Министерство образования и науки российской федерации

(минобрнауки россии)

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Санкт-Петербургский государственный политехнический университет» (ФГБОУ ВПО «СПбГПУ»)

**Институт менеджмента и информационных технологий**

(филиал)федерального государственного бюджетного образовательного учреждения высшего профессионального образования

«Санкт-Петербургский государственный политехнический университет» в г. Череповце (ИМИТ «СПбГПУ»)

Кафедра ПО ВТ и АС

ЛАБОРАТОРНАЯ РАБОТА №3

Дисциплина: «Теория языков программирования и методы трансляции»

Тема: «Синтаксический анализ. Нисходящий рекурсивный анализатор без возвратов»

Выполнил студент группы о.291 Шанин Игнат Леонидович

№ зачетной книжки о2080127

Проверил Михайлов Андрей Евгеньевич

«\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_201\_\_ г.

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

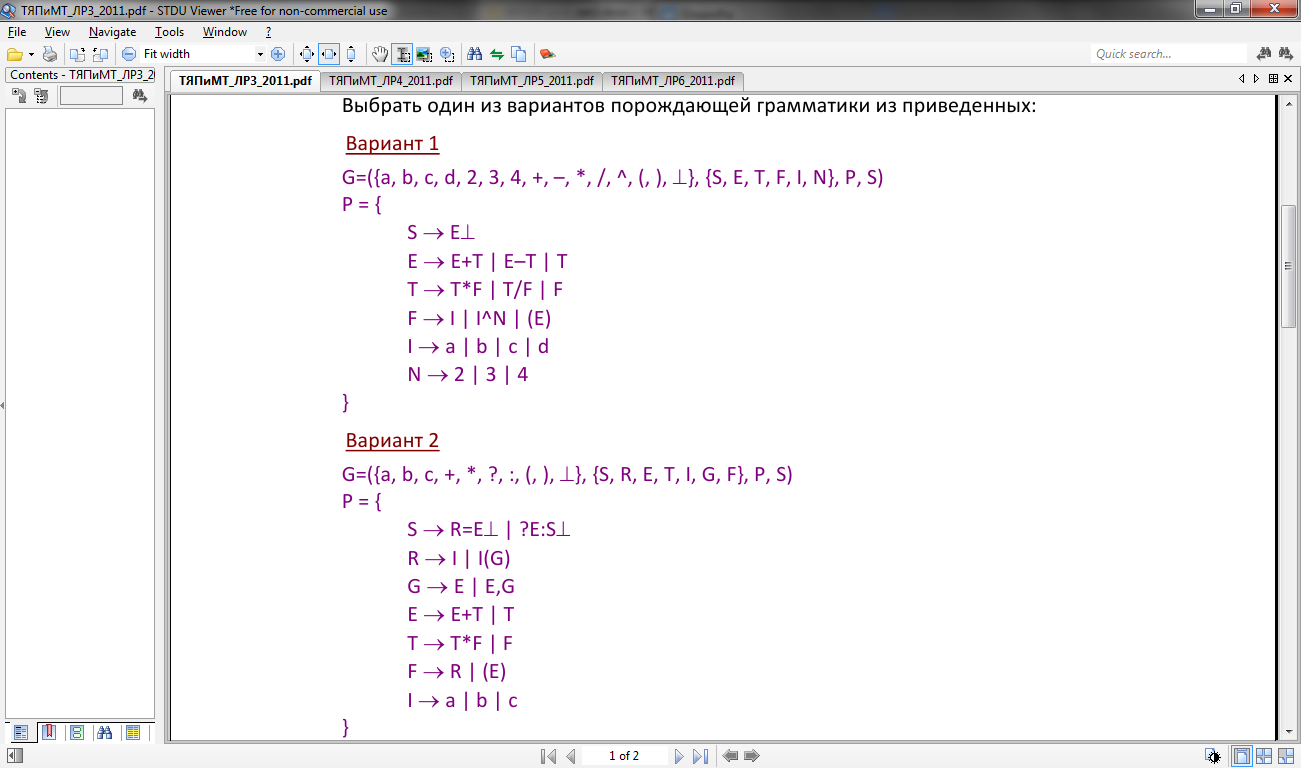
отметка о зачете подпись преподавателя

г. Череповец

2012

# Задание

# На основе порождающей грамматики реализовать синтаксический анализатор, работа которого основана на методе рекурсивного спуска без возвратов. Для реализации можно использовать любой ЯП. Программа должна принимать на вход текстовый файл со строками для анализа и генерировать на выходе файл, только с теми строками, которые были корректно распознаны анализатором.



# Краткая теоретическая справка

На этапе синтаксического анализа нужно установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и зафиксировать эту структуру. Следовательно, надо решать задачу разбора: дана цепочка лексем, и надо определить, выводима ли она в грамматике, определяющей синтаксис языка. Однако структура таких конструкций как выражение, описание, оператор и т.п., более сложная, чем структура идентификаторов и чисел. Поэтому для описания синтаксиса языков программирования нужны более мощные грамматики, чем регулярные. Обычно для этого используют укорачивающие контекстно-свободные грамматики (УКС-грамматики), правила которых имеют вид , где ,. Грамматики этого класса, с одной стороны, позволяют достаточно полно описать синтаксическую структуру реальных языков программирования; с другой стороны, для разных подклассов УКС-грамматик существуют достаточно эффективные алгоритмы разбора.

