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 №6

Дисциплина: «Теория языков программирования и методы трансляции»

Тема: «Синтаксически управляемая трансляция»

|  |  |  |
| --- | --- | --- |
| Выполнил студент группы о.291: | Шанин Игнат Леонидович | |
|  | № зачетной книжки: о2080127 | |
| Проверил: | Михайлов Андрей Евгеньевич | |
|  | «\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | отметка о зачете | подпись преподавателя |

# Задание

Написать программу подсчета значения арифметического выражения с учетом приоритета операций, состоящего из целых неотрицательных чисел, скобок и знаков ‘+’, ‘–’, ‘\*’, ‘/’.

1. Используя приведенную в таблице схему трансляции написать программу перевода арифметического выражения из инфиксной формы в постфиксную.



2. Добавить в программу возможность вычисления полученного выражения в постфиксной форме с использованием стека.

# Краткая теория

Синтаксически управляемая трансляция (СУТ, англ. Syntax-directed translation, SDT) — преобразование текста в последовательность команд, через добавление таких команд в правила грамматики. Во время обработки строки синтаксический анализатор находит последовательность применений правил. СУТ предоставляет простой способ связи такого синтаксиса с семантикой.

Синтаксически управляемая трансляция работает за счет добавления действий в контекстно-свободную грамматику. Эти действия будут осуществляться, когда соответствующее правило используется в выводе. Описание грамматики с такими действиями называется схемой синтаксически управляемой трансляции (или просто схемой трансляции).

Каждый символ в грамматике может иметь атрибуты, которые содержат данные. Обычно такие атрибуты могут включать в себя тип переменной, значение выражения, и т.п. Для символа X с атрибутом t обращение к атрибуту может выглядеть как X.t.

Таким образом, используя действия и атрибуты, грамматика может быть применена для перевода текста с языка, порождаемого ею, выполнением действий и переносом информации через атрибуты символов.

# Листинг программы

*#!/usr/bin/python*

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

"""Syntax-directed translation"""

**class** **LL1PDAWithSDT**(object):

    """Pushdown automaton for LL(1) grammar with syntax-directed translation

support.

LL1PDAWithSDT(grammar) -> object

\* grammar is dictionary represented formal context-free grammar like

{

'T': {'a', 'b', 'c'},

'N': {'A', 'B', 'C'},

'P': {

'A -> a A {print "Hello world"} | B',

'B -> b B | C',

'C -> c C | ',

},

'S': 'A'

}

Where

\* T - set of terminal symbols

\* N - set of nonterminal symbols

\* P - set of production rules

\* S - start symbol

Each rule is a string with nonterminal symbol, symol '->' and one ot more

rules separated by '|' composed of terminals, nonterminals and attributes

(in our case python code) in curly brackets separated by spaces.

In order for whole thing to work all rules must have no left recursion.

Symbols '|', '{', '}' and '$' are forbidden to use as terminals and

nonterminals.

"""

**def** **\_\_init\_\_**(self, grammar):

        self**.**\_\_dict\_\_**.**update(grammar) *# Adds T N P S attributes to the object*

*# Let's convert each rule from P to something we can work with*

*# string like 'E`-> + T {print "+",} E` | ' will be converted to*

*# 'E`': [['+', 'T', '{print "+",}', 'E`'], []] (set becomes dictionary)*

        converted\_rules **=** dict()

**for** production **in** self**.**P:

**try**:

                nonterm, rules **=** production**.**split('->', 1)

                nonterm **=** nonterm**.**replace(' ', '')

*# Splitting right part to separate rules*

                converted\_rules[nonterm] **=** []

**for** rule **in** rules**.**split('|'):

*# and each rule to symbols and code*

                    converted\_rule **=** list()

                    opened\_curlies **=** 0

**for** element **in** rule**.**split():

**if** opened\_curlies **>** 0: *# than it is continuation of code*

*# that has spaces*

                            converted\_rule[**-**1] **+=** ' ' **+** element

**elif** opened\_curlies **==** 0 : *# than it is new element*

                            converted\_rule**.**append(element)

