COMPILER DESIGN PROJECT DOCUMENTATION

Grammar Definition

Grammar can do addition, subtraction, multiplication and division. In grammar, attention has been paid to the priority of the process. n denotes a number.

E -> E + T | E - T | T
T -> T * F | T / F | F
F -> (E) | n

Step-by-Step Process Explanation

1. Lexical Analysis (Lexical Phase)

Purpose of the Code: The code performs the Lexical Analysis stage, which is the first step for an LL(1) parser. At this stage, the expression is divided into tokens. Unnecessary characters, such as spaces, are discarded. An error is thrown for undefined characters.

Code sections:
Token Specifications: token_specifications list includes each type of token and its corresponding regular expressions.
"SKIP" represents gaps, and such tokens are omitted during analysis.
The expression "MISMATCH" is used for undefined characters and works in case of error.

Regex Pairing: All token definitions are combined into a single regular expression. token_regex is a merged regular expression, so that a single lookup is performed on the expression.

Token Processing Cycle: Scans the expression given with regular expression and returns each match. Unnecessary characters, such as spaces, are omitted. If an undefined character is found, a SyntaxError is thrown.

```
def lexical_analysis(expression):
    # Tokens
    token_specifications = [
        ("NUMBER", r'\d+'),
        ("PLUS", r'\+'),
        ("MINUS", r'-'),
        ("TIMES", r'\*'),
        ("DIVIDE", r'\/'),
        ("LPAREN", r'\('),
        ("RPAREN", r'\)'),
        ("SKIP", r'\s+'),
        ("MISMATCH", r'.'),
    ]
    token_regex = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in token_specifications)
    tokens = []

    for match in re.finditer(token_regex, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == "SKIP":
          continue
        elif kind == "MISMATCH":
          raise SyntaxError(f"Geçersiz karakter: {value}")
        else:
          tokens.append((kind, value))
    return tokens
```

Input: 3 + 5
Output: Tokens of the mathematical expression: [('NUMBER', '3'), ('PLUS', '+'), ('NUMBER', '5')]

2. Removing Left Recursion

Purpose of the Code: The code checks if a given grammar has a left recursive and, if it does, removes the left recursive.

Code Sections:

Segregation of Left Recursive and Non-Left Recursive Productions: Left recursive productions are added to the left_recursive list. Non-Left recursive productions are added to the non_left_recursive list.

Creating a New Non-Terminal: For the removal of the left recursion, an auxiliary non-terminal is created for each left recursive non-terminal.

Writing New Production Rules: An auxiliary non-terminal is added to the left non-recursive productions. The auxiliary non-terminal is derived from left-recursive productions. In addition, the production of a terminating epsilon (ε) is added.

```
def remove_left_recursion(grammar):
  new_grammar = {}
```

```
for non_terminal, productions in grammar.items():
    left_recursive = []
    non_left_recursive = []

    for production in productions:
        if production[0] == non_terminal:
            left_recursive.append(production[1:])
        else:
            non_left_recursive.append(production)

    if left_recursive:
        new_non_terminal = f"{non_terminal}PRIME"
        new_grammar[non_terminal] = [[*list(p), new_non_terminal] for p in non_left_recursive]
        new_grammar[new_non_terminal] = [[*list(p), new_non_terminal] for p in left_recursive] + [['ε']]
    else:
        new_grammar[non_terminal] = [[*list(p)] for p in productions]

return new_grammar
```

Input:

```
        E -> E + T | E - T | T
        T -> T * F | T / F | F
        F -> (E) | n
```

Output: Left Recursion Removed.

```
{
        'E': [['T', 'EPRIME']],
        'EPRIME': [['+', 'T', 'EPRIME'], ['-', 'T', 'EPRIME'], ['ε']],
        'T': [['F', 'TPRIME']],
        'TPRIME': [['*', 'F', 'TPRIME'], ['/', 'F', 'TPRIME'], ['ε']],
        'F': [['(', 'E', ')'], ['n']]
}
```

3. Left Factoring Removed

Purpose of the Code: This code performs the left-factoring operation on a grammar. Left factoring rearranges productions that share a common initial part.