С теоретической точки зрения существует алгоритм, который по любой данной КС-грамматике и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой грамматикой.

Метод рекурсивного спуска (РС-метод). Для каждого нетерминала грамматики создается своя процедура, носящая его имя; ее задача – начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что цепочка не принадлежит языку, и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

# Решение

Перед тем, как применять метод рекурсивного спуска, необходимо устранить в грамматике левую рекурсию и левофакторизовать её:

## Вариант 1

* S → E┴
* E → E+T | E–T | T
* T → T\*F | T/F | F
* F → I | I^N | (E)
* I → a | b | c | d
* N → 2 | 3 | 4

Устранение левой рекурсии:

* E → E+T | E–T | T
  + E → ТE’
  + E’→ +ТЕ’| -ТЕ’|λ
* T → T\*F | T/F | F
  + Т → FТ’
  + T’→ \*FT’| /FT’|λ

Левая факторизация:

* F → I | I^N | (E)
  + F → IF’ | (E)
  + F’→ ^N | λ

Результат:

* S → E┴
* E → ТE’
* E’→ +ТЕ’| -ТЕ’|λ
* Т → FТ’
* T’→ \*FT’| /FT’|λ
* F → I F’ | (E)
* F’→ ^N | λ
* I → a | b | c | d
* N → 2 | 3 | 4

## Вариант 2

* S → R=E^ | ?E:S^
* R → I | I(G)
* G → E | E,G
* E → E+T | T
* T → T\*F | F
* F → R | (E)
* I → a | b | c

Устранение левой рекурсии:

* E → E+T | T
  + E → ТE’
  + E’→ +ТЕ’|λ
* T → T\*F | F
  + Т → FТ’
  + T’→ \*FT’|λ

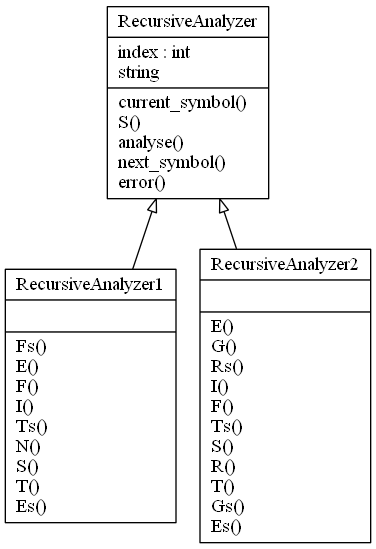
Левая факторизация:

* R → I | I(G)
  + R → IR’
  + R’→ (G)|λ
* G → E | E,G
  + G → EG’
  + G’→ ,G|λ

Результат:

* S → R=E^ | ?E:S^
* R → IR’
* R’→ (G)|λ
* G → EG’
* G’→ ,G|λ
* E → ТE’
* E’→ +ТЕ’|λ
* Т → FТ’
* T’→ \*FT’|λ
* F → R | (E)
* I → a | b | c

# Листинг



*#!/usr/bin/python*

*# -\*- coding: utf-8 -\*-*

*# Рекурсивный анализатор*

**class** **RecursiveAnalyzer**:

**def** **analyse**(self, string):

        self**.**string **=** string

        self**.**index **=** 0

**try**:

            self**.**S()

**except** **ValueError**:

**return** False

**return** self**.**index **==** len(string)

**def** **error**(self):

**raise** **ValueError**()

**def** **next\_symbol**(self):

        self**.**index **+=** 1

    @property

**def** **current\_symbol**(self):

**try**:

**return** self**.**string[self**.**index]

**except** **IndexError**:

**return** None

**def** **S**(self):

**raise** NotImplemented()

*# для первого варианта*

**class** **RecursiveAnalyzer1**(RecursiveAnalyzer):

**def** **S**(self):

        self**.**E()

**def** **E**(self):

        self**.**T()

        self**.**Es()

**def** **Es**(self):

**if** self**.**current\_symbol **in** ('+', '-'):

            self**.**next\_symbol()

            self**.**T()

            self**.**Es()

**def** **T**(self):

        self**.**F()

        self**.**Ts()

**def** **Ts**(self):

**if** self**.**current\_symbol **in** ('\*', '/'):

            self**.**next\_symbol()

            self**.**F()

            self**.**Ts()

**def** **F**(self):

**if** self**.**current\_symbol **is** '(':

            self**.**next\_symbol()

            self**.**E()

**if** self**.**current\_symbol **is** ')':

                self**.**next\_symbol()

**else**:

                self**.**error()

**else**:

            self**.**I()

            self**.**Fs()

**def** **Fs**(self):

**if** self**.**current\_symbol **is** '^':

            self**.**next\_symbol()

            self**.**N()