**else**:

**raise** **ValueError**("'{' missing")

                        opened\_curlies **+=** 1 **if** element[0] **==** '{' **else** 0

                        opened\_curlies **-=** 1 **if** element[**-**1] **==** '}' **else** 0

                    converted\_rules[nonterm]**.**append(converted\_rule)

**except** **ValueError** **as** e:

                e**.**args **=** (e**.**message **+** '\nIssues with rule "%s"' **%** rule, )

**raise**

        self**.**P **=** converted\_rules

*# Now we need to build transition dictionary i.e.*

*# transition\_dict[current\_rule][current\_input\_sybol] will give you rule*

*# that need to be applied in corresponding case*

        transition\_dict **=** dict([(A, dict()) **for** A **in** self**.**N])

**for** nonterm, rules **in** self**.**P**.**items():

**for** rule **in** rules:

*# add this rule for all possible first symbols*

                first\_symbols **=** self**.**\_first(rule)

**for** term **in** first\_symbols**.**difference(['']):

                    transition\_dict[nonterm][term] **=** rule

*# and if rule can become empty string add the rule for follows*

**if** '' **in** first\_symbols:

**for** term **in** self**.**\_follow(nonterm):

                        transition\_dict[nonterm][term] **=** rule

        self**.**transition\_dict **=** transition\_dict

**def** **\_first**(self, alpha):

        """\_first(string or sequence of strings) -> set of terminals

\* each string must be terminal or nonterminal symbol

The function takes single symbol or sequence of symbols (chain) and

returns set of terminals that you can meet in first position in a

derivation.

Given argument is 'A' and there is rule 'A -> a B', 'a' will be in the

set. If argument is ('A', 'B') and there are rules 'A -> a B | B ' and

'B -> c | d' , all 'a', 'b', and 'c' will be in the set.

"""

        s **=** set()

**if** type(alpha) **is** str: *# if alpha is single symbol*

*# if nonterminal*

**if** alpha **in** self**.**N:

*# than check rules*

**for** rule **in** self**.**P[alpha]:

**if** rule: *# is not empty*

                        s**.**update(self**.**\_first(rule))

**else**:

                        s**.**add('')

*# if terminal than return set only with the terminal*

**elif** alpha **in** self**.**T:

**return** set([alpha])

**else**: *# it is an attribute*

**return** set([''])

**else**: *# if chain*

**if** alpha:

*# checkin each symbol*

**for** i, symbol **in** enumerate(alpha):

                    first\_symbols **=** self**.**\_first(symbol)

                    s**.**update(first\_symbols**.**difference(['']))

*# if it can become empty string than all first symbols found*

**if** '' **not** **in** first\_symbols:

**break**

*# if all symbols in the chain can become empty string than*

*# the whole chain can*

**if** i **==** len(alpha) **-** 1 **and** '' **in** first\_symbols:

                        s**.**add('')

**else**:

**return** set([''])

**return** s

**def** **\_follow**(self, X):

        """\_follow(string) -> set of terminals

\* string must be nonterminal symbol

The function returns set of terminals that you can meet in right after

a derivation.

"""

        s **=** set()

*# if X is axiom*

**if** X **==** self**.**S:

            s**.**add('$')

*# for each rule that have X in right part*

**for** nonterm, rules **in** self**.**P**.**items():

**for** rule **in** filter(**lambda** e: X **in** e, rules):

*# for each X in rule*

**for** i, x **in** filter(**lambda** e: e[1] **==** X, enumerate(rule)):

*# if it isn't last one*

**if** i **<** len(rule) **-** 1:

*# for rule like A -> aXb b is the following chain*

                        following\_chain **=** rule[i **+** 1:]

                        following\_first **=** self**.**\_first(following\_chain)

                        s**.**update(following\_first**.**difference(['']))

*# if folowing chain can become empty*

**if** '' **in** following\_first:

*# do the same thing as X was at the end*

                            s**.**update(self**.**\_follow(nonterm))

*# if X at the end, than it can have same follow as*

*# nondeterminate on the left side*

**else**:

**if** nonterm **!=** X:

                            s**.**update(self**.**\_follow(nonterm))