Code Sections:

Grouping of Common Initial Parts: In this step, the production rules for each non-terminal are grouped according to their common starting characters.

Left Factorization Control: If any initial character has more than one production, left factoring is performed.
Yeni Non-Terminal Oluşturma: A helper non-terminal is created for productions with a common start.

Writing New Productions: A new production is added for the non-terminal. The common initial part is preserved. The non-common parts are transported to the productions of the auxiliary non-terminal.

```
def left_factoring(grammar):
    new_grammar = {}
    for non_terminal, productions in grammar.items():
        prefixes = {}
        for prod in productions:
            prefix = prod[0]
            if prefix not in prefixes:
                prefixes[prefix] = []
            prefixes[prefix].append(prod)

        if any(len(prods) > 1 for prods in prefixes.values()):
            for prefix, prods in prefixes.items():
                if len(prods) > 1:
                    new_non_terminal = non_terminal + "'"
                    new_grammar[non_terminal] = [prefix + new_non_terminal]
                    new_grammar[new_non_terminal] = [prod[len(prefix):] for prod in prods if prod[len(prefix):]] + ['ε']
                    break
        else:
            new_grammar[non_terminal] = productions
    return new_grammar
```

Input: Grammar with no left recursion.
Output: Left Factoring Removed.

```
{
        'E': [['T', 'EPRIME']],
        'EPRIME': [['+', 'T', 'EPRIME'], ['-', 'T', 'EPRIME'], ['ε']],
        'T': [['F', 'TPRIME']],
        'TPRIME': [['*', 'F', 'TPRIME'], ['/', 'F', 'TPRIME'], ['ε']],
        'F': [['(', 'E', ')'], ['n']]
}
```

4. First Set Hesaplama

Purpose of the Code: At this stage, we need to calculate the first set of each non-terminal for the parse table.

Code Sections:

Initialization of the First Set: An empty set is created for each non-terminal. The visited set holds the symbols that have already been calculated. This is used to avoid recalculations.

First Calculation Function (first_of): This helper function calculates the First set of a given symbol.

- **Terminal Control:** If the symbol is a terminal, the First set consists of only that terminal.
- **Visit Control:** The process is not repeated for a previously calculated non-terminal.
- **Processing of Productions:** Each production of the non-terminal is processed in turn. If it contains production ε, the ε is added directly to the First cluster. For each character, First is calculated and added to the existing set.

Calculation of First Sets of Non-Terminals: First sets are calculated for all non-terminals in grammar.

```
def calculate_first(grammar):
    first = {non_terminal: set() for non_terminal in grammar}
    visited = set()

    def first_of(symbol):
        if symbol not in grammar:
            return {symbol}

        if symbol in visited:
            return first[symbol]

        visited.add(symbol)

        for production in grammar[symbol]:
            if production == "ε":
                first[symbol].add("ε")
            else:
                for char in production:
                    first[symbol] |= first_of(char)
                    if "ε" not in first_of(char):
                        break
        return first[symbol]

    for non_terminal in grammar:
        first_of(non_terminal)

    return first
```

Input: Grammar with no left recursion and left factored.
Output: First set:

```
{
        'E': {'(', 'n'},
        'EPRIME': {'-', '+', 'ε'},
        'T': {'(', 'n'},
        'TPRIME': {'/', '*', 'ε'},
        'F': {'(', 'n'}
}
```

5. Follow Set Calculation

Purpose of the Code: At this stage, we need to calculate the follow set of each non-terminal for the parse table.

Code Sections:
Initial Settings: Empty Follow sets are initialized for all non-terminals. '$' is added to the initial non-terminal.

Change Detection: A while loop updates the Follow sets as they change. The cycle stops when there is no change.

