ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ РАДИОТЕХНИКИ,
ЭЛЕКТРОНИКИ И АВТОМАТИКИ (ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

КАФЕДРА ППО

КУРСОВАЯ РАБОТА

на тему

«Формальные грамматики и языки.

Элементы теории трансляции»

Выполнил:

Студент группы ВВ-х-06
няшка

Москва 2009

Оглавление

Задание 1. Разбор по диаграмме состояний

Основные определения

Теория формальных грамматик – теория построения слов по определённым правилам.

Алфавит - это конечное множество символов.

Предполагается, что термин "символ" имеет достаточно ясный интуитивный смысл и не нуждается в дальнейшем уточнении.

Цепочкой символов в алфавите V называется любая конечная последовательность символов этого алфавита.

Цепочка, которая не содержит ни одного символа, называется *пустой цепочкой*. Для ее обозначения будем использовать символ ∑.

Язык в алфавите V - это подмножество цепочек конечной длины в этом алфавите.

Порождающая грамматика G - это четверка (VT, VN, P, S), где

- VT - алфавит *терминальных символов (терминалов)*,
- VN - алфавит *нетерминальных символов (нетерминалов)*, не пересекающийся с VT,
- P - конечное подмножество *правил вывода,*
- S - *начальный символ (цель)* грамматики, принадлежит VN.

Грамматики можно разделить по степени строгости (количеству ограничений) на следующие категории:

1. Типа 0 – никаких ограничений.
2. Типа 1 – контекстно-зависимые грамматики.
3. Типа 2 – контекстно-свободные грамматики.

0. Типа 3 - регулярные.

Грамматика называется контекстно-свободной, если её правила порождают любую цепочку из терминальных и нетерминальных символов.

Постановка задания

Дана регулярная грамматика

$$\begin{cases} S \to a \mid Ba \\ B \to Bb \mid b \end{cases}$$

Узнать, какой язык она порождает. Построить по данной грамматике диаграмму состояний и написать анализатор для неё.

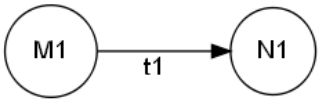
Порождаемый язык

Заданная грамматика порождает множество слов, которое на неформальном языке можно описать как либо символ *a*, либо произвольное количество символов *b*, за которыми следует символ *a*.

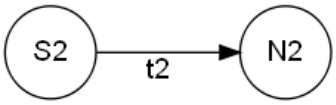
Диаграмма состояний

Построение диаграммы состояний производится по исходной грамматике по следующим правилам:

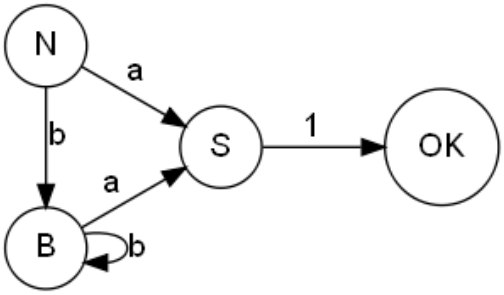
1. Правилу вида $N \to Mt$
сопоставляется ветвь на графе



2. Правилу вида $N \to t$
сопоставляется



Применяя эти правила к заданной грамматике, получим граф



Анализатор на языке Python

В ходе выполнения лабораторной работы была разработана программа, реализующая универсальный анализатор регулярных грамматик, причём упор в разработке делался на близость реализации к словесному алгоритму построения диаграммы состояний и анализа слов по ней.

В программе описаны классы state и FSM, реализующие функционал вершины графа в диаграмме состояний и всей диаграммы в целом, соответственно. Алгоритм

работы класса FSM можно описать следующим образом: при создании экземпляра класса происходит анализ заданной грамматики, выделение нетерминальных и терминальных символов, разбиение её на простейшие правила вида 1 и 2, и создание необходимых по алгоритму вершин диаграммы – соответствующих всем нетерминальным символам, а также вершины N, из которой и происходит начало анализа.

```
import re

class state():
    def __init__(self,name,isFinal=False):
        self.behave = {}
        isFinal = (isFinal or ('S'==name))
        self.final = isFinal
        self.name = name
    def setTransfer(self,input,end):
        self.behave[input] = end
    def setTransfers(self,transfers):
        for transfer in transfers:
            self.setTransfer(transfer[0], transfer[1])
    def transfer(self,input):
        return self.behave[input]

def getTerminals(str):
    terms = set([])
    for char in str:
        if (re.match('[a-z]',char) and not(char in terms)):
            terms.add(char)
    return terms

def getNonTerminals(str):
    nonTerms = set([])
    for char in str:
        if (re.match('[A-Z]',char) and not(char in nonTerms)):
            nonTerms.add(char)
    return nonTerms

def getElementaryRules(rule):
    '''splits a rule contaning multiple sub-rules'''
    ruleset = list()
    (definition,rules) = re.findall('(\S+)->(\S*)$', rule)[0]
    for match in re.findall('(\w+)',rules):
        ruleset.append("%s->%s" %(definition,match))
    return ruleset

def reduceList(inlist):
    inlist = list(set(inlist))

def parseGrammar(rules):
    (terms, nonTerms, ruleset) = (list(),list(),list())
    rules = rules.split('\n')
    for rule in rules:
        terms.extend(getTerminals(rule))
        nonTerms.extend(getNonTerminals(rule))
        ruleset.extend(getElementaryRules(rule))
    terms = list(set(terms))
    nonTerms = list(set(nonTerms))
    nonTerms.append('N')
    ruleset = list(set(ruleset))
    return (terms,nonTerms,ruleset)