**return** s

**def** **parse**(self, input\_):

        """parse(input)

input will be splitted in lexemes and processed by the automaton. All

python code from grammar's attributes will be executed inside this

method, you cat return something from it but return statement must be

the only statement in {}

"""

        rough\_split **=** input\_**.**split() *# first we split input by spaces*

        chain **=** []

**while** rough\_split:

            element **=** rough\_split**.**pop(0)

**if** element **in** self**.**T: *# if element in T than it was separated by*

*# spaces*

                chain**.**append(element)

**else**: *# we need to split it manualy*

                i **=** **-**1

**while** element[:i] **not** **in** self**.**T:

                    i **-=** 1

**if** i **<** **-**len(element):

**raise** **ValueError**('incorrect symbols: ' **+** element)

                chain**.**append(element[:i])

                rough\_split**.**insert(0, element[i:])

        chain **+=** ['$']

        \_stack **=** [self**.**S, '$']

**while** True:

*# code execution*

**if** \_stack[0][0] **==** '{' **and** \_stack[0][**-**1] **==** '}':

                code **=** \_stack**.**pop(0)[1:**-**1]

**try**:

**exec** code

**except** **SyntaxError** **as** e:

**if** e**.**args[0] **==** "'return' outside function":

**return** eval(code**.**split(None, 1)[1])

**else**:

**raise**

*# applying the rules*

**elif** \_stack[0] **in** self**.**N:

**try**:

                    rule **=** self**.**transition\_dict[\_stack[0]][chain[0]]

                    \_stack**.**pop(0)

                    \_stack **=** list(rule) **+** \_stack

**except** **KeyError**:

**raise** **ValueError**('Parsing Error')

*# outher actions*

**elif** \_stack[0] **in** self**.**T:

**if** \_stack**.**pop(0) **!=** chain**.**pop(0):

**raise** **ValueError**('Parsing Error')

**elif** \_stack[0] **==** '$':

**if** chain[0] **==** '$':

**return** True *# it mean that input is correct*

**else**:

**raise** **ValueError**('Parsing Error')

# Работа с программой

## Печать инфиксного выражения в постфиксной форме

### Схема трансляции

    DIGITS **=** [str(x) **for** x **in** range(10)]

    INFIX\_TO\_POSTFIX\_PRINT\_TRANSLATION\_SCHEME **=** {

        'T': {'+', '-', '\*', '/', '(', ')'}**.**union(DIGITS),

        'N': {'E', 'E`', 'T', 'T`', 'F', 'N', 'N`', 'D'},

        'P': {

            'E -> T E`',

            'E`-> + T {print "+",} E` | - T {print "-",} E` | ',

            'T -> F T`',

            'T`-> \* F {print "\*",} T` | / F {print "/",} T` | ',

            'F -> ( E ) | N',

            'N -> D N` {print "#",}', *# '#' for separating numbers*

            'N`-> D N` | ',

            'D -> ' **+** '|'**.**join('%s {print "%s",}' **%** (D, D) **for** D **in** DIGITS)

            },

        'S': 'E'}

### Работа анализатора

>>> automaton.parse("2 + 2/5 - 5\*(8 + 2)")

2 # 2 # 5 # / + 5 # 8 # 2 # + \* -

0: True

>>> automaton.parse("1 + 1\*25 + 634 - (1234 + 12)\*5")

1 # 1 # 2 5 # \* + 6 3 4 # + 1 2 3 4 # 1 2 # + 5 # \* -

1: True

>>> automaton.parse("((125 + 232) \* 33 - 44) / 66 + 35 / 2")

1 2 5 # 2 3 2 # + 3 3 # \* 4 4 # - 6 6 # / 3 5 # 2 # / +

2: True

>>> automaton.parse("11 \* (22 + 15) / 44 - 161 / 8")

