Лабораторная работа № 3.1
«Самоприменимый генератор компиляторов на основе предсказывающего анализа»

9 июня 2025 г.

Александр Старовойтов, ИУ9-61Б

Цель работы

Целью данной работы является изучение алгоритма построения таблиц предсказывающего анализатора.

Индивидуальный вариант

```
% аксиома
[axiom [E]]
% правила грамматики
[E    [T E']]
[E'   [+ T E'] []]
[T    [F T']]
[T'   [* F T'] []]
[F    [n] [( E )]]
```

Грамматика на входном языке

```
[axiom [GRAMMAR]]
[GRAMMAR [AXIOM RULES]]
[AXIOM [lb "axiom" lb NT rb rb]]
[RULES [RULE RULES] []]
[RULE [lb NT RHS rb]]
[NT [nonterm]]
[RHS [PRODUCTIONS RHSTAIL]]
[RHSTAIL [PRODUCTIONS RHSTAIL] []]
[PRODUCTIONS [lb PRODUCTIONSBODY rb]]
[PRODUCTIONSBODY [term PRODUCTIONSBODY] [nonterm PRODUCTIONSBODY] []]
```

Реализация

Генератор компиляторов

```
#!/usr/bin/env python3

from parser import Node, top_down_parse
import copy
import lexer
from table import TABLE

TEXT = """
[axiom [GRAMMAR]]
[GRAMMAR [AXIOM RULES]]
[AXIOM [lb "axiom" lb NT rb rb]]
[RULES [RULE RULES] []]
[RULE [lb NT RHS rb]]
[NT [nonterm]]
[RHS [PRODUCTIONS RHSTAIL]]
[RHSTAIL [PRODUCTIONS RHSTAIL] []]
[PRODUCTIONS [lb PRODUCTIONSBODY rb]]
[PRODUCTIONSBODY [term PRODUCTIONSBODY] [nonterm PRODUCTIONSBODY] []]
"""


# TABLE = {'Grammar': {'LB': ['Axiom', 'Rules']},
#         'Axiom': {'LB': ['LB', 'AXIOM', 'LB', 'Nt', 'RB', 'RB']},
#           'Rules': {'LB': ['Rule', 'Rules'], '$': []},
#           'Rule': {'LB': ['LB', 'Nt', 'Rhs', 'RB']},
#           'Nt': {'NONTERM': ['NONTERM']},
#           'Rhs': {'LB': ['Productions', 'Rhstail']},
#          'Rhstail': {'LB': ['Productions', 'Rhstail'], 'RB': []},
#          'Productions': {'LB': ['LB', 'Productionsbody', 'RB']},
#          'Productionsbody': {'TERM': ['TERM', 'Productionsbody'],
#                        'NONTERM': ['NONTERM', 'Productionsbody'],
#                               'RB': []}}

def get_child(node: Node, name: str) -> Node|None:
    return next(filter(lambda node: node.name == name, node.children), None)


def set_axiom(root: Node) -> None:
    nt = get_child(root, "Nt")
    assert nt is not None
    token = get_child(nt, "NONTERM")
    global AXIOM
    assert token is not None
    AXIOM = token.children[0].attr


def handle_rule(root: Node) -> None:
    lhs = get_child(root, "Nt")
    assert lhs is not None

    lhs = get_child(lhs, "NONTERM")
    assert lhs is not None

    lhs_token = lhs.children[0].name
    lhs = lhs.children[0].attr

    global RULES
    global RULES_TOKENS
    res_rhs = list()
    res_rhs_tokens = list()

    rhs = get_child(root, "Rhs")
    while rhs is not None and rhs.children[0].attr != "":
        productions = get_child(rhs, "Productions")
        assert productions is not None
        productions = get_child(productions, "Productionsbody")

        res_productions = list()
        res_tokens = list()

        while productions is not None and productions.children[0].attr != "":
            term = get_child(productions, "TERM")
            nonterm = get_child(productions, "NONTERM")
            if term is not None:
                res_productions.append(term.children[0].attr)
                res_tokens.append(term.children[0].name)
            if nonterm is not None:
                res_productions.append(nonterm.children[0].attr)
                res_tokens.append(nonterm.children[0].name)
            productions = get_child(productions, "Productionsbody")

        res_rhs.append(res_productions)
        res_rhs_tokens.append(res_tokens)

        rhs = get_child(rhs, "Rhstail")

    RULES_TOKENS.append((lhs_token, res_rhs_tokens))
    RULES.append((lhs, res_rhs))
```

```
ACTIONS = {
    "Axiom": set_axiom,
    "Rule": handle_rule,
}
AXIOM = None
RULES = list()
RULES_TOKENS = list()
FIRST = dict()


def dfs(tree: Node, level=0):
    if tree.name in ACTIONS:
        ACTIONS[tree.name](tree)
    print('  '*level+tree.name)
    for child in tree.children:
        dfs(child, level+1)


