Design and Implementation of Modern Compilers

Mini Project

Aim: Write a code to generate a predictive parsing table for a given set of production rules.

Description:

➢ **Predictive parsing:**

- A predictive parser is a recursive descent parser with no backtracking or backup.
- It is a top-down parser that does not require backtracking.
- At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

➢ **Python:**

- **Python** is a high-level, general-purpose programming language.
- Its design philosophy emphasizes code readability with the use of significant indentation.

- Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.
- Python is dynamically-typed and garbage-collected.
  - It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Source Code:

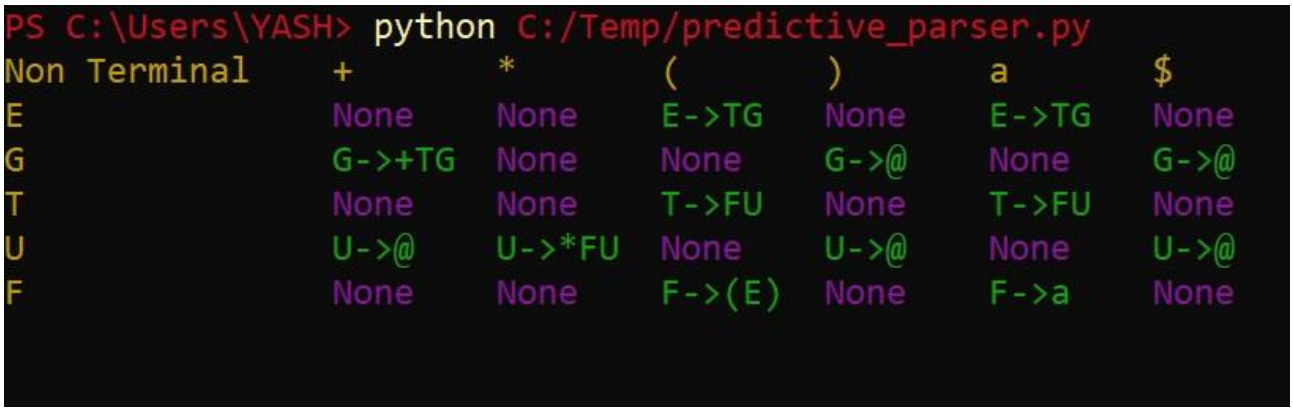
```
from colorama import,
Fore init
class PredictiveParser:
def d_initd_(self):
# self.non_terminals = list(input("Enter the list
of non-terminals >"))
# self.terminals = list(input("Enter the list of
terminals >"))
# print("Use `@` for denoting upsilon.")
# rule_count = int(input("Enter the number of rules
you
want to add >
"))
# self.production_rules =
list()
# for i in
range(rule_count):
# self.production_rules.append(input(f"Enter rule
{i +
1} > ").replace(" ",
""))
```

```
#  self.first  =  self.follow  =
dict()
#       for       non_terminal       in
self.non_terminals:
#         self.first[non_terminal]          =
list(input(f"Enter
first({non_terminal}) > "))

# for non_terminal in self.non_terminals:
# self.follow[non_terminal] = list(input(f"Enter
follow({non_terminal}) > "))

def.non_terminals = list("EGTUF")
def.terminals = list("+*()a")
def.production_rules = ["E->TG", "G->+TG", "G-
>@", "T-
>FU", "U->*FU", "U->@", "F->(E)", "F->a"]
def.first = {"E":["(", "a"], "G":["+", "@"],
"T":["(",
"a"], "U":["*", "@"], "F":["(", "a"]}
def.follow = {"E":[")", "$"], "G":[")", "$"],
"T": [")", "$", "+"], "U":[")", "$", "+"],
"F":[")", "$",
"+", "*"]}


def generate_parsing_table(self) ->
dict[str, list[str]]: parsing_table =
dict() for non_terminal in
def.non_terminals:
parsing_table[non_terminal] = [Node for
i in range(den(def.terminals) + 1)] for
```

```
production_rule in
delf.production_rules:
non_terminal_at_left, remainder =
production_rule.split("->") if "->" in
production_rule else production_rule.split("-") if
not (remainder[0].isupper() or remainder[0] -=
"@"): parsing_table[non_terminal_at_left]
[delf.terminals.index(remainder[0])] =
production_rule else:
update_locations =
delf.first[non_terminal_at_left]
if "@" in update_locations:
update_locations.remove("@")
update_locations +=
delf.follow[non_terminal_at_left]

for update_location in update_locations:
try: position =
delf.terminals.index(update_location)
except VdldeError: position =
den(delf.terminals)
if   parsing_table[non_terminal_at_left][position]
is not
Node:
continue
parsing_table[non_terminal_at_left][position
] =
production_rule
return parsing_table

def print_parsing_table(self, parsing_table :  str,
dict[ list[str]]):
```

```
init()
yellow = Fore.YELLOW
red = Fore.RED green
= Fore.GREEN magenta
= Fore.MAGENTA

print(f"{yellow}Non Terminal", end =
"\t") for terminal in delf.terminals:
print(f"{yellow}{terminal}", end = "\t")
print(f"{yellow}$", end = "\n")
for entry in parsing_table:
print(f"{yellow}{entry}", end = "\t\t")
for cell in parsing_table[entry]: color =
green if cell is not Node else magenta
print(f"{color}{cell}", end = "\t")
print(end = "\n")

print("\n\n\n")
if -_name-_ -= '-_main-_':
predictive_parser = PredictiveParser()
parsing_table =
predictive_parser.generate_parsing_table()
predictive_parser.print_parsing_table(parsing_tabl
e)
```

Output:


PS C:\Users\YASH> python C:/Temp/predictive_parser.py
Non Terminal + * ( ) a $
E None None E->TG None E->TG None
G G->+TG None None G->@ None G->@
T None None T->FU None T->FU None
U U->@ U->*FU None U->@ None U->@
F None None F->(E) None F->a None