1 1 # 2 2 # 1 5 # + \* 4 4 # / 1 6 1 # 8 # / -

3: True

## Перевод инфиксного выражения в постфиксную форму

### Схема трансляции

    INFIX\_TO\_POSTFIX\_STRING\_TRANSLATION\_SCHEME **=** {

        'T': {'+', '-', '\*', '/', '(', ')'}**.**union(DIGITS),

        'N': {'S', 'E', 'E`', 'T', 'T`', 'F', 'N', 'N`', 'D'},

        'P': {

            'S -> {result = ""} E {return result}',

            'E -> T E`',

            'E`-> + T {result += "+"} E` | - T {result += "-"} E` | ',

            'T -> F T`',

            'T`-> \* F {result += "\*"} T` | / F {result += "/"} T` | ',

            'F -> ( E ) | N',

            'N -> D N` {result += "#"}', *# '#' for separating numbers*

            'N`-> D N` | ',

            'D -> ' **+** '|'**.**join('%s {result += "%s"}' **%** (D, D) **for** D **in** DIGITS)

            },

        'S': 'S'}

### Работа анализатора

>>> automaton.parse("2 + 2/5 - 5\*(8 + 2)")

4: '2#2#5#/+5#8#2#+\*-'

>>> automaton.parse("1 + 1\*25 + 634 - (1234 + 12)\*5")

5: '1#1#25#\*+634#+1234#12#+5#\*-'

>>> automaton.parse("((125 + 232) \* 33 - 44) / 66 + 35 / 2")

6: '125#232#+33#\*44#-66#/35#2#/+'

>>> automaton.parse("11 \* (22 + 15) / 44 - 161 / 8")

7: '11#22#15#+\*44#/161#8#/-'

## Вычисление выражения

### Схема трансляции

    INFIX\_TO\_VALUE\_TRANSLATION\_SCHEME **=** {

        'T': {'+', '-', '\*', '/', '(', ')'}**.**union(DIGITS),

        'N': {'S', 'E', 'E`', 'T', 'T`', 'F', 'N', 'N`', 'D'},

        'P': {

            'S -> {s = []} E {return s[0]}',

            'E -> T E`',

            'E`-> + T {s.append(s.pop(-2) + s.pop())} E` | '

            ' - T {s.append(s.pop(-2) - s.pop())} E` | ',

            'T -> F T`',

            'T`-> \* F {s.append(s.pop(-2) \* s.pop())} T` | '

            ' / F {s.append(s.pop(-2) / s.pop())} T` | ',

            'F -> ( E ) | N',

            'N -> {s.append(0.)} D N`',

            'N`-> D N` | ',

            'D -> ' **+** '|'**.**join('%s {s.append(s.pop() \* 10 + %s)}' **%** (D, D)

**for** D **in** DIGITS)

            },

        'S': 'S'}

### Работа анализатора

>>> automaton.parse("2 + 2/5 - 5\*(8 + 2)")

8: -47.6

>>> automaton.parse("1 + 1\*25 + 634 - (1234 + 12)\*5")

9: -5570.0

>>> automaton.parse("((125 + 232) \* 33 - 44) / 66 + 35 / 2")

10: 195.33333333333334

>>> automaton.parse("11 \* (22 + 15) / 44 - 161 / 8")

11: -10.875

## Построение синтаксического дерева

### Схема трансляции

    INFIX\_TO\_GRAPHVIZ\_AST\_TRANSLATION\_SCHEME **=** {

        'T': {'+', '-', '\*', '/', '(', ')'}**.**union(DIGITS),

        'N': {'S', 'E', 'E`', 'T', 'T`', 'F', 'N', 'N`', 'D'},

        'P': {

            'S -> {t = []; E = []; l = []; id = 0} E {dot = "Digraph D{"'

                 '+ "label=\\"Syntax tree for expression " + input\_ + "\\";"'

                 '+ "".join("%i->%i;" % edge for edge in E)'

                 '+ "".join("%i[label=\\"%s\\"];" % label '

                                 'for label in enumerate(l))} '

                 '{dot += "}"} '

                 '{return dot}',

            'E -> T E`',

            'E`-> + T {l.append("+"); E.append((id, t.pop(-2))); '

                      'E.append((id, t.pop())); t.append(id); id += 1} E` | '

