LL(1) Parser in Python

A simple LL(1) parser implemented in Python, correctly and concisely, fully OOP.

Features

- 面向对象编程实现，依靠 `Parser` 类和 `Grammar` 类实现整个算法流程，思路清晰简洁。
- 使用Python开发，并将代码封装为了一个Python module。
- 生成预测分析表使用科学计算库 `pandas` ，将表保存在 `csv` 文件中，可用Microsoft Excel打开，表格信息清楚明了，实现优雅。
- 实现了课本所描述的错误处理，包括遇到空白表项和 `synch` 的情况。

Environment & Usages

我们采用Python3.6在Windows10系统下开发，代码已经封装成Package。我们认为只要你的Python版本是3.0+即可正常使用这个Package，但我们仍然推荐您使用3.6及以上的版本。代码中使用的第三方库为 `pandas` ，用于输出预测分析表。故请确保您已安装 `pandas` 。

安装方法：在terminal（Linux）或PowerShell/Cmd Prompt（Windows）中输入

```
pip install pandas
```

运行方法：

进入目录：

```
cd parser
```

用户需要提供几个参数，想了解具体用法可以在terminal或PowerShell/Cmd Prompt中输入：

```
python -m LL1_Parser --help
```

可以得到以下信息：

```
usage: LL1-Parser [-h] --grammar GRAMMAR --tokens TOKENS [--output OUTPUT]

optional arguments:
  -h, --help         show this help message and exit
  --grammar GRAMMAR  The filename of the grammar (default: None)
  --tokens TOKENS    The token stream to analyze (default: None)
  --output OUTPUT    The output filename (default: process.txt)
```

即用户需要提供包含文法的文件名、包含token流的文件名以及输出文件名；输出文件名不是必须的，默认值为 `process.txt` 。具体两个输入文件应使用何种格式，我们将稍后介绍。

示例：

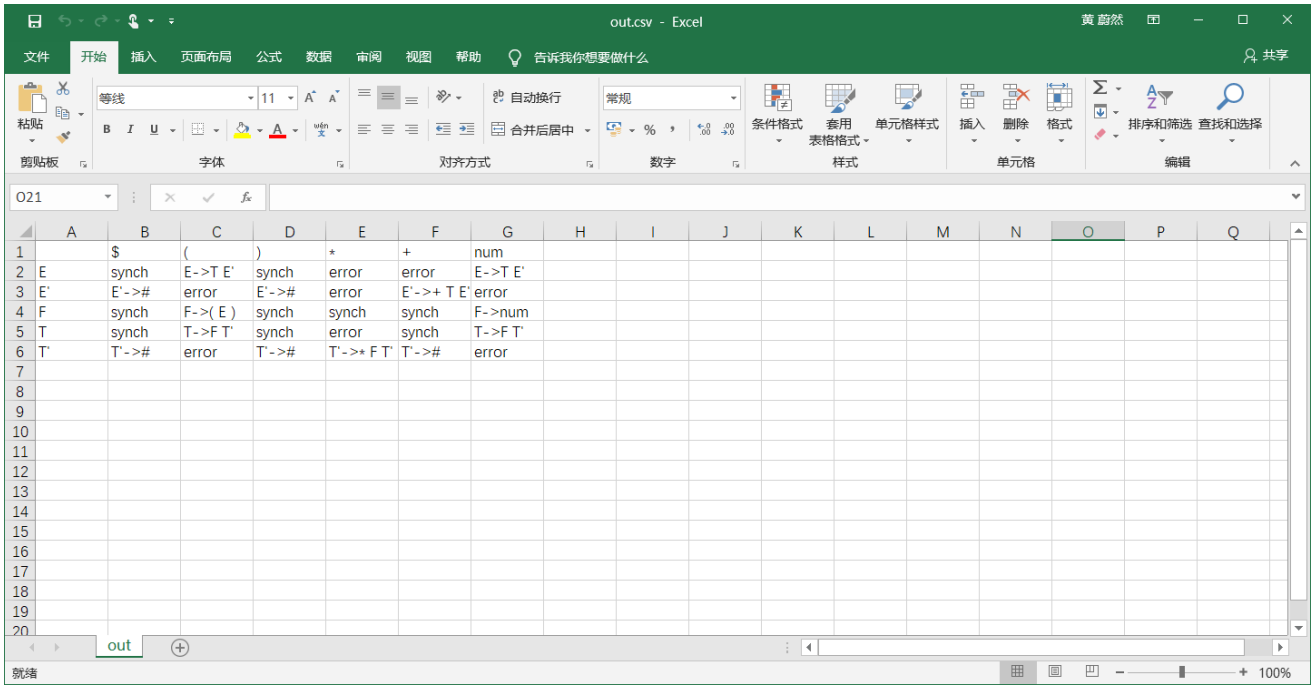
```
python -m LL1_Parser --grammar grammar2.txt --tokens tokens.txt
```