def is_nonterm(t):
    return t.isupper() or (t[:-1].isupper() and t[-1] == "'")


def first(rhs):
    if len(rhs) == 0:
        return {'ε'}
    if not is_nonterm(rhs[0]):
        return {rhs[0]}
    global FIRST
    f = FIRST[rhs[0]]
    if 'ε' not in f:
        return f
    return f.difference({'ε'}).union(first(rhs[1:]))


def calc_follow():
    follow = dict()
    for rule in RULES:
        lhs = rule[0]
        follow[lhs] = set()

    follow[AXIOM] = follow[AXIOM].union({'$'})

    for rule in RULES:
        lhs = rule[0]
        for rhs in rule[1]:
            for i, t in enumerate(rhs):
                if is_nonterm(t):
             follow[t] = follow[t].union(first(rhs[i+1:]).difference({'ε'}))

    tmp = None
    while tmp != follow:
        tmp = copy.deepcopy(follow)
        for rule in RULES:
            lhs = rule[0]
            for rhs in rule[1]:
                if len(rhs) == 0:
                    continue
                if is_nonterm(rhs[-1]):
               follow[rhs[-1]] = follow[rhs[-1]].union(follow[lhs])
                for i, t in enumerate(rhs[:-1]):
                    if is_nonterm(t) and 'ε' in first(rhs[i+1:]):
                        follow[t] = follow[t].union(follow[lhs])

    return follow


def calc_first():
    global FIRST

    for rule in RULES:
        lhs = rule[0]
        FIRST[lhs] = set()

    tmp = None
    while tmp != FIRST:
        tmp = copy.deepcopy(FIRST)
        for rule in RULES:
            lhs = rule[0]
            for rhs in rule[1]:
                FIRST[lhs] = FIRST[lhs].union(first(rhs))


def check_rules():
    lhs_parts = list()
    for rule in RULES:
        lhs = rule[0]
        lhs_parts.append(lhs)
    for i, rule in enumerate(RULES):
        lhs = rule[0]
        for j, rhs in enumerate(rule[1]):
            for k, el in enumerate(rhs):
                if is_nonterm(el) and el not in lhs_parts:
                    token = RULES_TOKENS[i][1][j][k]
             raise Exception(f'nonterminal {token} should be lhs of exactly one rule')


def gen_table(tree: Node):
    dfs(tree)
    print(AXIOM)
    print(RULES)
    table = dict()

    check_rules()

    calc_first()

    print(FIRST)
    follow = calc_follow()
    print(follow)

    for rule in RULES:
        lhs = rule[0]
        table[lhs] = dict()

    for i, rule in enumerate(RULES):
        lhs = rule[0]
        for rhs in rule[1]:
            f = first(rhs)
            for a in f:
                if a == 'ε':
                    continue
                if a in table[lhs]:
             raise Exception(f"grammar is not ll(1): {RULES_TOKENS[i][0]} is ambigious")
                table[lhs][a] = rhs
            if 'ε' in f:
                for b in follow[lhs]:
                    if b in table[lhs]:
                        raise Exception("grammar is not ll(1)")
                    table[lhs][b] = rhs
    return table


def main():
    tokens = lexer.tokenize(TEXT)
    derivation_tree = top_down_parse(tokens, 'Grammar',
                         ['$', 'LB', 'RB', 'AXIOM', 'NONTERM', 'TERM',
                              'LCB', 'RCB', 'NUM', 'STAR', 'PLUS'],
                                     TABLE)
    table = gen_table(derivation_tree)
    res_table = dict()
    for key, value in table.items():
        key = key[0]+key[1:].lower()
        res_table[key] = dict()
        for inner, inner_value in value.items():
            inner_array = copy.deepcopy(inner_value)
            inner_key = inner.upper()
            res = list()
            for el in inner_array:
                if el[0] == '"':
                    res.append(el[1:-1].upper())
                elif el.islower():
                    res.append(el.upper())
                else:
                    res.append(el[0]+el[1:].lower())
            res_table[key][inner_key] = res

    print()
    print(res_table)
    with open("table.py", "w") as f:
        print("TABLE =", res_table, file=f)


if __name__ == "__main__":
    main()
```

