«Теория автоматов и формальных языков»

тема: Создание синтаксического анализатора

Санкт-Петербург, 2025

**Задание**

1. Cоставить правила грамматики, описывающей синтаксис цикла с условием (с предусловием или постусловием – на ваш выбор).

2. Написать программу, которая будет получать на вход строку, преобразовывать ее в последовательность лексем (функционал лексера), проводить попытку синтаксического анализа и сообщать, соответствует ли строка грамматике цикла с условием, описанной в предыдущем пункте, или нет. Тип парсера LL или LR.

**Ход выполнения работы**

Создадим упрощенную грамматику для цикла for в Python:

FOR\_LOOP → for IDENTIFIER in ITERABLE: ACTION [else: ACTION]

ITERABLE → IDENTIFIER | RANGE | LIST | TUPLE | STRING

RANGE → range(ARGS)

ARGS → INT | INT, INT | INT, INT, INT

LIST → \[[ELEMENTS]\]

TUPLE → ([ELEMENTS])

ELEMENTS → ELEMENT | ELEMENT, ELEMENTS

ELEMENT → IDENTIFIER | INT | FLOAT | STRING

ACTION → STATEMENTS

STATEMENTS → STATEMENT | STATEMENT SEP STATEMENTS

STATEMENT → any valid Python statement

SEP → \n

ITERABLE: Для цикла for мы устанавливаем не условие в чистом виде, а некий массив данных: список, кортеж, строку, словарь, диапазон или любой другой итерируемый объект.

Для упрощения реализации будем считать, что тело цикла (ACTION) – это любой набор токенов после **:**, пока не встретится токен с меньшим отступом, т.к. именно отступы \t являются в Python главным маркером принадлежности действий <ACTION> → <FUNCTION> | <EXP> к телу цикла (но в нашей реализации мы просто проверим наличие **:** и что после него есть хотя бы один токен).

Теперь, когда примерная грамматика готова, составим вспомогательную таблицу разбора для LL(1)-парсера.

**Таблица для LL-парсера.**

⎯ Нетерминал на вершине стека

⎯ Текущий терминал во входной последовательности токенов

⎯ Входная последовательность не соответствует грамматике (ошибка во входной последовательности)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | for | ID | in | : | else | range | ( | ) | [ | ] | , | INT | FLOAT | STR | \n | STATE | EOF |
| FOR\_LOOP | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ITERABLE |  | 2 |  |  |  | 3 |  |  | 4 |  |  |  |  | 5 |  |  |  |
| RANGE |  |  |  |  |  | 6 |  |  |  |  |  |  |  |  |  |  |  |
| ARGS |  |  |  |  |  |  |  |  |  |  |  | 7 |  |  |  |  |  |
| LIST |  |  |  |  |  |  |  |  | 8 |  |  |  |  |  |  |  |  |
| TUPLE |  |  |  |  |  |  | 9 |  |  |  |  |  |  |  |  |  |  |
| ELEMENTS |  | 10 |  |  |  |  |  |  |  |  |  | 10 | 10 | 10 |  |  |  |
| ELEMENT |  | 11 |  |  |  |  |  |  |  |  |  | 12 | 13 | 14 |  |  |  |
| ACTION |  | 15 |  |  |  |  |  |  |  |  |  |  |  |  |  | 15 |  |
| STATEMENTS |  | 16 |  |  |  |  |  |  |  |  |  |  |  |  |  | 16 |  |
| STATEMENT |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 17 |  |
| SEP |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 18 |  |  |

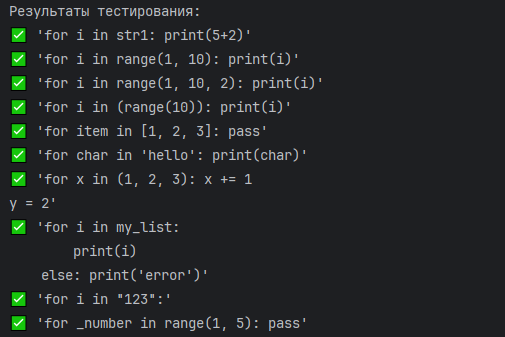
Нетерминалы: FOR\_LOOP, ITERABLE, RANGE, ARGS, LIST, TUPLE, ELEMENTS, ELEMENT, ACTION, STATEMENTS, STATEMENT, SEP