运行成功后，你将在命令行窗口看到预测分析程序的过程输出（测试用例及其输出结果将在下文讨论）；这部分内容同时也被保存在上述输出文件（ process.txt 或你指定的其他文件名）中。

输出的文件格式大致如此例：

```
**********Process Begin**********
------------------------------------------------------------
[Stack Top] E
[Current Token] (
[Stack] ['$', 'E']
[Accepted Tokens] []
[Using production] E -> T E'
------------------------------------------------------------
[Stack Top] T
[Current Token] (
[Stack] ['$', "E'", 'T']
[Accepted Tokens] []
[Using production] T -> F T'
------------------------------------------------------------
[Stack Top] F
[Current Token] (
[Stack] ['$', "E'", "T'", 'F']
[Accepted Tokens] []
[Using production] F -> ( E )
------------------------------------------------------------
[Stack Top] (
[Current Token] (
[Stack] ['$', "E'", "T'", ')', 'E', '(']
[Accepted Tokens] []
[Successful Match] pop symbol: (
------------------------------------------------------------
[Stack Top] E
[Current Token] num
[Stack] ['$', "E'", "T'", ')', 'E']
[Accepted Tokens] ['(']
[Using production] E -> T E'
------------------------------------------------------------
< 以下省略 >
```

还有另一个输出文件： out.csv 。这个文件保存分析文法后所求得的预测分析表。为了更好地展示结果，我们选择将它设置为 csv 文件。 csv 文件用Microsoft Excel可以打开，打开后如下：



	A	B	C	D	E	F	G
1		$	(	)	*	+	num
2	E	synch	E->T E'	synch	error	error	E->T E'
3	E'	E'->#	error	E'->#	error	E'->+ T E'	error
4	F	synch	F->(E)	synch	synch	synch	F->num
5	T	synch	T->F T'	synch	error	synch	T->F T'
6	T'	T'->#	error	T'->#	T'->* F T'	T'->#	error

Lab Description

题目：语法分析程序的设计与实现

实验内容：编写语法分析程序，实现对算术表达式的语法分析。要求所分析算术表达式由如下的文法产生：

$$E \to E+T \mid E-T \mid T$$
$$T \to T*F \mid T/F \mid F$$
$$F \to (E) \mid a$$

实验要求：在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

编写LL(1)语法分析程序，为给定文法自动构造预测分析表。

1. 编程实现算法4.2， 为给定文法自动构造预测分析表。
2. 编程实现算法4.1，构造LL(1)预测分析程序。

Pseudocode for involved algorithms

*The following pseudocodes are mostly from our textbook, for those not written in a concise way, I replaced them with their counterpart in the **dragon book**.*

算法4.2：**构造一个预测分析表**，伪代码如下：

input: 文法G

output：文法G的预测分析表M

> for(文法G的每一个产生式$A \to \alpha$){
>
> for(每个终结符号 $a \in FIRST(\alpha)$) 把 $A \to \alpha$ 加入$M[A,a]$中;
>
> if ($\epsilon \in FIRST(\alpha)$)
>
> for (每个$b \in FOLLOW(A)$) 把$A \to \alpha$ 放入表项$M[A,b]$中;
>
> }
>
> for (所有无定义的表项$M[A,a]$) 标上错误标志。