**def** **I**(self):

**if** self**.**current\_symbol **in** ('a', 'b', 'c', 'd'):

            self**.**next\_symbol()

**else**:

            self**.**error()

**def** **N**(self):

**if** self**.**current\_symbol **in** ('2', '3', '4'):

            self**.**next\_symbol()

**else**:

            self**.**error()

*# для второго варианта*

**class** **RecursiveAnalyzer2**(RecursiveAnalyzer):

**def** **S**(self):

**if** self**.**current\_symbol **is** '?':

            self**.**next\_symbol()

            self**.**E()

**if** self**.**current\_symbol **is** ':':

                self**.**next\_symbol()

                self**.**S()

**if** self**.**current\_symbol **is** '^':

                    self**.**next\_symbol()

**else**:

                    self**.**error()

**else**:

                self**.**error()

**else**:

            self**.**R()

**if** self**.**current\_symbol **is** '=':

                self**.**next\_symbol()

                self**.**E()

**if** self**.**current\_symbol **is** '^':

                    self**.**next\_symbol()

**else**:

                    self**.**error()

**else**:

                self**.**error()

**def** **R**(self):

        self**.**I()

        self**.**Rs()

**def** **Rs**(self):

**if** self**.**current\_symbol **is** '(':

            self**.**next\_symbol()

            self**.**G()

**if** self**.**current\_symbol **is** ')':

                self**.**next\_symbol()

**else**:

                self**.**error()

**def** **G**(self):

        self**.**E()

        self**.**Gs()

**def** **Gs**(self):

**if** self**.**current\_symbol **is** ',':

            self**.**next\_symbol()

            self**.**G()

**def** **E**(self):

        self**.**T()

        self**.**Es()

**def** **Es**(self):

**if** self**.**current\_symbol **is** '+':

            self**.**next\_symbol()

            self**.**T()

            self**.**Es()

**def** **T**(self):

        self**.**F()

        self**.**Ts()

**def** **Ts**(self):

**if** self**.**current\_symbol **is** '\*':

            self**.**next\_symbol()

            self**.**F()

            self**.**Ts()

**def** **F**(self):

**if** self**.**current\_symbol **is** '(':

            self**.**next\_symbol()

            self**.**E()

**if** self**.**current\_symbol **is** ')':

                self**.**next\_symbol()

**else**:

                self**.**error()

**else**:

            self**.**R()

**def** **I**(self):

**if** self**.**current\_symbol **in** ('a', 'b', 'c'):

            self**.**next\_symbol()

**else**:

            self**.**error()

**if** \_\_name\_\_ **==** '\_\_main\_\_':

**with** open("test.txt") **as** f:

        chains **=** f**.**read()**.**replace(" ", "")**.**split('\n')

    rc1 **=** RecursiveAnalyzer1()

    rc2 **=** RecursiveAnalyzer2()

    chains1 **=** filter(rc1**.**analyse, chains)

    chains2 **=** filter(rc2**.**analyse, chains)

    other **=** filter(**lambda** x: **not** (rc1**.**analyse(x) **or** rc2**.**analyse(x)), chains)

**with** open("1.txt", "w") **as** f:

        f**.**write('\n'**.**join(chains1))

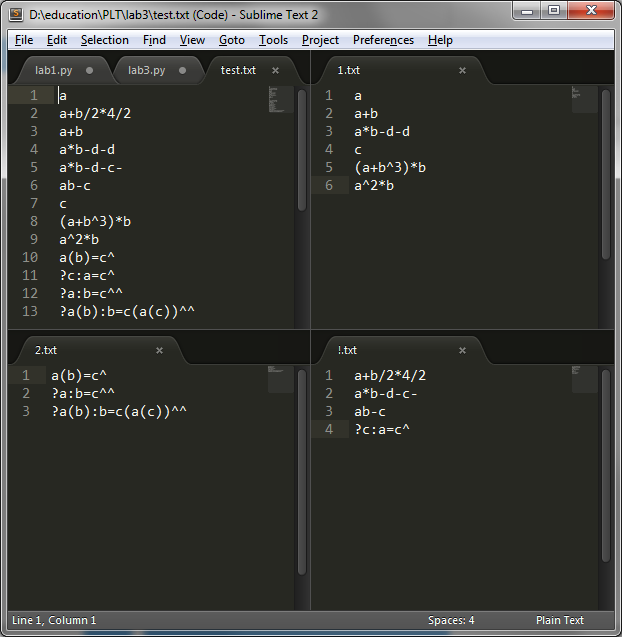
**with** open("2.txt", "w") **as** f:

        f**.**write('\n'**.**join(chains2))

**with** open("!.txt", "w") **as** f:

        f**.**write('\n'**.**join(other))

# Результат работы программы



# Вывод

В ходе выполнения лабораторной работы был реализован синтаксический анализатор, работа которого основана на методе рекурсивного спуска без возвратов. Перед построением анализатора грамматики были приведены к более удобному виду путем удаления левой рекурсии и левофакторизацией.