Processing of Productions: Each non-terminal and its productions are processed sequentially.
Case 1: If there is a symbol after the symbol we are looking for:

- **If the next symbol is non-terminal:** All terminal symbols in the First set of the next symbol are added to the Follow set of the symbol (except ε). If the next symbol can derive ε, the first set of this symbol is added to the Follow set. If this becomes a chain and the last symbol is reached, and the last symbol has ε in the first set, the Follow set of left hand side is also added to the Follow set of the symbol.
- **If the next symbol is terminal:** The terminal symbol is added directly to the Follow set.

Case 2: If the symbol is at the end of production or the next symbol can derive ε, the Follow set of the left hand side is added to the Follow set of the symbol.

Change Check: If there is a change in a Follow set, the calculation is done again.

```
def compute_follow_sets(grammar, first_sets):
    follow_sets = {non_terminal: set() for non_terminal in grammar}
    start_symbol = next(iter(grammar))
    follow_sets[start_symbol].add('$')

    changed = True

    while changed:
        changed = False

        for lhs, productions in grammar.items():
            for production in productions:
                for i, symbol in enumerate(production):
                    if symbol in grammar:
                        follow_before = follow_sets[symbol].copy()

                        if i + 1 < len(production):
                            next_symbol = production[i + 1]
                            if next_symbol in grammar:
                                follow_sets[symbol].update(first_sets[next_symbol] - {'ε'})

                                if 'ε' in first_sets[next_symbol]:
                                    j = i + 1
                                    while j < len(production):
                                        if 'ε' in first_sets.get(production[j-1], set()):
                                            next_symbol = production[j]
                                            follow_sets[symbol].update(first_sets.get(next_symbol, set()) - {'ε'})
                                            j += 1
                                        else:
                                            break

                            else:  # Terminal
                                follow_sets[symbol].add(next_symbol)

                        if i + 1 == len(production) or 'ε' in first_sets.get(production[i + 1], set()):
                            follow_sets[symbol].update(follow_sets[lhs])
                        if follow_before != follow_sets[symbol]:
                            changed = True

    return follow_sets
```

Input: Grammar with no left recursion and left factoring, first set.
Output: Follow set:

```
{
        'E': {')', '$'},
        'EPRIME': {')', '$'},
        'T': {')', '-', '$', '+'},
        'TPRIME': {')', '-', '$', '+'},
        'F': {')', '$', '*', '/', '-', '+'}
}
```

6. Computed Sets Hesaplama:

Purpose of the Code: At this stage, we must calculate the computed sets table that we will use for the parse table.

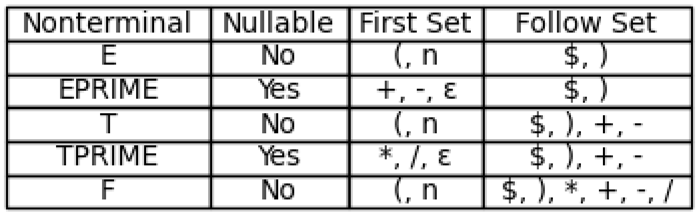
Code Sections:
Non-Terminal Loop: All non-terminals in the grammar are visited.

Nullable Control: If the first set of the non-terminal has ε, it is assigned as Nullable, otherwise No.

Preparation of Line Information: The sequential First and Follow sets are added to the table, separated by commas.

```
def computed_sets(grammar, first, follow):
    data = []
    for non_terminal in grammar.keys():
        nullable = "Yes" if "ε" in first.get(non_terminal, set()) else "No"

        row = {
            "Nonterminal": non_terminal,
            "Nullable": nullable,
            "First Set": ", ".join(sorted(first.get(non_terminal, set()))),
            "Follow Set": ", ".join(sorted(follow.get(non_terminal, set()))),
        }
        data.append(row)
    return pd.DataFrame(data)
```

Output:



Nonterminal	Nullable	First Set	Follow Set
E	No	(, n	$,)
EPRIME	Yes	+, -, ε	$,)
T	No	(, n	$,), +, -
TPRIME	Yes	*, /, ε	$,), +, -
F	No	(, n	$,), *, +, -, /

7. Create Parse Table

Purpose of the Code: Thanks to the parse table, we will be able to do the parsing process. We can understand which transition to apply by looking at the input value we see in the parsing process from the parse table.