算法4.1：**表驱动的预测语法分析**，伪代码如下：（以下参考自 *Compilers:Principles, Techniques, &Tools* ）

> **input**: 输入符号串ω, 文法G的预测分析表M
>
> **output**: 若$\omega \in G$，则输出ω的最左推导，否则报告错误
>
> 令ip是输入指针，设置ip使它指向ω的第一个符号, 令a是它指向的符号;
>
> 令X是栈顶文法符号;
>
> while (X! = \$) { /* 栈非空 */
>
> if ($X == a$) {从栈顶弹出X； ip向前移一个位置;}
>
> else if (X是一个终结符号) error();
>
> else if ($M[X,a]$是一个报错条目) error();
>
> else if ($M[X,a] = X \to Y_1 Y_2 \cdots Y_k$){
>
> 输出产生式 $X \to Y_1 Y_2 \cdots Y_k$;
>
> 弹出栈顶符号;
>
> 将$Y_K, Y_{k-1}, \cdots, Y_1$ 压入栈中，其中Y_1 位于栈顶。
>
> }
>
> 令 X = 栈顶符号;
>
> }

计算FISRT集：（以下参考自 *Compilers:Principles, Techniques, &Tools* ）

> 计算各个文法符号X的$FIRST(X)$时，不断应用下列规则，直到再也没有新的终结符号或 ϵ 可以被加入到任何FIRST集合中为止：
>
> 1. 如果X是一个终结符号，那么$FIRST(X) = X$.
> 2. 如果X是一个非终结符号，且 $X \to Y_1 Y_2 \cdots Y_k$ 是一个产生式，其中$k \geq 1$，那么如果对于某个 i，$Y_1 Y_2 \cdots Y_{i-1} \Rightarrow \epsilon$，且 $a \in FIRST(Y_i)$，把a加入$FIRST(X)$中。如果$\epsilon \in FIRST(Y_i) \forall i \in [1,k]$，把 ϵ 加入 $FIRST(X)$。
> 3. 如果 $X \to \epsilon$ 是一个产生式，那么将 ϵ 加入 $FIRST(X)$.

计算FOLLOW集：

> 计算所有非终结符号A的$FOLLOW(A)$集合时，不断应用下列规则，直到再也没有新的终结符可以被加入任何FOLLOW集合中为止：
>
> 1. 将$ 放入$FOLLOW(S)$中.
> 2. 如果存在产生式$A \to \alpha B\beta$，那么$FIRST(\beta)$中除了ϵ的所有符号都在$FOLLOW(B)$中.
> 3. 如果存在一个产生式$A \to \alpha B$，或存在产生式$A \to \alpha B\beta$且$FIRST(\beta)$包含ϵ，那么$FOLLOW(A)$中的所有符号都在$FOLLOW(B)$ 中.

Our implementation

Input format

如上所述，我们需要两个输入文件： `GRAMMAR` 是我们依据的文法， `TOKENS` 是需要分析的记号流（它可能是文法的一个句子）。

`GRAMMAR` 所规定的输入格式如下：

- **产生式格式**：我们规定非终结符必须以大写字母开头，且所有以大写字母开头的符号都是非终结符； `#` 表示空字符，即ϵ；用 `->` 表示产生符号， `|` 表示产生式右端的并列。产生式中任意两个不同的符号用空格隔开，否则会被认为是同一个符号。如我们认为： `Anything -> You Say` 是由 `Anything` 这个符号，产生两个符号 `You` 和 `Say` 。
- 第一行为起始符的产生式。

A example

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | num
```

Handle the grammar

首先需要对文法进行处理。我们建立了一个 `Grammar` 类，来完成读取文法、消左递归、求FIRST集、求FOLLOW集的操作。其大致框架如下：

```python
class Grammar(object):
    def __init__(self):
        pass

   def read_grammar(self, filename):
        '''
        read the grammar from file.
        '''
        pass
    def remove_recursion(self):
        '''
        eliminate left-recursion in the grammar
        '''
        pass
```

```
    def get_first(self):
        '''
        get the first set of a given string or symbol
        '''
        pass
    def get_first_dict(self):
        '''
        get the first sets for all symbols
        '''
    def get_follow_dict(self):
        '''
        get the follow sets for all non-terminals
        '''
        pass
```

Read the grammar

从文件中读取文法。按上面的格式保存的文法将被我们的算法正确地读取，保存在一个Python `dict` 中。产生式的左侧和右侧分别用一个 `tuple` 来保存，以便区分不同长度的符号。以上面示例中的文法为例，我们读取后的字典如下所示：

