Лабораторная работа №2. Ручное построение нисходящих синтаксических анализаторов. Сластин Александр М33391

Вариант 2. Регулярные выражения

- Регулярные выражения с операциями:
  - конкатенации (простая последовательная запись строк)
  - выбора (вертикальная черта)
  - замыкания Клини
- Приоритет операций стандартный. Скобки могут использоваться для изменения приоритета.
- Для обозначения базовых языков используются маленькие буквы латинского алфавита. Используйте один терминал для всех символов.

Пример: $((abc*b|a)*ab(aa|b*)b)*$

1. Разработка грамматики

Построим грамматику:

$S \to Or$
$S \to \varepsilon$

$Or \to Or|And$
$Or \to And$

$And \to AndSt$
$And \to St$

$St \to St*$
$St \to C$

$C \to (Or)$
$C \to char$

Нетерминал	Описание
S	регулярное выражение или "пустой" язык
Or	операция "или"
And	операция "и"
St	операция "замыкание Клини"
C	символ / скобки
Or	регулярное выражение / операция "или"

В грамматике есть левая рекурсия. Устраним ее.

Получится грамматика:

$S \to Or$
$S \to \varepsilon$

$Or \to AndOr'$
$Or' \to |Or$
$Or' \to \varepsilon$

$And \to StAnd'$
$And' \to And$
$And' \to \varepsilon$

$St \to CSt'$
$St' \to *St'$
$St' \to \varepsilon$

$C \to (Or)$
$C \to char$

Нетерминал	Описание
S	регулярное выражение или "пустой" язык
Or	операция "или"
Or'	продолжение операции "или"
And	операция "и"
And'	продолжение операции "и"
St	операция "замыкание Клини"
St'	продолжение операции "замыкание Клини"
C	символ / скобки

2. Построение лексического анализатора

В нашей грамматике 5 терминалов: $|$, $*$, $($, $)$, $char$.

Построим лексический анализатор.

Заведем класс $Token$ для хранения терминалов (не забудем также про конец строки):

```
class Token(Enum):
    OR = auto()
    STAR = auto()
    LBRACKET = auto()
    RBRACKET = auto()
    CHAR = auto()
    END = auto()
```

Лексический анализатор построим как генератор нах входной строкой:

```
def lexical_analyzer(input_str):
    for pos, char in enumerate(input_str):
        if char.isspace():
            continue

        token = None
         if char == "|":
            token = Token.OR
        elif char == "*":
            token = Token.STAR
        elif char == "(":
            token = Token.LBRACKET
        elif char == ")":
            token = Token.RBRACKET
        elif "a" <= char <= "z":
            token = Token.CHAR

        if token is None:
            raise ParseException(pos=pos, char=char)

        yield token, char, pos
    yield Token.END, None, len(input_str)
```

3. Построение синтаксического анализатора

Построим множества FIRST и FOLLOW для нетерминалов нашей грамматики.

Нетерминал	FIRST	FOLLOW
S	$($, $char$, ε	\$
Or	$($, $char$	\$, $)$
Or'	$\|$, ε	\$, $)$
And	$($, $char$	\$, $)$, $\|$
And'	$($, $char$, ε	\$, $)$, $\|$
St	$($, $char$	\$, $)$, $($, $char$, $\|$
St'	$*$, ε	\$, $)$, $($, $char$, $\|$
C	$($, $char$	\$, $)$, $($, $char$, $*$, $\|$

Заведем структуру данных для хранения дерева:

```
class Tree:
    def __init__(self, value, *children):
        self.value = value
        self.children = children
```

```
    def __str__(self):
        chars = []
        self._str(chars)
        return "".join(chars)

    def is_terminal(self):
        return not len(self.children) and \
               not self.value[0].isupper()

    def _str(self, chars):
        if self.is_terminal():
            chars.append(self.value)
        for child in self.children:
            child._str(chars)

    def to_dot(self):
        dot = graphviz.Digraph(name=f'Syntax tree')
        dot.node("0", str(self), shape='rectangle', color='red')
        dot.node("1", self.value)
        dot.edge("0", "1", color='white')
        self._to_dot(dot, [1], "1")
        return dot

    def _to_dot(self, dot, n, cur_str):
        for child in self.children:
            n[0] += 1
            child_str = str(n[0])
            dot.node(
                child_str,
                child.value,
                color=('blue' if child.is_terminal() else 'black')
            )
            dot.edge(cur_str, child_str)
            child._to_dot(dot, n, child_str)
```

Синтаксический анализатор с использованием рекурсивного спуска:

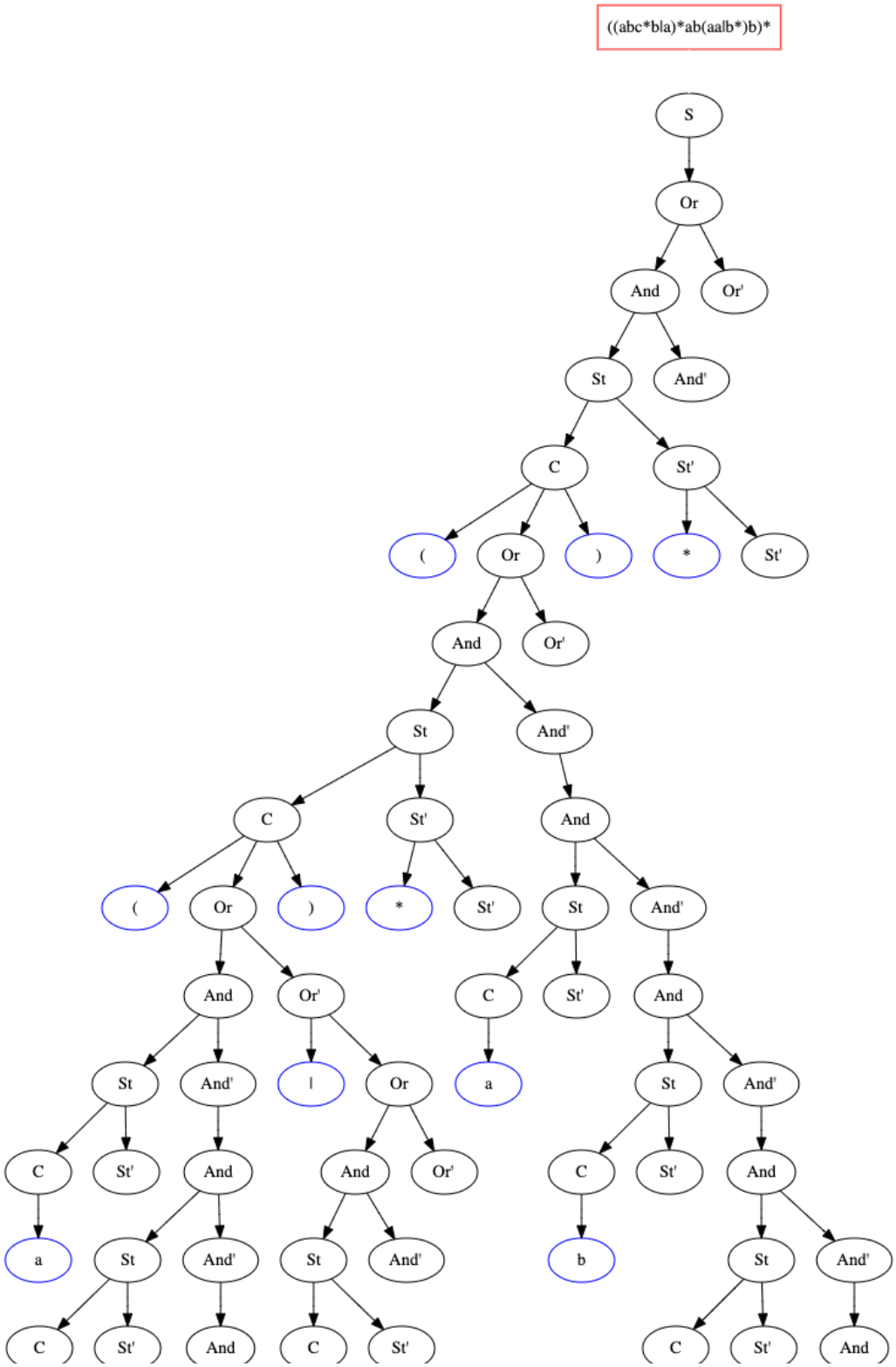
```
class Parser:
    def parse(self, input_str):
        self._lexical_analyzer = lexical_analyzer(input_str)
        self._next_token()
        return self._s()

    def _next_token(self):
        token, char, pos = next(self._lexical_analyzer)
        self._token = token
        self._char = char
```