Code Sections:

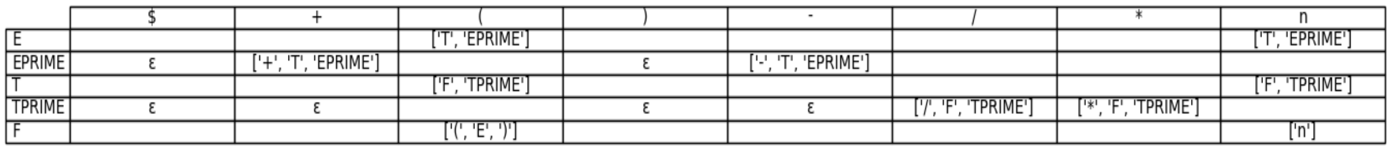
Parse Table Initialization: All terminal and non-terminal symbols to be included in the table are determined.

Update based on the First Set: By navigating through the DataFrame, the First Set is checked for each non-terminal. If a terminal is in the First Set, the production rule for that terminal is added to the table.

Update Based on Follow Set: If a non-terminal is nullable (Nullable == "Yes"), the table is updated for each terminal in the Follow Set.

```
def create_parse_table(grammar, computed_sets, first_sets):
    terminals = {')', '(', '+', '-', '*', '/', '$'}
    non_terminals = grammar.keys()

    parse_table = {non_terminal: {terminal: None for terminal in terminals} for non_terminal in non_terminals}

    for row in computed_sets.itertuples():
        non_terminal = row.Nonterminal
        first_set = row._3.split(", ") if row._3 else []
        follow_set = row._4.split(", ") if row._4 else []
        for terminal in first_set:
            if terminal != "ε":
                for production in grammar[non_terminal]:
                    if production[0] in non_terminals:
                        if terminal in first_sets[production[0]]:
                            parse_table[non_terminal][terminal] = production
                    else:
                        parse_table[non_terminal][production[0]] = production

        if row.Nullable == "Yes":
            for terminal in follow_set:
                if parse_table[non_terminal][terminal] is None:
                    parse_table[non_terminal][terminal] = "ε"

    for non_terminal in parse_table:
        if "ε" in parse_table[non_terminal]:
            del parse_table[non_terminal]["ε"]

    return parse_table
```

Output:



	$	+	(	)	-	/	*	n
E			['T', 'EPRIME']					['T', 'EPRIME']
EPRIME	ε	['+', 'T', 'EPRIME']		ε	['-', 'T', 'EPRIME']			
T			['F', 'TPRIME']					['F', 'TPRIME']
TPRIME	ε	ε		ε	ε	['/', 'F', 'TPRIME']	['*', 'F', 'TPRIME']	
F			['(', 'E', ')']					['n']

8. Parsing

Purpose of the Code: This code is a parsing function that resolves an array of inputs using an LL(1) parse table. The input symbols are analyzed according to the rules in the parse table and it is determined whether it is a valid language or not.

Code Sections:
Initial State: The stack is initialized with $ (terminator) and start_symbol.
Stack and Input Processing: The cycle continues until the stack is empty.

Stack Checks: If the symbol at the top of the stack is $ and the current token is $, the entry is accepted.

Terminal and Non-Terminal Processing: If the symbol at the top of the stack is a terminal, it must match the first token of the input. If it matches, it is removed from the stack and the entry is advanced. If there is no match, the error is thrown. If the symbol at the top of the stack is a non-terminal, the corresponding rule is applied in the Parse Table. If there is no valid rule, the error is thrown.

Epsilon Productions: If ε (null production) is to be applied, nothing is added to the stack.

