THE UNIVERSITY OF SYDNEY

COMP3109 Assignment 2

Parsing LL(1) Grammars (10 marks)

This second assignment is an **individual assignment**, and is due Friday, 6pm, in **Week 8**. Please submit this assignment via eLearning. Your assignment will not be assessed unless the following criteria are met: (1) For each task have at least 15 test cases. (2) The documentation should be done in form of comments. Please provide plenty of them.

In this exercise, you will be implementing a *table-driven predictive* parser[1] for LL(1) grammars, that constructs a table in a pre-processing step using first- and follow sets. After the construction of the table, the input is traversed by reading terminal by terminal from the input, and the parser will cross-check with the means of a stack and a predictive parsing table whether the input is well-formed.

This assignment is split into four tasks. The first task is to compute "first and follow" sets of a given grammar, which are necessary for parsing sentences. The second task is to compute a predictive parsing table based on first- and follow sets. The third task is to implement a table-driven predictive parser which checks whether an input sentence is member of a language. The forth task rewrites an EBNF grammar to a BNF grammar.

We can define a context-free grammar (CFG) as a 4-tuple $\langle T, V, P, S \rangle$, where

- T is a finite set of terminals,
- V is a finite set of non-terminals,
- P is a set of production rules of the form $V \rightarrow (T \cup V)^*$, and
- $S \in V$ is the start symbol for the grammar.

Here is an example of a CFG in BNF:

```
S ::= A B B A
A ::= a
A ::= epsilon
B ::= b
```

where `epsilon` denotes the empty string ε. For sake of simplicity, we further assume that small letters are terminals and capital letters are non-terminals. The alternatives of a non-terminal are spelled out in separate production rules rather splitting alternative productions by a separator symbol. Note that the set of terminals T is $\{a, b\}$, the set of nonterminals V is $\{S, A, B\}$, and the start symbol is S for the above grammar.

[1] Some of you may be familiar with parsing CFGs via a method called *recursive descent* – the table-based method is different.

Sentences of the language which is generated by the grammar above are:

$$bb$$
$$abb$$
$$bba$$
$$abba$$

Your programs for accomplishing the following tasks must run in the ED workspaces, and should be implemented in Python. The symbol for derivation should be `::=`. Note that your program(s) will have to work out the start symbol for the grammar; the LHS of the first grammar rule is the start symbol for the grammar. Note also that the case of each symbol does not define whether or not the symbol is a terminal or non-terminal; your program will have to work out whether each symbol is a terminal or not. You should have one (or more) common files which do all of the "first and follow" logic, and your three front end programs (one for each task) should be calling functions from this common library. For example, our sample implementation consists of four files:

```
bash$ wc -l *.py
120 common.py
 20 question1.py
 20 question2.py
 19 question3.py
179 total
```

Task 1 (3 marks)

For the construction of parsers you need to compute two functions called FIRST and FOLLOW associated with a grammar G. During top-down parsing the two functions choose which production is used for expanding the left-most non-terminal in the sentential form.

FIRST sets

We define FIRST(α) as the set of terminal symbols that are the first symbols in the language of sentential form α. If ε is in the language of α, then ε will be also in FIRST(α). E.g., FIRST(B) is b for our example grammar because $B \to b$, and FIRST(A) is set $\{\varepsilon, a\}$ because A either derives to ε or a.

We define FOLLOW(A) for a non-terminal A to be the set of terminals a that can appear immediately to the right of A, i.e., this is the set of terminal symbols a such that there exists a derivation $S \xrightarrow{*} \alpha A a \beta$ for some α and β.

For parsing we introduce a special symbol called the input end marker denoted by \$. Ths input end marker is used as a stop symbol for the input and needs to be considered in parsing since we cannot really check for ε as an input for the language.

First, we compute FIRST(X) for all grammar symbols $X \in V \cup T$ by applying following rules

1. If $X \in T$, then FIRST$(X) = \{X\}$.
2. If $X \in V$ and $X \to Y_1 \dots Y_k \in P$ for some $k \geq 1$, then place a in FIRST (X) if $a \in$ FIRST(Y_1) or for some i, $a \in$ FIRST(Y_i), and ε is in all of FIRST$(Y_1), \dots,$ FIRST(Y_{i-1}). If ε is in all of FIRST$(Y_1), \dots,$ FIRST(Y_k), place ε in FIRST(X).
3. If $X \in V$, and $X \to \varepsilon$, then place ε in FIRST(X).

until no more terminals or ε can be added to any FIRST set.

