西安财经大学 信息学院

编译原理 实验报告

姓名：陈伯硕
学号：1831050010
班级：计本 1801
指导教师：李薇

成绩：________

实验名称：预测分析 LL(1)

实验日期：2020 年 11 月 28 日

一、实验目的

1. 掌握 FIRST 和 FOLLOW 集合的构造方法；
2. 掌握预测分析表的构造；
3. 了解自顶向下预测分析的过程。

二、实验内容

1. 设计并实现 FIRST 和 FOLLOW 集合的求解算法；
2. 构造预测分析表并输出；
3. 判断该文法是否为 LL(1)文法；

三、 实验要求

1. 输入一个文法并输出显示；
2. 分别编写求 FIRST 和 FOLLOW 集合的函数；
3. 构造预测分析表并输出；
4. 根据预测分析表判断是否为 LL(1)文法；
5. 对任意输入串进行预测分析并给出分析过程。（选做）

四、 数据结构和算法设计

1. 文法的存储

接下来我们约定: 文法的输入需要遵守 LaTeX 语法并保证符号之间用空格隔开，ϵ 表示空串，一组非终结符需要加粗。对于用户来说，这样的输入输出较为美观，对于程序来说，可以直接根据空格拆分文法的每一个符号。

根据输入约定，构造数据结构 *rules*, 可以通过 Python 字典实现，对于每一个字典项，用列表存储文法的条目，每个条目是一个列表，作为符号序列，即 *Dict*[*str*, *List*[*List*[*str*]]], 该字典的键 (key) 为非终结符集合，同时记录非终结符集合 *termials*, 便于接下来算法处理, 也要记录开始符号，便于接下来使用

例如，对于

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' | \epsilon \\ T &\to FT' \\ T' &\to *FT' | \epsilon \\ F &\to (E) | \textbf{id} \end{aligned} \tag{1}$$

按照输入约定，输入一个列表 *g* 表示文法 (1)

```
g = [r"E \to T E'",
    r"E' \to + T E' | \epsilon ",
    r"T \to F T'",
    r"T' \to * F T' | \epsilon ",
    r"F \to ( E ) | \textbf{id}"]
```

在 python 中存储结构如下

```
defaultdict(list,
         {'E': [['T', "E'"]],
          "E'": [['+', 'T', "E'"], ['\\epsilon']],
          'T': [['F', "T'"]],
          "T'": [['*', 'F', "T'"], ['\\epsilon']],
          'F': [['(', 'E', ')'], ['\\textbf{id}']]})
```

更多详细的说明可以查阅本项目的 API 说明文档 https://compilers-homework.readthedocs.io/en/latest/ll1_parser/api/

2. first 集合的构造

程序 first(X) 集合规则如下

1. 如果 X 为终结符，则 first$(X) = \{X\}$

2. 若 X 为非终结符，$X \rightarrow Y_1, Y_2, \ldots, Y_k (k \geqslant 1), Y_1 Y_2 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$, 即 $\epsilon \in \text{first}(Y_j), j = 1, 2, \ldots i-1$, 将 $\text{first}(X_i)$ 中所有元素加入 $\text{first}(X)$
3. 对于 $X \rightarrow \epsilon$, 直接将 ϵ 加入 $\text{first}(X)$

根据对应规则和数据结构设计 算法 1

算法 1 生成终结符的 first 集合并存储

```
create_first()
queue nonterminals_rules                          ▷ 存储暂时没法处理的终结符
for left, rule in rules:                          ▷ 迭代每一条规则
    x ← rule[0]                                   ▷ x 为产生式左边第一个符号
    if x ∈ terminals:                             ▷ 产生式的第一个符号是终结符
        first(left) ← first(left) ∪ {x}           ▷ 将终结符加入左部的 first 集合
    else if x = ϵ:
        contain_empty ← contain_empty ∪ {x}
    else :                                        ▷ 终结符稍后处理
        nonterminals_rules .enque((left,rule))    ▷ 将对应的信息入队
while nonterminals_rules:                         ▷ 若队不空
    left, rule ← nonterminals.dequeue()
    x ← rule[0]
    if x ∈ firsts:                                ▷ firsts 为当前有 first 集合的非终结符的集合，
                                                  ▷ 也就是说，他们已经有非终结符在 first 集合中
        first(left) ← first(left) ∪ first(x)
    else :
        nonterminals_rules .enque((left, rule))
    if x ∈ contain_empty:                         ▷ x 可能包含空串
        if rule[1]:                               ▷ 有其他的符号
            nonterminals.enque((left, rule[1 :]))   ▷ 将后续字符加入处理队列
```

