Документация и теория работы парсера регулярных выражений

Супрун Артём Сергеевич

14 апреля 2025 г.

Содержание

1 Введение

В данном документе представлена теоретическая основа и реализация парсера регулярных выражений на языке C. Использование метода рекурсивного спуска позволяет поэтапно обрабатывать различные компоненты выражения — от литералов до операторов конкатенации, альтернативы и повторения (оператор `*`).

2 Теоретическая основа

2.1 Метод рекурсивного спуска

Рекурсивный спуск — это метод синтаксического анализа, при котором для каждой нетерминальной грамматики создаётся отдельная функция. Каждая функция принимает указатель на текущую позицию во входной строке и обновляет его по мере анализа. Такой подход позволяет эффективно обрабатывать вложенные структуры и обеспечивать понятное разделение логики.

Грамматика регулярных выражений в рассматриваемом примере выглядит следующим образом:

$$\begin{aligned} \text{Expression} &\rightarrow \text{Term}\,\{\,'|'\,\text{Term}\,\} \\ \text{Term} &\rightarrow \text{Factor}\,\{\,\text{Factor}\,\} \\ \text{Factor} &\rightarrow \text{Base}\,\{\,'*'\,\} \\ \text{Base} &\rightarrow' (' \,\text{Expression}\, ')' \mid \text{Литерал} \end{aligned}$$

Каждая конструкция реализована отдельной функцией, создающей соответствующие узлы дерева разбора.

2.2 Обработка регулярных выражений

Регулярные выражения — мощный инструмент для поиска и обработки строк. В нашем случае синтаксический анализ включает:

- **Литералы** — отдельные символы.
- **Конкатенацию** — последовательное соединение символов и подвыражений.
- **Альтернативу** — выбор между несколькими выражениями с помощью оператора `'|'`.
- **Клини-стар** — оператор повторения (`'*'`), позволяющий повторять предыдущий элемент неограниченное число раз.

3 Описание структуры данных

Для представления дерева разбора используются структура `Node` и перечисление возможных типов узлов.

3.1 Типы узлов

Определены следующие типы узлов:

- **NODE_LITERAL:** узел, представляющий конкретный символ.

- **NODE_CONCAT:** узел для конкатенации двух подвыражений.
- **NODE_ALTERNATION:** узел для альтернативы (оператор `'|'`).
- **NODE_STAR:** узел для оператора повторения (оператор `'*'`).

3.2 Структура узла

Структура `Node` включает следующие поля:

- `type` — тип узла.
- `c` — символ, используемый для узлов типа литерал.
- `left` — указатель на левый подузел (например, первый операнд).
- `right` — указатель на правый подузел (например, второй операнд для конкатенации или альтернативы).

Пример определения:

```
typedef enum {
    NODE_LITERAL,
    NODE_CONCAT,
    NODE_ALTERNATION,
    NODE_STAR
} NodeType;

typedef struct Node {
    NodeType type;
    char c; //
    struct Node *left; //
    struct Node *right; //
} Node;
```

Листинг 1: Определения типов узлов и структуры

4 Функции обработки ошибок и создания узлов

4.1 Функция `error`

Функция `error` выводит сообщение об ошибке. При наличии символа он включается в сообщение, после чего программа завершается с кодом ошибки.

```
void error(const char *message, char symbol) {
    if (symbol != '\0') {
        fprintf(stderr, ": %s '%c'\n", message, symbol);
    } else {
        fprintf(stderr, ": %s\n", message);
    }
    exit(EXIT_FAILURE);
}
```

Листинг 2: Функция обработки ошибок

4.2 Функция `create_node`

Функция `create_node` выделяет память для нового узла, инициализирует его поля и возвращает указатель на узел. При неудачном выделении памяти вызывается функция `error`.

```
Node* create_node(NodeType type, char c, Node* left, Node* right) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (!node) {
        error("  ", '\0');
    }
    node->type = type;
    node->c = c;
    node->left = left;
    node->right = right;
    return node;
}
```

Листинг 3: Функция создания узла

5 Рекурсивный спуск: функции синтаксического анализа

Ниже приведены функции, реализующие парсер с рекурсивным спуском. Каждая функция принимает аргумент `input` — указатель на указатель на текущую позицию во входной строке, что обеспечивает удобное продвижение по тексту.

5.1 `parse_expression`

Функция `parse_expression` анализирует выражение согласно грамматике:

$$\text{Expression} \rightarrow \text{Term} \, \{ \, '|' \, \text{Term} \, \}$$

При обнаружении оператора `'|'` создаётся узел типа `NODE_ALTERNATION`.

```
Node* parse_expression(const char **input) {
    Node *node = parse_term(input);
    while (**input == '|') {
        (*input)++; //   '|'
        Node *right = parse_term(input);
        node = create_node(NODE_ALTERNATION, '\0', node, right);
    }
    return node;
}
```