```

```
55 def joinList(srclist):
56     str = ''
57     for element in srclist:
58         str+=element
59     return str
60 
61 def parseSimpleRule(srcrule,terms,nonTerms):
62     (definition,rule) = re.findall('(\S+)->(\S*)$', srcrule)[0]
63     regexp1 = "[%s][%s]$" %(joinList(nonTerms),joinList(terms))
64     regexp2 = "[%s]$" %(joinList(terms))
65     if (re.match(regexp1,rule)):
66         type = 1
67         src = rule[0]
68         dest = definition
69         symbol = rule[1]
70     elif (re.match(regexp2,rule)):
71         type = 2
72         src = 'N'
73         dest = definition
74         symbol = rule[0]
75     else:
76         raise "Not a determinative regular grammar given!"
77     return (src, symbol, dest)
78 
79 class FSM():
80     def __init__(self,rules):
81         self.states = {}
82         self.transfers = list()
83         (self.terms,self.nonTerms,self.ruleset) = parseGrammar(rules)
84         for rule in self.ruleset:
85             self.transfers.append(parseSimpleRule(rule,self.terms,self.nonTerms))
86 
87     def initStates(self):
88         for statenm in self.nonTerms:
89             self.states[statenm] = state(statenm)
90 
91     def resetState(self):
92         self.cstate = self.states['N']
93 
94     def createLinks(self):
95         for transfer in self.transfers:
96             self.states[transfer[0]].setTransfer(transfer[1],self.states[transfer[2]])
97 
98     def analyze(self,sentence):
99         for schr in sentence:
100             try:
101                 self.cstate = self.cstate.transfer(schr)
102             except:
103                 return False
104         return self.cstate.final
105 
106 test = FSM('S->a|Ba\nB->Bb|b')
107 #test = FSM('S->Sz|Ss|Dz|Ds\nD->Hd\nH->z|s|Hz|Hs')
108 test.initStates()
109 test.createLinks()
110 test.resetState()
111 print test.analyze('bbbbba')
112 #print test.analyze('szssszzsz')
```

Вывод

Разработанный анализатор успешно выполняет построение диаграммы состояний для регулярных грамматик и анализ заданного слова на соответствие ей.

Задание 2. Разработка лексического анализатора методом рекурсивного спуска

Основные понятия и определения

Метод рекурсивного спуска позволяет определить, принадлежит ли определённая цепочка заданному языку. Последовательность применений правил вывода эквивалентна построению дерева разбора методом "сверху вниз"

Метод рекурсивного спуска (РС-метод) реализует этот способ практически "в лоб": 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.

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики имеет вид:

а) либо $A \to \alpha$, где $\alpha \in (VT \cup VN)^*$ и это единственное правило вывода для этого нетерминала;

b) либо $A \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$, где $a_i \in VT^*$ для всех i = 1,2,...,n; $a_i \neq a_j$ для $i \neq j$, $\alpha_i \in (VT \cup VN)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по выше изложенной схеме.

Метод рекурсивного спуска применим к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, - входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован, в результате на обработку цепочки длины n расходуется время cn.

Исходный код

```
'''S -> E~
   E -> T | T+E | T-E
   T -> a | b | (E)'''

class syntaxTest():
    def __init__(self,phrase):
        self.text = phrase
        self.cur = 0
    def _advance(self):
        self.cur+=1
    def _check(self,expr):
        if not(expr):
            raise Exception('fail')
    def _curchr(self):
        if (self.cur==len(self.text)):
            return
        return self.text[self.cur]

    def _T(self):
        if not(self._curchr() in set(['a','b'])):
            if ('(' == self._curchr()):
                self._advance()
                self._E()
                self._check(')' == self._curchr())
                return
            self._check(False)
        else:
            return

    def _E(self):
        self._T()
        self._advance()
        if (self._curchr() in set(['+','-'])):
            self._advance()
            self._E()

    def _S(self):
        try:
            self._E()
            return (self.cur==len(self.text))
        except:
            return False

tester = syntaxTest('b-(b+a)+a+(a+b+b-a-(a-b))+a*b')
print tester._S()
```

Тестовые результаты

Входные данные	Результат
`b-(b+a)+a+(a+b+b-a-(a-b))+a*b`	`True`
`b-a`	`True`
`b-+a+(a+b+b-a-(a-b))+a*b`	`False`
`a*b+b*b`	`True`
`ab`	`False`

Вывод

Разработан анализатор для заданной грамматики, работающий по методу рекурсивного спуска, и успешно выполняющий задачу определения принадлежности заданной цепочки алфавиту.

Общее заключение

В ходе выполнения курсовой работы были разработаны программы для лексического синтаксического анализов языков по заданным грамматикам (регулярной и контекстно-свободной).

В качестве теоретической основы послужили теория формальных грамматик и теория трансляции.

Разработка производилась на языке Python.

Литература

1. Волкова И.А, Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции. МГУ, 1999.

2. А. Ахо, Дж.Ульман. Теория синтаксического анализа, перевода и компиляции. – Т. 1,2. – М., Мир, 1979.

3 Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоритические основы проектирования компиляторов. – М., Мир, 1979.

4. Д.Грис. Конструирование компиляторов для цифровых вычислительных машин. – М., Мир, 1975.

5. Ф.Вайнгартен. Трансляция языков программирования. – М., Мир, 1977.

6. И.Л. Братчиков. Синтаксис языков программирования. – М., Наука, 1975.

7 С.Гинзбург. Математическая теория контекстно-свободных языков. – М., Мир, 1970.

8. Дж.Фостер. Автоматический синтаксический анализ. – М., Мир, 1975.

9. В.Н. Лебедев. Введение в системы программирования. – М., Статистика, 1975.