算法 1 的对应 python 实现见 `https://compilers-homework.readthedocs.io/en/latest/_modules/LL1Parser/#LL1Parser.create_first`

类似的，可以求表达式的文法，我们可以利用 算法 1 的结果。

算法 2 求 $first(\alpha)$

get_first(α)

$x \leftarrow \alpha[0]$ ▷ x 为表达式第一个字符

if $x \in$ *terminals* **or** $x = \epsilon$:

  return $\{x\}$

$s \leftarrow$ first(x)

while $\epsilon \in s$:

  $n \leftarrow get\_first(\alpha[1:])$ ▷ 递归的求出接下来的表达式的 first 集合

  $s \leftarrow s \cup n$

return s

这部分代码在`https://compilers-homework.readthedocs.io/en/latest/_modules/LL1Parser/#LL1Parser.get_first`中实现

3. follow 集合的构造

follow 集合规则如下

1. 如果 S 为终结符，则输入串的结束标记符 $\$ \in$ follow(S)
2. 对于产生式 $A \rightarrow \alpha B\beta$, (first$(\beta) - \{\epsilon\}) \subset$ follow(B)
3. 对于产生式 $A \rightarrow \alpha B$, 或 $A \rightarrow \alpha B\beta$, $\beta \overset{*}{\Rightarrow} \epsilon$, 则 follow$(A) \subset$ follow(B)

根据对应规则和数据结构设计 算法 3

算法 3 生成终结符的 first 集合并存储

create_follow()

follow$(S) \leftarrow$ follow$(s) \cup \{\$\}$ $\triangleright$ 将输入右标记加入起始符的 follow 集合中
queue q $\triangleright$ 保存 (A, B)follow$(B) \subset$ follow(A)
for l, α **in** rules:
    $add(\alpha, \epsilon)$ $\triangleright$ 在规则最后添加空串标记
    for $i = 0$ **to** length(α):
        if $\alpha[i] \in$ *nonterminals*:
            if $\alpha[i+1] \in$ *nonterminals*:
                follow$(\alpha[i]) \leftarrow$ follow$(\alpha[i]) +$ first$(\alpha[i+1]) - \{\epsilon\}$
            if $\alpha[i+1] \in \{X | X \overset{*}{\Rightarrow} \epsilon\}$:
                $q.enque\big((\alpha[i], l)\big)$
            else if $\alpha[i+1] \in$ *terminals*:
                follow$(\alpha[i]) =$ follow$(\alpha[i]) \cup \{\alpha[i+1]\}$
    $q.enque\big((None, None)\big)$ $\triangleright$ 结尾的标志符
    has_enlarged = False
    while q:
        $t, \ell = q$.deque()
        if ℓ**is None**:
            if not has_enlarged:
                break $\triangleright$ 集合不在增大，退出
            else :
                has_enlarged $\leftarrow$ False
        else :
            $A \leftarrow$ follow(t) $\triangleright$ 末尾作为较大的集合
            $B \leftarrow$ follow(ℓ) $\triangleright$ $B \subset A$
            $N \leftarrow A \cup B$
            if $A \neq N$: $\triangleright$ 集合已经被扩大
                has_enlarged $\leftarrow$ True
                follow(t) $\leftarrow N$
            q.enque$\big((t, \ell)\big)$

算法 3 的对应 python 实现见 `https://compilers-homework.readthedocs.io/en/latest/_modules/LL1Parser/#LL1Parser.create_follow`

4. 语法分析表的构造

语法分析表需要读写 $M[A,a]$ 的结构，可以用两重字典嵌套，对应值为一条产生式 *List*[*str*], 最终数据类型为 *Dict*[*str*, *Dict*[*str*, *List*[*str*]]]

语法分析表规则如下, 对于产生式 $A \to \alpha$, 执行

1. $\forall a \in \mathrm{first}(\alpha)$, 将 $A \to \alpha$ 加入 $M[A,a]$
2. 若 $\epsilon \in \mathrm{first}(\alpha)$, 对 $\forall$ 终结符 $b \in \mathrm{follow}(A)$, 将 $A \to \alpha$ 加入 $M[A,b]$. 若 $\epsilon \in \mathrm{first}(\alpha)$ 且 $\$ \in \mathrm{follow}(A)$, 将 $A \to \alpha$ 加入 $M[A,\$]$.
3. 若加入语法分析表时有别的产生式，抛出异常