Калькулятор

```
#!/usr/bin/env python3

import lexer
from parser import *
from table import TABLE


TEXT = """
(2 + 3) * 4
"""


def dfs(root: Node):
    if root.attr is not None and root.attr in '()':
        return None
    if root.attr is not None and root.attr != '':
        return [root.attr]
    attrs = list()
    for child in root.children:
        tmp = dfs(child)
        if tmp is not None:
            for el in tmp:
                attrs.append(el)
    if len(attrs) == 3:
        return [calc(*attrs)]
    return attrs


def calc(lhs, op, rhs):
    if op == "+":
        return int(lhs)+int(rhs)
    elif op == "*":
        return int(lhs)*int(rhs)
    raise Exception("unreachable")


def calc_expr(derivation_tree: Node):
    return dfs(derivation_tree)[0]


def main():
    tokens = lexer.tokenize(TEXT)
    derivation_tree = top_down_parse(tokens,
   'E', ['$', 'LB', 'RB', 'AXIOM', 'NONTERM', 'TERM', 'LCB', 'RCB', 'NUM', 'STAR', 'PLUS'], TABLE
    # print(get_dot(derivation_tree))
    # print()
    # print()
    print(calc_expr(derivation_tree))


if __name__ == "__main__":
    main()
```

Тестирование

Генератор компиляторов

Таблица для калькулятора

```
TABLE = {'E': {'LCB': ['T', "E'"], 'NUM': ['T', "E'"]}, "E'": {'PLUS': ['PLUS', 'T', "E'"], '$':
```

```
'T': {'LCB': ['F', "T'"], 'NUM': ['F', "T'"]}, "T'": {'STAR': ['STAR', 'F', "T'"], 'PLUS': [], '$
'RCB': []}, 'F': {'NUM': ['NUM'], 'LCB': ['LCB', 'E', 'RCB']}}
```

Таблица для собственной грамматики

```
 TABLE = {'Grammar': {'LB': ['Axiom', 'Rules']},
           'Axiom': {'LB': ['LB', 'AXIOM', 'LB', 'Nt', 'RB', 'RB']},
           'Rules': {'LB': ['Rule', 'Rules'], '$': []},
           'Rule': {'LB': ['LB', 'Nt', 'Rhs', 'RB']},
           'Nt': {'NONTERM': ['NONTERM']},
           'Rhs': {'LB': ['Productions', 'Rhstail']},
           'Rhstail': {'LB': ['Productions', 'Rhstail'], 'RB': []},
           'Productions': {'LB': ['LB', 'Productionsbody', 'RB']},
           'Productionsbody': {'TERM': ['TERM', 'Productionsbody'],
                               'NONTERM': ['NONTERM', 'Productionsbody'],
                               'RB': []}}
```

Калькулятор

```
#!/usr/bin/env python3

import lexer
from parser import *
from table import TABLE


TEXT = """
(2 + 3) * 4
"""


def dfs(root: Node):
    if root.attr is not None and root.attr in '()':
        return None
    if root.attr is not None and root.attr != '':
        return [root.attr]
    attrs = list()
    for child in root.children:
        tmp = dfs(child)
        if tmp is not None:
            for el in tmp:
                attrs.append(el)
    if len(attrs) == 3:
        return [calc(*attrs)]
    return attrs
```

```
def calc(lhs, op, rhs):
    if op == "+":
        return int(lhs)+int(rhs)
    elif op == "*":
        return int(lhs)*int(rhs)
    raise Exception("unreachable")


def calc_expr(derivation_tree: Node):
    return dfs(derivation_tree)[0]


def main():
    tokens = lexer.tokenize(TEXT)
    derivation_tree = top_down_parse(tokens, 'E',
        ['$', 'LB', 'RB', 'AXIOM', 'NONTERM', 'TERM', 'LCB', 'RCB', 'NUM', 'STAR', 'PLUS'],
    # print(get_dot(derivation_tree))
    # print()
    # print()
    print(calc_expr(derivation_tree))


if __name__ == "__main__":
    main()
```

Вывод

Изучил алгоритм построения таблиц предсказывающего анализатора.