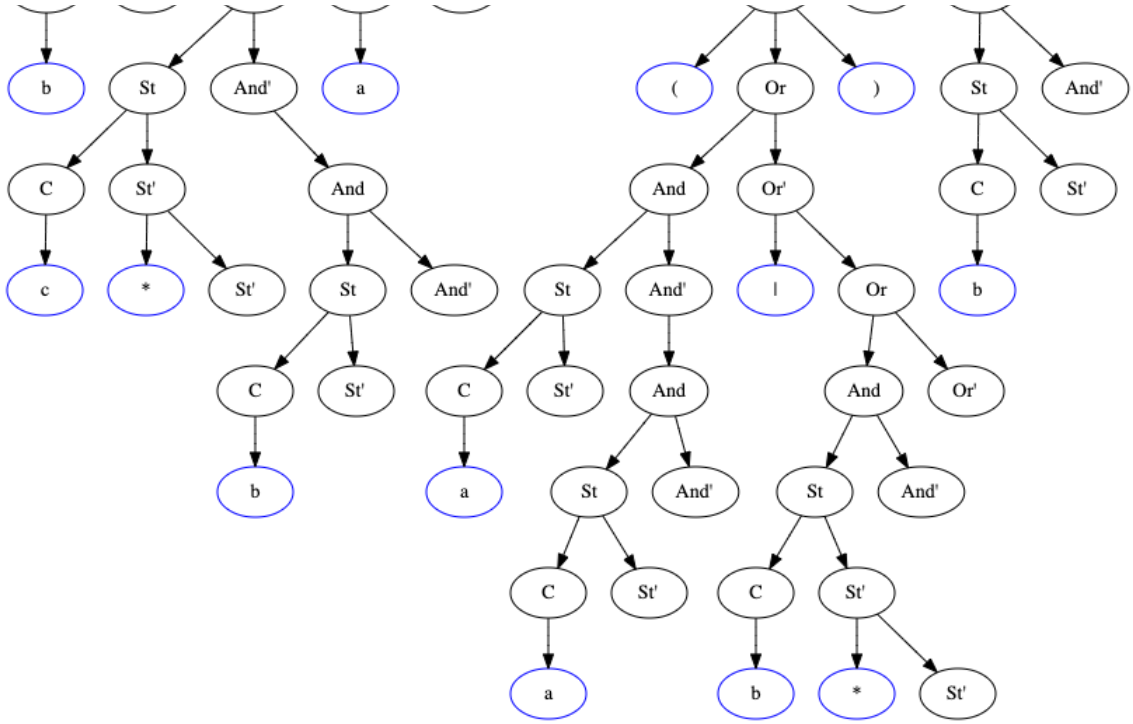
```
        self._pos = pos

    def _raise_exception(self, info=None, expected=None):
        raise ParseException(
            pos=self._pos,
            char=self._char,
            info=info,
            expected=expected
        )

    def _s(self):
        if self._token == Token.END:
            return Tree("S")
        return Tree("S", self._or())

    def _or(self):
        return Tree("Or", self._and(), self._or_prime())

    def _or_prime(self):
        if self._token == Token.OR:
            self._next_token()
            return Tree("Or'", Tree("|"), self._or())
        return Tree("Or'")

    def _and(self):
        return Tree("And", self._st(), self._and_prime())

    def _and_prime(self):
        if self._token in (Token.END, Token.RBRACKET, Token.OR):
            return Tree("And'")
        return Tree("And'", self._and())

    def _st(self):
        return Tree("St", self._c(), self._st_prime())

    def _st_prime(self):
        if self._token == Token.STAR:
            self._next_token()
            return Tree("St'", Tree("*"), self._st_prime())
        return Tree("St'")

    def _c(self):
        if self._token == Token.LBRACKET:
            self._next_token()
            subtree = self._or()
            if self._token != Token.RBRACKET:
                self._raise_exception(expected="Right bracket")
            self._next_token()
```

```
            return Tree("C", Tree("("), subtree, Tree(")"))
        elif self._token == Token.CHAR:
            res = Tree("C", Tree(self._char))
            self._next_token()
            return res
        self._raise_exception(info="Unknown symbol")
```

4. Визуализация дерева разбора

Построим дерево разбора для примера из условия задачи - $((abc*b|a)*ab(aa|b*)b)*$:





5. Подготовка набора тестов

```
class ParserTestCase(unittest.TestCase):
    def _parse_str(self, input_str):
        return str(self.parser.parse(input_str))

    def _assert_inputs(self, inputs):
        for input_str in inputs:
            self.assertEqual(input_str, self._parse_str(input_str))

    def setUp(self):
        self.parser = Parser()

    # Проверяем, что при корректном вводе
    # str класса tree совпадает с вводом
    def test_tree_str(self):
        inputs = [
            "((abc*b|a)*ab(aa|b*)b)*",
            "((a|a|acb))**acf|m",
            "((b|c)*a(b|c)*)((b|c)*a(b|c)*)((b|c)*a(b|c)*)",
            "ba(c)",
            "a|b*",
            "((b|c)*a**(b|c)*)a|b*"
        ]
        self._assert_inputs(inputs)

    def test_s(self):
        self.assertEqual(self._parse_str("    "), "")
        self.assertEqual(self._parse_str(""), "")
```

```
    def test_or(self):
        self._assert_inputs([
            "a**|ab|(fm*)*da",
            "abc",
            "a|b*c",
            "a|b|(md)",
            "a|b*",
            "ab",
            "a*b(c)"
        ])

    def test_and(self):
        self._assert_inputs(["a*b*", "(c)"])

    def test_st(self):
        self._assert_inputs(["ab(c)(a)", "a(b**)", "a*bd|c"])

    def test_c(self):
        self._assert_inputs(["c", "(c)", "(a|bc**(ds))"])

    def test_bad(self):
        bad_inputs = [
            "()",
            "(abc",
            "(ab41c|s",
            "*(ac",
            "(a||d)",
            ")()()",
        ]
        for bad_input in bad_inputs:
            with self.assertRaises(ParseException):
                self._parse_str(bad_input)
```