根据对应规则和数据结构设计算法 4

算法 4 构造语法分析表

```
create_table()
for A, α in rules:
    f ← get_first(α)        ▷ 获得产生式的 first 集合
    for a in f:
        add(M[A, a], α)        ▷ 具体实现函数中会检测 M[A, a] 中是否已经有产生式并抛出异常
    if ϵ ∈ f:
        for b ∈ follow(A):
            add(M[A, b], α)
        if $ ∈ follow(A):
            add(M[A, $], α)
```

算法 4 的对应 python 实现见 `https://compilers-homework.readthedocs.io/en/latest/_modules/LL1Parser/#LL1Parser.create_table`

五、 实验步骤与结果

1. 测试文法的输入

根据上述数据结构首先测试文法的输入与保存，对于文法1

在 python 中导入写好的库，调用

```python
from LL1Parser import LL1Parser
g = [r"E \to T E'",
    r"E' \to + T E | \epsilon ",
    r"T \to F T'",
```

```
    r"T' \to *F T' | \epsilon ",
    r"F \to ( E ) | \textbf{id}"]
grammer.display(raw=True)
```

输出文法 (1) 的源代码和渲染结果

```
\begin{array}{l}
  E & \to T E' \\
  E' & \to + T E \\
     &\;\, |\;\, \epsilon \\
  T & \to F T \\
  T' & \to *F T' \\
     &\;\, |\;\, \epsilon \\
  F & \to ( E ) \\
     &\;\, |\;\, \textbf{id}\\
\end{array}
```

2. 求 first,follow 集合并测试

接下来根据算法 1, 用文法 (1) 测试集合，得到

$$\begin{aligned}
\text{first}(E') &= \{+\} \\
\text{first}(T') &= \{*\} \\
\text{first}(F) &= \{(, \textbf{id}\} \\
\text{first}(T) &= \{(, \textbf{id}\} \\
\text{first}(E) &= \{(, \textbf{id}\}
\end{aligned} \tag{2}$$

对于文法 (1) 测试 follow，根据 算法 3 测试得到该文法的 follow 集合

$$\begin{aligned}
\text{follow}(E) &= \{\$,)\} \\
\text{follow}(T) &= \{+, \$,)\} \\
\text{follow}(F) &= \{+, *, \$,)\} \\
\text{follow}(E') &= \{\$,)\} \\
\text{follow}(T') &= \{+, \$,)\}
\end{aligned} \tag{3}$$

3. 构造语法分析表

接下来根据??, 得到文法 (??) 的文法分析表如表 1

表 1　文法 (1) 语意分析表

terminal	**id**	\$	$*$	(	$+$	)
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to \epsilon$			$E' \to +TE'$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

4. 测设非 LL(1) 文法

下面测试非 LL(1) 文法4

$$\begin{aligned} S &\to iEtSS' \\ &\mid a \\ S' &\to eS \\ &\mid \epsilon \\ E &\to b \end{aligned} \tag{4}$$

非 LL(1) 文法在将语法添加到语法分析表的时候会报错，此时抛出异常如下

```
RuntimeError: It isn't a LL(1) grammar,
          new added M[S'][e] = ['\\epsilon']
          conflict with existing M[S'][e] = ['e', 'S']
          parsing table
```

六、 实验总结

这次报告网上现有的资料较少，搜出的代码可读性不高，所以本文算法设计根据编译原理 [1] 的描述直接做出，可能算法不够精炼，但是实现的过程感受到该书描述的严谨细致，通过语言描述可以直接构造相应的算法和数据结构。

代码实现是通过我对规则的理解给出，实现的过程中也在不断改进美化代码和简化逻辑，这样的过程增加了我程序设计的经验和技巧，利于接下来进一步学习。另外代码的许多部分不够简洁，如 first(X) 和 first(α) 的算法有逻辑重叠，希望在有时间的时候对代码进行优化。

通过这些设计研究，我了解了计算机设计的抽象问题，分析问题，解决问题的方法和步骤，这些思考和实现有利于接下来的程序设计和逻辑思维。

参考文献

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*, volume 7. Addison Wesley, 2 edition, 2006.