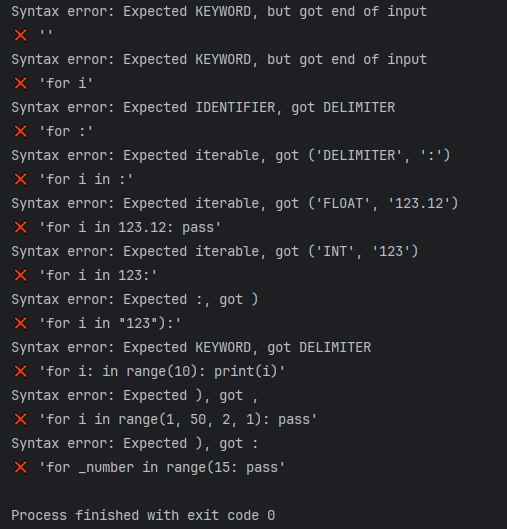
Терминалы: for, IDENTIFIER, in, :, else, range, (, ), [, ], ,, INT, FLOAT, STRING, \n, STATEMENT, EOF

**Правила**

1. FOR\_LOOP → for IDENTIFIER in ITERABLE: ACTION [else: ACTION]
2. ITERABLE → IDENTIFIER
3. ITERABLE → RANGE
4. ITERABLE → LIST
5. ITERABLE → STRING
6. RANGE → range(ARGS)
7. ARGS → INT | INT, INT | INT, INT, INT
8. LIST → \[[ELEMENTS]\]
9. TUPLE → ([ELEMENTS])
10. ELEMENTS → ELEMENT | ELEMENT, ELEMENTS
11. ELEMENT → IDENTIFIER
12. ELEMENT → INT
13. ELEMENT → FLOAT
14. ELEMENT → STRING
15. ACTION → STATEMENTS
16. STATEMENTS → STATEMENT | STATEMENT SEP STATEMENTS
17. STATEMENT → (any valid Python statement)
18. SEP → \n

**Результаты работы программы**





**Вывод**

В ходе выполнения работы был реализован LL(1)-парсер для синтаксического анализа циклов for в языке Python, проверяющий соответствие кода заданной грамматике.

Грамматика включает правила для:

* Цикла for с итерируемыми объектами (range, списки, кортежи, строки).
* Тела цикла (ACTION) с поддержкой простых операторов.
* Опционального блока else.

Программа корректно распознает синтаксически верные циклы for и выявляет ошибки в некорректных конструкциях.

Проведенные тесты подтверждают, что корректные примеры обрабатываются без ошибок, а ошибочные входные данные отклоняются с соответствующими сообщениями.

