>	<i>S</i> → <i>aF</i>			
<i>F</i>	<i>F</i> → <i>aC</i>	<i>F</i> → <i>bA</i>	<i>F</i> → <i>cBA</i>	
<i>A</i>				<i>A</i> → <i>dD</i>
<i>D</i>	<i>D</i> → <i>aD</i>	<i>D</i> → <i>bD</i>	<i>D</i> → <i>AaD</i>	<i>D</i> → $\epsilon$
<i>B</i>	<i>B</i> → <i>bG</i>			
<i>G</i>	<i>G</i> → <i>bH</i>		<i>G</i> → <i>cB</i>	
<i>H</i>	<i>H</i> → <i>BB</i>			<i>H</i> → $\epsilon$
<i>C</i>	<i>C</i> → <i>cE</i>			
<i>E</i>	<i>E</i> → <i>cE</i>			<i>E</i> → $\epsilon$

Программная реализация анализатора

Пример запуска:

```
> ./ll1.elf
[OK] '': got expected error('at 0: unexpected end of text: 'a' is
expected')
[OK] 'qqq': got expected error('at 0: unexpected symbol 'q'')
[OK] 'a': got expected error('at 1: unexpected end of text')
[OK] 'b': got expected error('at 0: unexpected symbol 'b': 'a' is
expected')
[OK] 'c': got expected error('at 0: unexpected symbol 'c': 'a' is
expected')
[OK] 'd': got expected error('at 0: unexpected symbol 'd': 'a' is
expected')
[OK] 'abda': success
[OK] 'acbbd': success
[OK] 'aacccccccccc': success
```

Смысловая часть исходного кода (полностью доступен по [ссылке](#)):

```
enum RuleType {
    VALID,
    ERROR,
};

struct Rule {
    // Terminating symbol
    char t;
    // Rule type
    enum RuleType type;
    union {
```

```

        // Right path of the rule
        char *right;
        // Error message
        char *error;
    };
};

struct Row {
    // Non-Terminating symbol
    char nt;
    // Rules
    struct Rule *rules[6]; // abcd$ and NULL
};

struct Row TABLE[] = {
    {
        .nt = 'S',
        .rules = {
            &(struct Rule){ 'a', VALID, .right = "aF"},
            &(struct Rule){ 'b', ERROR, .error = ERR_EXPECTED(a)},
            &(struct Rule){ 'c', ERROR, .error = ERR_EXPECTED(a)},
            &(struct Rule){ 'd', ERROR, .error = ERR_EXPECTED(a)},
            &(struct Rule){ EOF, ERROR, .error = ERR_EXPECTED(a)},
            NULL,
        },
    },
    {
        .nt = 'F',
        .rules = {
            &(struct Rule){ 'a', VALID, .right = "aC"},
            &(struct Rule){ 'b', VALID, .right = "bA"},
            &(struct Rule){ 'c', VALID, .right = "cBA"},
            &(struct Rule){ 'd', ERROR, .error = ERR_NO_REASON},
            &(struct Rule){ EOF, ERROR, .error = ERR_NO_REASON},
            NULL,
        },
    },
    {
        .nt = 'A',
        .rules = {
            &(struct Rule){ 'a', ERROR, .error = ERR_EXPECTED(d)},
            &(struct Rule){ 'b', ERROR, .error = ERR_EXPECTED(d)},
            &(struct Rule){ 'c', ERROR, .error = ERR_EXPECTED(d)},
            &(struct Rule){ 'd', VALID, .right = "dD"},
            &(struct Rule){ EOF, ERROR, .error = ERR_EXPECTED(d)},
            NULL,
        },
    },
    {
        .nt = 'D',
        .rules = {
            &(struct Rule){ 'a', VALID, .right = "aD"},
            &(struct Rule){ 'b', VALID, .right = "bD"},
            &(struct Rule){ 'c', ERROR, .error = ERR_NO_REASON},