附录 A　依赖的安装

本文代码的依赖如下

```
network2tikz
IPython
jupyter-sphinx
```

可以通过以下命令在清华镜像网站安装全部

```
  pip install -i http://mirrors.aliyun.com/pypi/simple \
  --trusted-host mirrors.aliyun.com/pypi/simple/ \
  -r requirements.txt
```

附录 B　参考文档

- API 文档:https://compilers-homework.readthedocs.io/en/latest/ll1_parser/api/
- 完整项目 https://github.com/chenboshuo/compilers_homework

附录 C　源代码

```python
from collections import defaultdict, deque
from typing import *
from IPython.display import display, Math, Latex
import itertools


class LL1Parser:
    r"""a set of algorithms to analyse the LL(1) grammer

        :param rules: the LL(1) grammer,you need input using latex gammer and
            sparated by spaces.

            for instance, in you want to input:

            :math:`E \to T E`
```

```
    :math:`E' \to + T E | \epsilon`

    :math:`T \to F T`

    :math:`T' \to *F T' | \epsilon`

    :math:`F \to ( E ) | \textbf{id}`

    then run:

    .. jupyter-execute::

        from LL1Parser import LL1Parser
        g = [r"E \to T E'",
            r"E' \to + T E' | \epsilon ",
            r"T \to F T'",
            r"T' \to * F T' | \epsilon ",
            r"F \to ( E ) | \textbf{id}"]
        grammer = LL1Parser(g)

:type rules: List[str]
:param start_symbol: the start symbol
:type start_symbol: str, optional

then you can display the rules using `display_rules()`, for example:

.. jupyter-execute::

    grammer.display_rules()

You can see the first sets and follow sets
using `display_first_sets()`, for examples:

.. jupyter-execute::

    grammer.display_first_sets()

Similarly, you can see the follow set using:

.. jupyter-execute::

    grammer.display_follow_sets()

The :meth:`LL1Parser.display_parsing_table`
can show the parsing table
```

```
        .. jupyter-execute::

            grammer.display_parsing_table()

    """

    def __init__(self, rules: List[str], start_symbol: str = None) -> None:
        """init the LL(1) grammer
        """
        # set the start symbol
        if start_symbol:
            self.start_symbol = start_symbol # : the start symbol
        else:
            self.start_symbol = rules[0].split()[0]
        # save rules
        self.rules: Dict[str, List[str]] = defaultdict(list)
        """the rules of the gammer,
            `self.rules[T] = L`,
            `L[i]= items`,
            `items[k] = symbol`,
            where `T` is a terminal,
            `i` is the index of alternatives
            `items` is the list of the symbols of a rule
        """

        self.terminals: set = set()
        """
        The list of terminals in table parsing table
        (include $, not include :math:`\epsilon`)
        """

        for rule in rules:
            alternatives = rule.split('|') # find rules connect by |
            first_part = alternatives[0].split()
            left = first_part[0] # the nonterminal can find in the first part
            # add elements except left symbol and ->
            self.rules[left].append(first_part[2:])
            self.terminals.update(first_part[2:])

            for alternative in alternatives[1:]:
                alternative_symbols = alternative.split()
                self.rules[left].append(alternative_symbols)
                self.terminals.update(alternative_symbols)

            self.terminals = self.terminals - \
                set(self.rules.keys()) - set([r'\epsilon'])
            self.terminals.update([r'\$'])
```

```
        # create first
        self.first: Dict[str, set] = defaultdict(set)
        """the first symbol dicts
        """
        self.contains_empty: set = set([r'\epsilon'])
        """the set of terminals that contains empty strings
        """
        self.create_first()

        # create follow set
        self.follow = defaultdict(set)
        """the follow set of every nonterminal
        """
        self.create_follow()

        self.parsing_table: Dict[str, Dict[str, List[str]]] \
            = defaultdict(dict)
        """ the parsing table
        self.parsing_table[A][a] = (rule)
        where A is the nonterminal at the left of rule,
        a is the nonterminal
        """

        # create parsing table
        self.create_table()

    def display_rules(self, raw=False):
        """display the latex code of the gammer

        :param raw: show the raw code, defaults to False
        :type raw: bool, optional
        """
        begin = r"\begin{array}{ll}" + "\n"
        end = r"\end{array}"+"\n"

        for left, rules in self.rules.items():
            s = self.display_rule(left, rules[0])
            for r in rules[1:]:
                s += self.display_rule(left, rule=r, is_alternative=True)
            begin += s

        if raw:
            print(begin+end)

        display(Math(begin+end))
```

```python
def display_rule(self, left: str,
                 rule: List[str], new_line=True,