            ' - T {l.append("-"); E.append((id, t.pop(-2))); '

                      'E.append((id, t.pop())); t.append(id); id += 1} E` | ',

            'T -> F T`',

            'T`-> \* F {l.append("\*"); E.append((id, t.pop(-2))); '

                      'E.append((id, t.pop())); t.append(id); id += 1} T` | '

            ' / F {l.append("/"); E.append((id, t.pop(-2))); '

                      'E.append((id, t.pop())); t.append(id); id += 1} T` | ',

            'F -> ( E ) | N',

            'N -> {t.append(0)} D N` {l.append(t.pop()); t.append(id); id += 1}',

            'N`-> D N` | ',

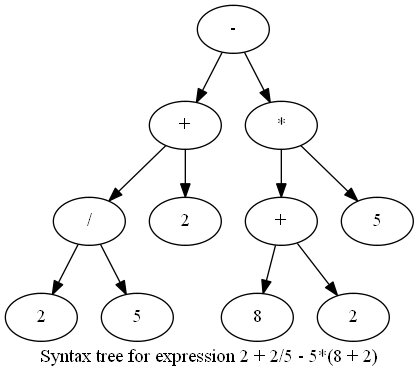
            'D -> ' **+** '|'**.**join('%s {t.append(t.pop() \* 10 + %s)}' **%** (D, D)

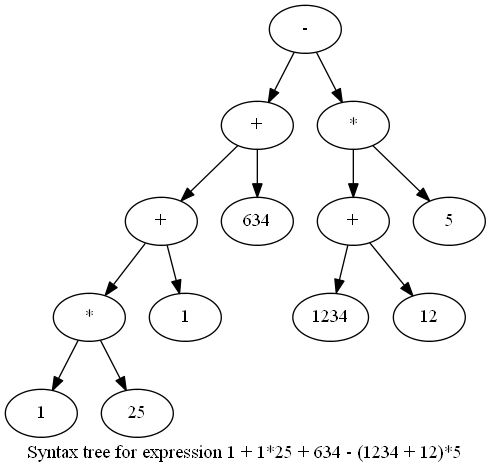
**for** D **in** DIGITS)

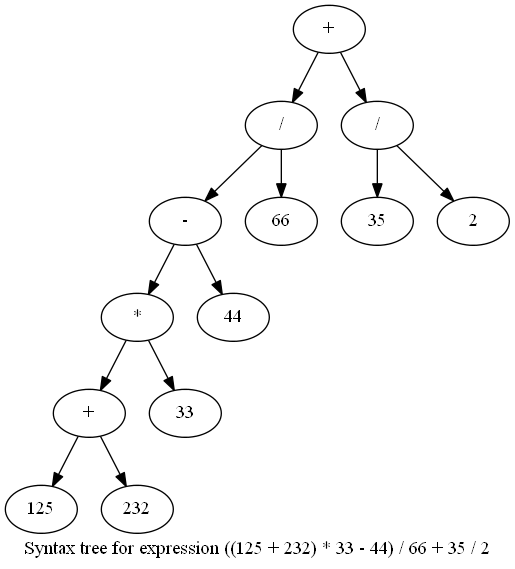
            },

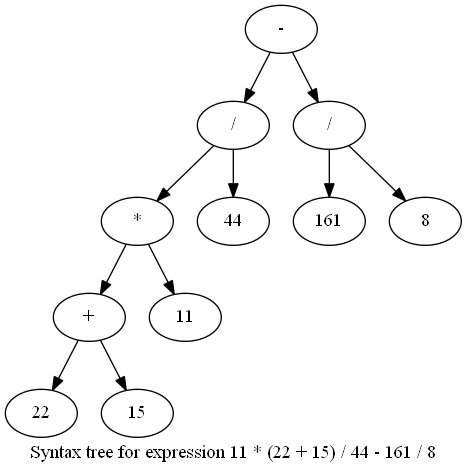
        'S': 'S'}

### Работа анализатора









# Вывод

В ходе лабораторной работы мы освоили синтаксически управляемую трансляцию