**Листинг программы lab6**

|  |
| --- |
| import re # Импорт модуля для работы с регулярными выражениями  # Определение токенов языка Python, которые мы будем распознавать # Каждый токен имеет имя и регулярное выражение для его распознавания TOKENS = [  ('SPACE', r'[\s]+'), # Пробельные символы (пропускаем)  ('COMMENT', r'\#.\*'),  ('KEYWORD', r'\b(for|in|else)\b'), # Убрали range из ключевых слов  ('RANGE', r'\brange\b'), # Добавили отдельный тип для range  ('DELIMITER', r'\(|\)|\[|\]|\,|\:|\n'),  ('OPERATOR', r'=|\+|-|\\*|\/|\\*\\*|\/\/|%'),  ('COMPARISON\_OPERATOR', r'<=|>=|==|!=|>|<'),  ('LOGICAL\_OPERATOR', r'&|\||\^'),  ('FLOAT', r'-?\d+\.\d+'),  ('INT', r'-?\d+\b'),  ('STRING', r'''('[^']\*'|"[^"]\*")'''),  ('IDENTIFIER', r'\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b'), ]   class Parser:  def \_\_init\_\_(self, tokens): # Инициализация парсера с токенами  self.tokens = [t for t in tokens if t[0] not in ['SPACE', 'COMMENT']]  self.pos = 0 # Текущая позиция в списке токенов  self.current\_token = self.tokens[self.pos] if self.tokens else None   # Переход к следующему токену  def advance(self):  self.pos += 1  if self.pos < len(self.tokens):  self.current\_token = self.tokens[self.pos]  else:  self.current\_token = None   # Основной метод парсинга  def parse(self):  try:  self.parse\_for\_loop() # Парсим цикл for  if self.current\_token is not None: # Проверяем, что разобрали все токены  raise SyntaxError(f"Unexpected token {self.current\_token}")  return True  except SyntaxError as e:  print(f"Syntax error: {e}")  return False   # Парсинг конструкции for: for <id> in <iterable>: <action> [else: <action>]  def parse\_for\_loop(self):  self.match('KEYWORD', 'for') # Проверяем ключевое слово for  self.match('IDENTIFIER') # Проверяем идентификатор переменной  self.match('KEYWORD', 'in') # Проверяем ключевое слово in  self.parse\_iterable() # Парсим итерируемый объект  self.match('DELIMITER', ':') # Проверяем двоеточие  self.parse\_action() # Парсим тело цикла  # Проверяем необязательную часть else  if self.current\_token and self.current\_token[1] == 'else':  self.match('KEYWORD', 'else')  self.match('DELIMITER', ':')  self.parse\_action()   # Парсинг итерируемого объекта ITERABLE после in  def parse\_iterable(self):  if not self.current\_token:  raise SyntaxError("Unexpected end of input")   if self.current\_token[0] == 'IDENTIFIER':  self.advance() # Простой идентификатор (переменная)  elif self.current\_token[0] == 'RANGE':  self.parse\_range() # Вызов функции range()  elif self.current\_token[1] == '[':  self.parse\_list() # Список в квадратных скобках  elif self.current\_token[1] == '(':  self.parse\_tuple\_or\_range() # Кортеж или range в скобках  elif self.current\_token[0] == 'STRING':  self.advance() # Строковый литерал  else:  raise SyntaxError(f"Expected iterable, got {self.current\_token}")   # Обработка неоднозначности между кортежем и range в скобках  def parse\_tuple\_or\_range(self):  start\_pos = self.pos # Запоминаем текущую позицию для возможного отката  try:  self.match('DELIMITER', '(') # Открывающая скобка  if self.current\_token and self.current\_token[0] == 'RANGE':  self.parse\_range() # Если после скобки идет range - это вызов range в скобках  if self.current\_token and self.current\_token[1] == ')':  self.match('DELIMITER', ')')  else: # Иначе это кортеж  if self.current\_token and self.current\_token[1] != ')':  self.parse\_elements() # Элементы кортежа  self.match('DELIMITER', ')') # Закрывающая скобка  except SyntaxError: # Если возникла ошибка - откатываемся и пробуем разобрать как кортеж  self.pos = start\_pos  self.current\_token = self.tokens[self.pos]  self.parse\_tuple()   # Парсинг вызова range: range(<args>)  def parse\_range(self):  if self.current\_token[0] == 'RANGE':  self.advance() # Пропускаем 'range'  self.match('DELIMITER', '(') # Открывающая скобка  self.parse\_range\_args() # Аргументы range  self.match('DELIMITER', ')') # Закрывающая скобка   # Парсинг аргументов range: 1, 2 или 3 целых числа  def parse\_range\_args(self):  self.match('INT') # Первый аргумент (обязательный)  if self.current\_token and self.current\_token[1] == ',':  self.match('DELIMITER', ',') # Запятая  self.match('INT') # Второй аргумент  if self.current\_token and self.current\_token[1] == ',':  self.match('DELIMITER', ',') # Запятая  self.match('INT') # Третий аргумент   # Парсинг списка: [<elements>]  def parse\_list(self):  