Transition Records: Applied rules and matches are added to the list of transitions.

```python
def parse_input(parse_table, input_tokens, start_symbol):
    stack = ['$', start_symbol]
    transitions = []

    while stack:
        top_of_stack = stack.pop()
        current_token = input_tokens[0]

        print(f"Stack top: {top_of_stack}, Current token: {current_token}")

        if top_of_stack == '$' and current_token == '$':
            print('Accept')
            break

        if top_of_stack not in parse_table:
            if top_of_stack == current_token:
                input_tokens.pop(0)
            else:
                raise ValueError(f"Unexpected token: {current_token}. Expected: {top_of_stack}")
        else:
            if current_token in parse_table[top_of_stack]:
                rule = parse_table[top_of_stack][current_token]
                if rule and rule != 'ε':
                    transitions.append(f"Apply rule: {top_of_stack} -> {' '.join(rule)}")
                    stack.extend(reversed(rule))
                else:
                    transitions.append(f"Apply rule: {top_of_stack} -> ε")
            else:
                raise ValueError(f"Unexpected token: {current_token}. Expected: {top_of_stack}")
```

```
return transitions
```

Output:

```
Stack top: E, Current token: n
Stack top: T, Current token: n
Stack top: F, Current token: n
Stack top: n, Current token: n
Stack top: TPRIME, Current token: +
Stack top: EPRIME, Current token: +
Stack top: +, Current token: +
Stack top: T, Current token: n
Stack top: F, Current token: n
Stack top: n, Current token: n
Stack top: TPRIME, Current token: *
Stack top: *, Current token: *
Stack top: F, Current token: $
Stack top: TPRIME, Current token: $
Stack top: EPRIME, Current token: $
Stack top: $, Current token: $
Accept

Apply rule: E -> T EPRIME
Apply rule: T -> F TPRIME
Apply rule: F -> n
Apply rule: TPRIME -> ε
Apply rule: EPRIME -> + T EPRIME
Apply rule: T -> F TPRIME
Apply rule: F -> n
Apply rule: TPRIME -> * F TPRIME
Apply rule: F -> ε
Apply rule: TPRIME -> ε
Apply rule: EPRIME -> ε
```

9. Drawing Parse Tree

Purpose of the Code: Bu kod, verilen geçişlere göre bir **parse ağacı** oluşturur ve bunu Graphviz ile çizer.

Code Sections:

Graphviz Launch: A Digraph (directional graph) object is created.

Düğüm ve Kenar Yönetimi: Node information is stored in a list (nodes). Each node is represented by a dictionary.

Creating the Root Node: The first node starts with start_symbol and is added to the nodes list.