```
dict(('E',):{('T', "E'")},
     ("E'",):{('+', 'T',"E'"), ('#',)},
     ('T',):{('F', "T'")},
     ('T',):{('*','F',"T'"), ('#',)},
     ('F',):{('(','E',')'), ('a',)})
```

即我们把推导式的右侧保存在一个 `set` 中，作为推导式左侧字符（也就是字典的每一个 `key` ）所对应的 `value` 。

remove left recursions

这里我们仅处理**直接左递归**。形如：

$$A \to A\alpha \mid \beta$$

我们把它处理为：

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$

首先进行扫描，查找包含左递归的非终结符；我们规定新符号为原符号加上一个 `'` （prime符号），产生新推导式，并更新。

get the first set

在手工求解FIRST集的时候，我们通常采用自下而上的方法，即从终结符开始。对于手工求解而言，这样做是符合直觉且简单的；但是在编写程序时，自上而下会使代码更加简洁。

自上而下，意味着我们要采取**递归**的方法。算法流程如下：

> 对于一串字符 `string` ：

- 如果它以终结符开头（按我们的约定，即 `string[0].isUpper() == True` ），那么这串字符的FIRST集仅有一个元素，就是这个终结符。
- 如果 `string[0]` 是非终结符，那么讨论这个非终结符的推导式。
  - 这个非终结符能推导出的所有非 ϵ 终结符显然都应该被加入该串的FIRST集。
  - 如果它能推导出 ϵ，则串 `string[1:]` 的FIRST集的所有都可以加入 `string` 的FIRST集。

这个算法效率有限，但写起来非常简洁，在处理经过消左递归、提左因子的文法时，也足以胜任。

get the follow set

求解FOLLOW集时，我们采用了和上面算法类似的实现：

- 先将 $ 加入 $FOLLOW(S)$中
- 遍历每组产生式，对于一个产生式的右侧，再遍历其每一个符号。为求方便我们称当前符号为 `cur` ，下一个符号为 `next` 。
  - 如果 `cur` 是终结符号，不进行操作
  - 如果 `cur` 是非终结符号，且是当前产生式最后一个符号，把产生式左侧的FOLLOW集加到FOLLOW(cur)中。
  - 如果 `cur` 是非终结符号，且不是最后一个字符，把FIRST(next)中非 ϵ 的字符加入FOLLOW(cur)；如果FIRST(next)中包含 ϵ，把产生式左侧非终结符的FOLLOW集合加到FOLLOW(cur)中。

  重复上述操作，直到所有FOLLOW集合都不再改变。

Generating the table

遍历所有产生式：

- 对一个产生式的右端求FIRST集，再遍历这个FIRST集中的元素：
  - 如果这个元素不是 ϵ: 在表中[产生式左端，该元素]的位置加入该产生式
  - 否则：遍历产生式左端非终结符的FOLLOW集：
    - 把这个产生式加入表中[产生式左端，FOLLOW集中元素]
- 生成完毕后，如果表中一个位置有多个产生式：

  报错，**非LL(1)文法**
- 再遍历所有非终结符的FOLLOW集的每一个元素：
  - 如果表中[非终结符，FOLLOW集的当前元素]为空：

    在该位置加入 `synch`

至此，文法的部分就完成了。接下来的工作交给 `Parser` 。

Parsing

`class Parser` 从文件中读取文法存入其成员 `grammar` 中，并对 `grammar` 执行上述文法求FIRST/FOLLOW集合、生成分析表的操作。

之后 `parser` 再从文件中读取待分析的token流，开始进行分析匹配。token流文件可以有多行，我们的程序会逐行分析。

`parser` 根据 `grammar` 生成的分析表工作，工作算法大致遵循算法4.1。由于我们加入了 `synch` ，因此与算法4.1的区别在于：遇到 `synch` ，我们执行弹出操作。

Test

测试的用例和结果分别已经输出在文件 `tokens.txt` 和 `process.txt` 中，经过分析可以看出过程的正确性。

Summary

这次实验是我第一次将python代码打包为完整的package来实现算法。在实现过程中，由于Python代码的简洁特性，在实现中我得以将更多的注意集中在LL1文法本身；而打包为完整package，也让我更好地体会到了工程开发的过程。很遗憾，在写完之后经过和老师的沟通，得到了不允许使用Python的回复，我将在之后单独给老师补交一个C++的版本。尽管如此，这次的实验过程依然使我获益匪浅。