                 array_environment=True,
                 is_alternative=False) -> str:
    """display a rule

    :param left: left symbol
    :type left: str
    :param rule: the list of the rule
    :type rule: List[str]
    :param new_line: whether need a new line at the end, defaults to True
    :type new_line: bool, optional
    :param array_environment: whether the expression in array environment
    :type array_environment: bool
    :param is_alternative: whether the rule is the alternative, defaults to False
    :type is_alternative: bool, optional
    :return: the latex string of the rule
    :rtype: str
    """
    if not is_alternative: # need left symbol
        s = "\t"+left
        if array_environment:
            s += r" & " # TODO handle the & symbol
        s += r" \to "
    else:
        s = "\t\t"
        if array_environment:
            s += r"&"
        s += r"\;\, |\;\,"

    s += " ".join(rule)
    if new_line:
        s += r" \\" + "\n"

    return s

def create_first(self):
    """create the first sets

    :ivar nonterminals: the queue to save the first nonterminal of
        a rule
    :vartype nonterminals: collections.deque
    :ivar can_emptystring: the set of nonterminals can be empty
    :vartype can_emptystring: set
    """
    nonterminals = deque()
```

```
        for left, rule in self.iter_rules():
            if rule[0] in self.terminals: # symbol rule[0] is terminal
                self.first[left].add(rule[0])
            elif rule[0] == r'\epsilon':
                self.contains_empty.add(left)
            else: # rule[0] is nonterminal
                nonterminals.append((left, rule))

        while nonterminals:
            left, rule = nonterminals.popleft()
            if rule[0] in self.first: # first symbol is nonterminal
                self.first[left].update(self.first[rule[0]])
            else: # pending
                nonterminals.append((left, rule))

            if rule[0] in self.contains_empty: # first symbol can empty string
                if rule[1:]:
                    nonterminals.append((left, rule[1:]))

    def iter_rules(self) -> tuple:
        """iter all the rules

        :yield: the iterator of (left,rule)
        :rtype: Iterator[tuple]
        """

        for left, rules in self.rules.items():
            for rule in rules:
                yield (left, rule)

    def display_sets(self, name, raw=False):
        """display the sets

        :param names: the set names(first,follow)
        :type name: str
        :param raw: show the raw latex code, defaults to False
        :type raw: bool, optional
        """
        begin = r"\begin{array}{ll}" + "\n"
        end = r"\end{array}" + "\n"
        contents = begin
        for left, first_set in self.__dict__[name].items():
            s = "\t" + r"\mathrm{" + name + r"}(" + left + r") &= \{"
            s += ", ".join(list(first_set))
            s += r"\} \\" + "\n"
            contents += s
        contents += end
```

```python
        if raw:
            print(contents)

        display(Math(contents))

    def display_first_sets(self, raw=False):
        """render the first(i) in jupyter notebook

        :param raw: the raw code of latex, defaults to False
        :type raw: bool, optional
        """
        self.display_sets(raw=raw, name='first')

    def display_follow_sets(self, raw=False):
        """render the follow(i) in jupyter notebook

        :param raw: the raw code of LaTeX, defaults to False
        :type raw: bool, optional
        """
        self.display_sets(raw=raw, name='follow')

    def create_follow(self):
        r"""create the follow sets of all nonterminals

            1. place the $ symbol in follow(S),
            where S is the start symbol,
            and $ is the input right endmarker

            2. if there is a production :math:`A \to \alpha B \beta`,
            then everything in `first(B)` except :math:`\epsilon`
            is in `follow(B)`

            3. if there is a production :math:`A \to \alpha B`,
            or a production :math:`A \to aB\beta`,
            where first(:math:`\beta`) contains `\epsilon`,
            then everything in `follow(A)` is in `follow(B)`
        """

        self.follow[self.start_symbol].add(r'\$') # add end symbol
        to_union = deque() # (A,B) such that set B is the subset of set A
        empty_symbol = set([r'\epsilon'])
        for left, rule in self.iter_rules():
            for cur, post in \
                itertools.zip_longest(rule, rule[1:],
                                      fillvalue=r'\epsilon'):
                if cur in self.rules: # cur is nonterminal
                    if post in self.rules: # post is nonterminal
```

```
                self.follow[cur].update(self.first[post]-empty_symbol)
            if post in self.contains_empty:
                # follow(cur) contains follow(left)
                to_union.append((cur, left))
            elif post in self.terminals:
                self.follow[cur].add(post)
    to_union.append((None, None)) # add the terminal symbol
    has_enlarged = False
    while to_union:
        tail, left = to_union.popleft()
        if left is None:
            if not has_enlarged:
                break
            else:
                has_enlarged = False
                to_union.append((None, None))
        else:
            set_, subset = self.follow[tail], self.follow[left]
            new_set = set_.union(subset)
            if set_ != new_set:
                has_enlarged = True
                self.follow[tail] = new_set
            to_union.append((tail, left))

def add_to_table(self, left: str, terminal: str,
                 rule: List[str]):
    r"""add the rule to the predictive parsing table

    :param left: the left of the production
    :type left: str
    :param terminal: the related terminal
    :type terminal: str
    :param rule: the rule of the right
    :type rule: List[str]
    :raises RuntimeError: conflict in add items,
        that means the grammer isn't LL(1) grammer.
        for example, if you have the grammer

        .. jupyter-execute::

            from IPython.display import display, Math, Latex
            w = [r"S \to i E t S S' | a",
                r"S' \to e S | \epsilon",
                r"E \to b"]
            for g in w:
                display(Math(g))
```

```
        then you create the instance,
        you will see the information

        .. jupyter-execute::
            :raises:

            wrong = LL1Parser(w)
    """

    if terminal in self.parsing_table[left] and rule != self.parsing_table[left][terminal]:
        raise RuntimeError(f"""It isn't a LL(1) grammar,
        new added M[{left}][{terminal}] = {rule}
        conflict with existing M[{left}][{terminal}] = {self.parsing_table[left][terminal]}
        parsing table.
        """)
    self.parsing_table[left][terminal] = rule

def create_table(self):
    r"""create a predictive parsing table
    For each production :math:`A \to \alpha` of the grammar,
    do the following:

    1. For each terminal :math:`a`, add :math:`A \to \alpha`
    to `M[A,a]`.

    2. If :math:`\epsilon` in first(:math:`\alpha`),
    then for each terminal b in follow(A),
    add :math:`A \to \alpha` to `M[A,b]`.
    If :math:`\epsilon` in first(:math:`\alpha`)
    and $ in follow(A),
    add :math:`A \to \alpha` to M[A,$] as well.
    """
    for left, rule in self.iter_rules():
        first = self.get_first(rule)
        for terminal in first:
            self.add_to_table(left=left, terminal=terminal, rule=rule)
        if r'\epsilon' in first: # epsilon is in first symbol
            for i in self.follow[left]: # every symbol should add to table
                self.add_to_table(left=left,
                                  terminal=i, rule=rule)
            if r'\$' in self.follow[left]:
                self.add_to_table(left=left,
                                  terminal=r'\$', rule=rule)

# TODO calculate the first of production, and store them
# Then calculate tht first of nonterminal
def get_first(self, rule: List[str]) -> set:
```

```
    r"""return the first symbol of a expression(calculate first(:math:`\alpha`))

    :param rule: the list of rule symbols
    :type rule: List[str]
    :return: the set of the first set
    :rtype: set
    """
    if rule[0] in self.terminals or rule[0] == r'\epsilon':
        return set([rule[0]])
    first_set = self.first[rule[0]]
    first_part = self.first[rule[0]]
    while r'\epsilon' in first_part:
        first_part = self.get_first(rule[1:])
        first_set.update(first_part)

    return first_set

def display_parsing_table(self, raw=False):
    begin = r"\begin{array}{|"
    begin += r"|".join(["c"]*(len(self.terminals)+1))
    begin += r"|}" + '\n'
    end = r"\end{array}"

    # create the header
    header = "\t\hline \n" + "\t" +r"\text{terminal}"
    for terminal in self.terminals:
        header += "\t&"
        header += terminal
    header += r'\\ \hline' + "\n"
    cells = ""
    for nonterminal in self.rules.keys():
        line = "\t" + nonterminal
        for terminal in self.terminals:
            line += " \t&"
            if terminal in self.parsing_table[nonterminal]:
                line += self.display_rule(left=nonterminal,
                                          rule=self.parsing_table[nonterminal][terminal],
                                          new_line=False,
                                          array_environment=False)
        # line += r"\\ \hline" + '\n'
        line += r"\\ " + '\n'
        cells += line

    cells += r"\hline" + "\n"
    content = begin+header + cells + end
    display(Math(content))
    if raw:
```

```
        print(content)
```