Example 1. *For our grammar the FIRST sets are given below:*

Symbol	FIRST
a	$\{a\}$
b	$\{b\}$
A	$\{a, \varepsilon\}$
B	$\{b\}$
S	$\{a, b\}$

We extend the definition of FIRST to arbitrary sentential forms, $\alpha = Y_1 \dots Y_k$ for some $k > 1$, as follows,

$$\text{FIRST}(Y_1 \dots Y_k) = \begin{cases} \text{FIRST}(Y_1), & \text{if } \varepsilon \notin \text{FIRST}(Y_1), \\ \text{FIRST}(Y_1) \setminus \{\varepsilon\} \cup \text{FIRST}(Y_2 \dots Y_k), & \text{otherwise,} \end{cases}$$

and FIRST(ε) is $\{\varepsilon\}$. For example, FIRST$(ABBA)$ is $\{a, b\}$.

FOLLOW sets

Second, we compute FOLLOW(B) for all nonterminals by applying the following rules until nothing can be added to any FOLLOW set:

1. Place \$ in FOLLOW(S) for start symbol S.
2. If there is a production $A \to \alpha B \beta$, then all symbols in FIRST(β) except ε is in FOLLOW(B).
3. If there is a production $A \to \alpha B$ or ε is in FIRST(β) of a production $A \to \alpha B \beta$, then all symbols in FOLLOW(A) are in FOLLOW(B).

Example 2. *For our grammar the* FOLLOW *sets are given below:*

Non-terminal	FOLLOW
S	$\{\$\}$
A	$\{b, \$\}$
B	$\{a, b, \$\}$

Your task is to write a program which implements the "first and follow" algorithm. Your program should take a filename as a command line argument, which will contain a grammar definition in the same BNF format as our example grammars. In this file there will be one rule per line, with production alternatives split over multiple lines. There will be no blank lines in the input file. Each symbol in the production of a rule will be separated by a single space character. The literal string "`epsilon`" will be the symbol representing ε. All other symbols will consist of a single character only (strings of length 1).

Your program should determine the FIRST and FOLLOW sets for all terminals and non-terminals in the provided grammar file, and then output the values for all non-terminals in the format shown below.

```
bash$ cat example.grammar
S ::= A B B A
A ::= a
A ::= epsilon
B ::= b
bash$ ./question1.py example.grammar
First:
  A -> a epsilon
  B -> b
  S -> a b
Follow:
  A -> b $
  B -> a b $
  S -> $
```

Task 2 (2 marks)

Algorithm 1 collects the information from FIRST and FOLLOW into a predictive parsing table $M[A, a]$ where A is a non-terminal, and a is a terminal symbol or the input end marker \$. Based on the predictive table the parsing is performed on the idea that the production rule $A \to \alpha$ is chosen if the next input symbol a is in $\text{FIRST}(\alpha)$. The only problem occurs if ε can be derived from the sequence α. In this case, we choose $A \to \alpha$, if the current symbol is in $\text{FOLLOW}(A)$, or if the input end marker \$ has been reached and \$ is in $\text{FOLLOW}(A)$. If there is no production at entry $M[A, a]$, then we set $M[A, a]$ to **error**.

Algorithm 1 Construction of the Parsing Table

```
for all A → α ∈ P do
  for all a ∈ FIRST(α) do
    let M[A, a] := A → α
  end for
  if ε ∈ FIRST(α) then
    for all b ∈ FOLLOW(A) do
      let M[A, b] := A → α
    end for
    if $ ∈ FOLLOW(A) then
      let M[A, $] := A → α
    end if
  end if
end for
```

Algorithm 1 can be applied to any LL(1) grammar, and produces a single table entry that is either a production or signals an error. It can be shown that if you assign a table entry more than once, the grammar is not LL(1).