Листинг 4: Парсинг выражения

5.2 `parse_term`

Функция `parse_term` анализирует терм по правилу:

$$\text{Term} \rightarrow \text{Factor} \, \{ \, \text{Factor} \, \}$$

Конкатенация реализована посредством последовательного объединения результатов разбора с созданием узлов типа `NODE_CONCAT`.

```
Node* parse_term(const char **input) {
    Node *node = parse_factor(input);
    while (**input && **input != ')' && **input != '|') {
        Node *next = parse_factor(input);
        node = create_node(NODE_CONCAT, '\0', node, next);
    }
    return node;
}
```

Листинг 5: Парсинг терма

5.3 parse_factor

Функция `parse_factor` обрабатывает фактор по правилу:

$$\text{Factor} \to \text{Base}\,\{\,'*'\,\}$$

При каждом обнаружении оператора `'*'` создаётся узел типа `NODE_STAR`.

```
Node* parse_factor(const char **input) {
    Node *node = parse_base(input);
    while (**input == '*') {
        (*input)++; //   '*'
        node = create_node(NODE_STAR, '\0', node, NULL);
    }
    return node;
}
```

Листинг 6: Парсинг фактора

5.4 parse_base

Функция `parse_base` обрабатывает базовые элементы:

- Если встречается символ `'('`, вызывается функция `parse_expression`, после чего ожидается символ `')'`.
- Если символ не является управляющим (то есть не `'('`, `')'`, `'|'`) и не является концом строки, создаётся узел типа `NODE_LITERAL`.

В случае обнаружения ошибки (например, отсутствия закрывающей скобки) вызывается функция `error`.

```
Node* parse_base(const char **input) {
    if (**input == '(') {
        (*input)++; //  '('
        Node *node = parse_expression(input);
        if (**input != ')') {
            error(" ')'", **input);
        }
        (*input)++; //  ')'
        return node;
    }
    if (**input == '\0' || **input == '|' || **input == ')') {
```

```
        error(" ", **input);
    }
    char c = **input;
    (*input)++; //
    return create_node(NODE_LITERAL, c, NULL, NULL);
}
```

Листинг 7: Парсинг базового элемента

6 Функции вывода и освобождения памяти

6.1 print_tree

Функция `print_tree` рекурсивно выводит дерево разбора в удобном для чтения формате. В зависимости от типа узла выводятся:

- Для литерала — конкретный символ.
- Для конкатенации и альтернативы — соответствующие теги (`"CONCAT("` или `"ALT("`) с рекурсивным выводом поддеревьев.
- Для оператора повторения — тег `"STAR("` с выводом содержимого.

```
void print_tree(Node *node) {
    if (!node)
        return;

    switch (node->type) {
        case NODE_LITERAL:
            printf("%c", node->c);
            break;
        case NODE_CONCAT:
            printf("CONCAT(");
            print_tree(node->left);
            printf(", ");
            print_tree(node->right);
            printf(")");
            break;
        case NODE_ALTERNATION:
            printf("ALT(");
            print_tree(node->left);
            printf(", ");
            print_tree(node->right);
            printf(")");
            break;
        case NODE_STAR:
            printf("STAR(");
            print_tree(node->left);
            printf(")");
            break;
        default:
            break;
    }
```

```
}
```

Листинг 8: Вывод дерева разбора

6.2 free_tree

После завершения работы с деревом важно освободить выделенную память. Функция `free_tree` рекурсивно проходит по дереву в порядке post-order и освобождает память для каждого узла.

```
void free_tree(Node *node) {
    if (!node)
        return;
    free_tree(node->left);
    free_tree(node->right);
    free(node);
}
```

Листинг 9: Освобождение памяти дерева

7 Основная функция программы

Функция `main` выполняет следующие шаги:

1. Проверяет, передано ли регулярное выражение в качестве аргумента командной строки.
2. Вызывает функцию `parse_expression` для построения дерева разбора.
3. При наличии не обработанных символов выводит сообщение об ошибке.
4. Выводит дерево разбора с помощью `print_tree`.
5. Освобождает память, занятую деревом, с помощью `free_tree`.

Пример кода функции `main`:

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, ": %s <_>\n", argv[0]);
        return EXIT_FAILURE;
    }
    const char *input = argv[1];
    Node *root = parse_expression(&input);
    if (*input != '\0') {
        fprintf(stderr, ":   : %s\n", input);
        free_tree(root);
        return EXIT_FAILURE;
    }
    printf("   :\n");
    print_tree(root);
    printf("\n");
    free_tree(root);
    return EXIT_SUCCESS;
```

```
18 }
```

Листинг 10: Основная функция

8 Заключение

В документе подробно рассмотрена реализация парсера регулярных выражений на языке C, основанного на методе рекурсивного спуска. Были рассмотрены:

- Теоретическая основа синтаксического анализа регулярных выражений.
- Структура данных для представления дерева разбора.
- Функции для создания узлов, обработки ошибок, построения и вывода дерева.
- Основной алгоритм разбора, реализованный с помощью функций `parse_expression`, `parse_term`, `parse_factor` и `parse_base`.

Такой подход обеспечивает эффективный синтаксический анализ входной строки и позволяет использовать построенное дерево для дальнейшей интерпретации или компиляции регулярных выражений.