self.match('DELIMITER', '[') # Открывающая квадратная скобка  if self.current\_token and self.current\_token[1] != ']':  self.parse\_elements() # Элементы списка (если не пустой)  self.match('DELIMITER', ']') # Закрывающая квадратная скобка   # Парсинг кортежа: (<elements>)  def parse\_tuple(self):  self.match('DELIMITER', '(')  if self.current\_token and self.current\_token[1] != ')':  self.parse\_elements()  self.match('DELIMITER', ')')   # Парсинг элементов списка/кортежа: элемент или несколько через запятую  def parse\_elements(self):  self.parse\_element() # Первый элемент  while self.current\_token and self.current\_token[1] == ',':  self.match('DELIMITER', ',') # Запятая  self.parse\_element() # Следующий элемент   # Парсинг одного элемента (идентификатор, число, строка)  def parse\_element(self):  if not self.current\_token:  raise SyntaxError("Unexpected end of input")   if self.current\_token[0] in ('IDENTIFIER', 'INT', 'FLOAT', 'STRING'):  self.advance()  else:  raise SyntaxError(f"Expected element, got {self.current\_token}")   def parse\_action(self):  self.parse\_statements() # Парсинг тела цикла (после двоеточия)   # Парсинг одного или нескольких операторов  def parse\_statements(self):  self.parse\_statement() # Первый оператор  while (self.current\_token and  (self.current\_token[0] == 'DELIMITER' and self.current\_token[1] == '\n' or  self.is\_statement\_start())):  if self.current\_token[1] == '\n':  self.match('DELIMITER', '\n') # Разделитель операторов  if self.is\_statement\_start():  self.parse\_statement() # Следующий оператор   # Упрощенный парсинг оператора Python (пропускаем все до ключевых слов)  def parse\_statement(self):  if not self.current\_token:  return   while (self.current\_token and  not (self.current\_token[0] == 'KEYWORD' and  self.current\_token[1] in ('for', 'if', 'else', 'while'))):  self.advance()   # проверка, может ли токен начинать оператор  def is\_statement\_start(self):  if not self.current\_token:  return False  return (self.current\_token[0] in ('IDENTIFIER', 'INT', 'FLOAT', 'STRING', 'DELIMITER') or  (self.current\_token[0] == 'KEYWORD' and  self.current\_token[1] not in ('for', 'if', 'else', 'while')))   # Проверка соответствия текущего токена ожидаемому  def match(self, token\_type, value=None):  if not self.current\_token:  raise SyntaxError(f"Expected {token\_type}, but got end of input")  if self.current\_token[0] != token\_type:  raise SyntaxError(f"Expected {token\_type}, got {self.current\_token[0]}")  if value is not None and self.current\_token[1] != value:  raise SyntaxError(f"Expected {value}, got {self.current\_token[1]}")  self.advance()  # Лексический анализатор - разбивает строку на токены def tokenize(input\_code):  tokens = []  while input\_code:  for token\_type, pattern in TOKENS:  regex = re.compile(pattern)  match = regex.match(input\_code)  if match:  value = match.group(0)  if token\_type not in ['SPACE', 'COMMENT']:  tokens.append((token\_type, value))  input\_code = input\_code[len(value):]  break  else: # Если токен не распознан - пропускаем один символ  input\_code = input\_code[1:]  return tokens  # Основная функция проверки валидности цикла for def is\_valid\_for\_loop(input\_str):  try:  token\_list = tokenize(input\_str) # Лексический анализ  parser = Parser(token\_list) # Создание парсера  return parser.parse() # Запуск парсинга  except Exception as e:  print(f"Error: {e}")  return False   # примеры для проверки работы парсера test\_examples = [  ("for i in str1: print(5+2)", True),  ("for i in range(1, 10): print(i)", True),  ("for i in range(1, 10, 2): print(i)", True),  ("for i in (range(10)): print(i)", True),  ("for item in [1, 2, 3]: pass", True),  ("for char in 'hello': print(char)", True),  ("for x in (1, 2, 3): x += 1\ny = 2", True),  ("for i in my\_list:\n\t\tprint(i)\n\telse: print('error')", True),  ("for i in \"123\":", True),  ("for \_number in range(1, 5): pass", True),   ("", False),  ("for i", False),  ("for :", False),  ("for i in :", False),  ("for i in 123.12: pass", False),  ("for i in 123:", False),  ("for i in \"123\"):", False),  ("for i: in range(10): print(i)", False),  ("for i in range(1, 50, 2, 1): pass", False),  ("for \_number in range(15: pass", False), ]  print("Результаты тестирования:") for example, expected in test\_examples:  result = is\_valid\_for\_loop(example)  status = '✅' if (result == True and result == expected) else '❌'  print(f"{status} '{example}'") |