```



```

        &(struct Rule){'d', VALID, .right = "AaD"},
        &(struct Rule){EOF, VALID, .right = ""},
        NULL,
    }
},
{
    .nt = 'B',
    .rules = {
        &(struct Rule){'a', ERROR, .error = ERR_EXPECTED(b)},
        &(struct Rule){'b', VALID, .right = "bG"},
        &(struct Rule){'c', ERROR, .error = ERR_EXPECTED(b)},
        &(struct Rule){'d', ERROR, .error = ERR_EXPECTED(b)},
        &(struct Rule){EOF, ERROR, .error = ERR_EXPECTED(b)},
        NULL,
    }
},
{
    .nt = 'G',
    .rules = {
        &(struct Rule){'a', ERROR, .error = ERR_NO_REASON},
        &(struct Rule){'b', VALID, .right = "bH"},
        &(struct Rule){'c', VALID, .right = "cB"},
        &(struct Rule){'d', ERROR, .error = ERR_NO_REASON},
        &(struct Rule){EOF, ERROR, .error = ERR_NO_REASON},
        NULL,
    }
},
{
    .nt = 'H',
    .rules = {
        &(struct Rule){'a', ERROR, .error = ERR_EOF_OR_EXPECTED(b)},
        &(struct Rule){'b', VALID, .right = "BB"},
        &(struct Rule){'c', ERROR, .error = ERR_EOF_OR_EXPECTED(b)},
        &(struct Rule){'d', ERROR, .error = ERR_EOF_OR_EXPECTED(b)},
        &(struct Rule){EOF, VALID, .right = ""},
        NULL,
    },
},
{
    .nt = 'C',
    .rules = {
        &(struct Rule){'a', ERROR, .error = ERR_EXPECTED(c)},
        &(struct Rule){'b', ERROR, .error = ERR_EXPECTED(c)},
        &(struct Rule){'c', VALID, .right = "cE"},
        &(struct Rule){'d', ERROR, .error = ERR_EXPECTED(d)},
        &(struct Rule){EOF, ERROR, .error = ERR_EXPECTED(d)},
        NULL,
    },
},
{
    .nt = 'E',
    .rules = {
        &(struct Rule){'a', ERROR, .error = ERR_EOF_OR_EXPECTED(c)},
        &(struct Rule){'b', ERROR, .error = ERR_EOF_OR_EXPECTED(c)},

```

```

        &(struct Rule){'c', VALID, .right = "cE"},
        &(struct Rule){'d', ERROR, .error = ERR_EOF_OR_EXPECTED(c)},
        &(struct Rule){EOF, VALID, .right = ""},
        NULL,
    },
},
};

bool is_non_terminal(char c) {
    return 'A' <= c && c <= 'Z';
}

bool is_terminal(char c) {
    return !is_non_terminal(c);
}

struct Rule *find_rule_in_table(char non_terminal, char terminal) {
    if (terminal == '\\0') {
        terminal = EOF;
    }

    for (size_t i = 0; i < sizeof(TABLE) / sizeof(TABLE[0]); i++) {
        struct Row row = TABLE[i];
        if (row.nt != non_terminal) {
            continue;
        }

        size_t j = 0;
        while (row.rules[j] != NULL) {
            if (row.rules[j]->t == terminal) {
                return row.rules[j];
            }
            j++;
        }
    }
    return NULL;
}

// analyze_string returns NULL if s is a valid text.
// Returns an error message otherwise. You shall free() it.
char *analyze_string(const char *s) {
    stack_clear();
    stack_push('S');

    size_t length = strlen(s);
    size_t head = 0;

    while (head <= length && !stack_is_empty()) {
        char in = s[head];
        char c = stack_top();

        if (is_terminal(c)) {
            if (c == in) {
                stack_pop();
            }
        }
    }
}

```

```
        head++;
    } else {
        return fmt_err_c_is_expected(head, c, in);
    }
} else {
    struct Rule *rule = find_rule_in_table(c, in);
    if (rule == NULL) {
        return fmt_err_unexpected(head, in, "");
    }
    switch (rule->type) {
        case VALID: {
            stack_pop();
            stack_push_str(rule->right);
            break;
        }
        case ERROR: {
            struct Rule *end_rule = find_rule_in_table(c, EOF);
            if (end_rule == NULL) {
                return fmt_err_unexpected(head, in, "");
            }
            if (end_rule->type == VALID) {
                stack_pop();
                break;
            }
            return fmt_err_unexpected(head, in, rule->error);
        }
    }
}

if (stack_is_empty()) {
    return NULL;
}

char c = stack_top();
if (is_terminal(c)) {
    return fmt("at %zu: unexpected end of text: '%c' is expected",
head, c);
} else {
    return fmt("at %zu: unexpected end of text", head);
}
}
```