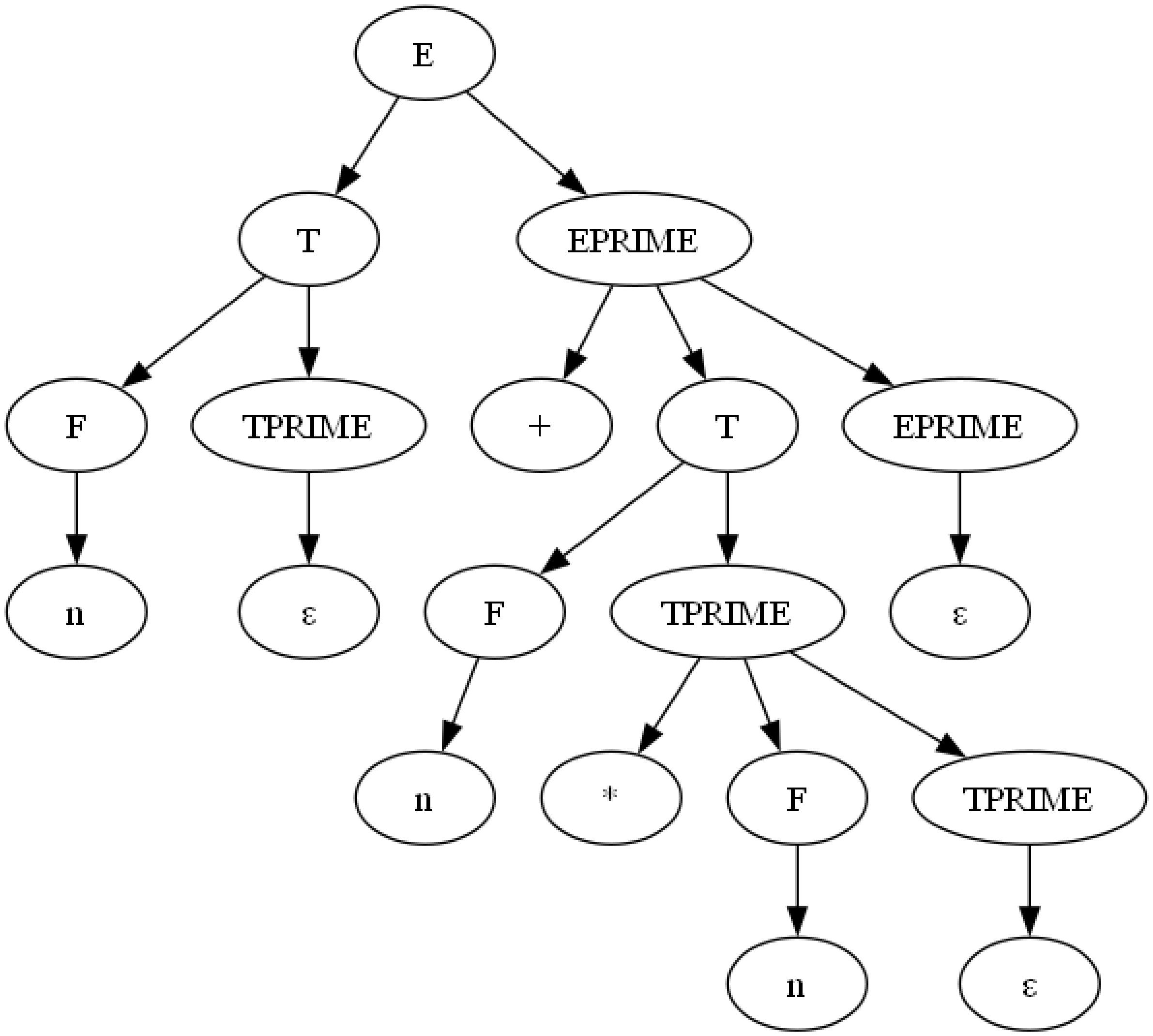
Processing of Transitions: With each pass (parent_symbol, child_symbols), the corresponding node is expanded using a DFS scan.

Node Expansion (DFS): dfs_find_and_expand finds the appropriate node and adds the child nodes.

```
def draw_parse_tree(transitions, start_symbol):
    tree = Digraph(format="png", graph_attr={"rankdir": "TB"})

    nodes = []
    edges = {}

    root = {"id": 0, "label": start_symbol, "children": []}
    nodes.append(root)
    tree.node(str(root["id"]), label=start_symbol)

    def dfs_find_and_expand(symbol, children):
        for node in nodes:
            if node["label"] == symbol and not node["children"]:
                for child_label in children:
                    child_id = len(nodes)
                    child_node = {"id": child_id, "label": child_label, "children": []}
                    nodes.append(child_node)
                    node["children"].append(child_node)
                    tree.node(str(child_id), label=child_label)
                    tree.edge(str(node["id"]), str(child_id))
                return
        raise ValueError(f"Eşleşen bir düğüm bulunamadı: {symbol}")

    for transition in transitions:
        parent_symbol, child_symbols = transition
        dfs_find_and_expand(parent_symbol, child_symbols)
```

```
tree.render("C:/Intel/parse_tree", format="png", view=True)
```

Input: Transitions
Output:


E
T
EPRIME
F
TPRIME
+
T
EPRIME
n
ε
F
TPRIME
ε
n
*
F
TPRIME
n
ε