Example 3. *The predictive parsing table of our grammar is given below:*

M	a	b	$\$$
S	$S \to ABBA$	$S \to ABBA$	***error***
A	$A \to a$	$A \to \varepsilon$	$A \to \varepsilon$
B	***error***	$B \to b$	***error***

Your task is to write an algorithm that generates a predictive parsing table and reports an error if the grammar is not LL(1). Your program should produce as output one of two things. If the grammar is not LL(1), then it should output "`Grammar is not LL(1)!`". Otherwise, your program should output a readable representation of the parsing table for the grammar. The output format should be lines of the format $R[A, a] = n$ where A is the non-terminal, a is the terminal, and n is the rule number (counting from zero). The order of these lines of output does not matter. An example usage of your program is as follows:

```
bash$ cat isLL1.grammar
S ::= A B B A
A ::= a
A ::= epsilon
B ::= b
bash$ ./question2.py isLL1.grammar
R[A, a] = 1
R[A, b] = 2
R[A, $] = 2
R[S, a] = 0
R[S, b] = 0
R[B, b] = 3
bash$ cat notLL1.grammar
S ::= A B B A
A ::= a
A ::= epsilon
B ::= b
B ::= epsilon
bash$ ./question2.py notLL1.grammar
Grammar is not LL(1)!
```

Task 3 (3 marks)

We construct a non-recursive predictive parser by utilising a stack that contains either non-terminals or terminals. The contents of the stack represents a sequence of non-terminals and terminals α (read from the top of the stack to the bottom) such that $w\alpha$ is a derivable sentential form of the start symbol, i.e,

$$S \xrightarrow{*} w\alpha$$

where $w \in T^*$ is the input that has already been matched so far. Initially, the stack is set to the value $\langle S, \$ \rangle$ where S is the start symbol and $\$$ is the input end marker. This stack configuration denotes the state that we have not consumed any input symbols from the input stream yet.

Algorithm 2 Table-driven Predictive Parser

```
push S$
let a be the first symbol in the input stream
while stack is not empty do
  if X is a non-terminal then
    if M[X, a] = A → α then
      pop
      push α
    else
      report syntax error
    end if
  else
    if X = a then
      if X ≠ $ then
        pop
        let a be the next symbol in the input stream
      end if
    else
      report syntax error
    end if
  end if
end while
if a ≠ $ then
  report syntax error
else
  report sentence is in the language
end if
```

The parser considers the top of the stack symbol X and the current input symbol a. If X is a non-terminal symbol, then X will be replaced by the entry $M[X, a]$ of the predictive table. Note if $M[X, a]$ has an **error** entry, then a syntax error will be reported. Otherwise, X is a terminal symbol and if X matches the current input symbol a, we advance with the next terminal in the input stream and pop the element X from the stack. If X does not match a, we will report a syntax error. The parsing is successful if the stack is empty and we have consumed all symbols in the input stream. The parsing of a table-driven predictive parser is summarised in Algorithm 2. In the algorithm we use a **pop** operation that pops one symbol from the stack. The **push** operation pushes a sequence of terminals

and non-terminals onto the stack whereby the right-most symbol of the sequence is pushed first on the stack. Note that the order is *relevant* otherwise the sequences of the productions will be reversed.

Your task here is to write a predictive table-driven parser that reads in a LL(1) grammar and a sentence and either reports that the sentence is in the language or reports a syntax error. Your program should now take two command line arguments; the name of two files. The first file will be the grammar format as in the previous question. The second file will contain a set of strings, one string per line. Your program should use the parse table constructed in the previous question to parse the strings. For each string in the second file, your program should output either "`accept`" or "`reject`" depending on whether or not the string is accepted by the grammar. If the grammar is not LL(1), then your program should only output "`Grammar is not LL(1)!`".

```
bash$ cat isLL1.input
abba
aba
ab
bb
bba
bbb
chicken
bash$ ./question3.py isLL1.grammar isLL1.input
accept
reject
reject
accept
accept
reject
reject
bash$ ./question3.py notLL1.grammar anything_here
Grammar is not LL(1)!
```

Task 4 (2 marks)

This tasks is to transform a grammar given in EBNF into an equivalent BNF grammar. EBNF and BNF have two main differences (for the purpose of this question). Firstly, EBNF has curly brackets (`{ }`), which are used to indicate zero or more repetitions of the enclosed content. Additionally, EBNF has square brackets (`[ ]`) which indicates zero or one repetitions of the enclosed content.

For example, the following is a grammar in EBNF:

```
S ::= A { B b } e
A ::= a | epsilon
B ::= [ c ] d
```

Its equivalent BNF form is:

```
S ::= A T e
T ::= T B b
T ::= epsilon
A ::= a
A ::= epsilon
B ::= C d
C ::= c
C ::= epsilon
```