S -> a A | b
A -> a S | b

Program -> Statement ; Program
Program -> epsilon

Statement -> DeclarationStatement
Statement -> AssignStatement
Statement -> IOStatement
Statement -> IfStatement
Statement -> WhileStatement
Statement -> MatchStatement
Statement -> CompoundStatement

DeclarationStatement -> Identifier : Type
DeclarationList -> DeclarationStatement
DeclarationList -> DeclarationStatement , DeclarationList

Type -> TypeValue
Type -> ArrayDecl

TypeValue -> nothing
TypeValue -> boolean
TypeValue -> character
TypeValue -> integer
TypeValue -> text
TypeValue -> structure
TypeValue -> list
TypeValue -> dictionary

ArrayDecl -> TypeValue [Size]

AssignStatement -> Identifier = Expression

Expression -> Term ExpressionRest
ExpressionRest -> + Term ExpressionRest
ExpressionRest -> epsilon

Term -> Factor TermRest
TermRest -> * Factor TermRest
TermRest -> epsilon

Factor -> (Expression)
Factor -> Identifier

IOStatement -> input (String)
IOStatement -> output (Identifier)

CompoundStatement -> { StatementList }

IfStatement -> if (Condition) { StatementList }
IfStatement -> if (Condition) { StatementList } else { StatementList }

MatchStatement -> match (Expression) { CaseList }

WhileStatement -> while (Condition) { StatementList }

Condition -> Expression RelationalOperator Expression

RelationalOperator -> ==
RelationalOperator -> <
RelationalOperator -> <=
RelationalOperator -> =
RelationalOperator -> !=
RelationalOperator -> >=
RelationalOperator -> >

CaseList -> CaseStatement
CaseList -> CaseStatement ; CaseList
CaseStatement -> Expression : Statement

DeclarationStatement -> Identifier : Type

Identifier -> Letter RestOfIdentifier
RestOfIdentifier -> epsilon
RestOfIdentifier -> Letter RestOfIdentifier
RestOfIdentifier -> Digit RestOfIdentifier
RestOfIdentifier -> _ RestOfIdentifier

Size -> IntConstant

Letter -> A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | _

Digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

NonZeroDigit -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

MaybeSign -> +
MaybeSign -> -
MaybeSign -> epsilon

IntConstant -> MaybeSign PositiveNumber
IntConstant -> 0
PositiveNumber -> NonZeroDigit TheRestOfIntConstant
TheRestOfIntConstant -> epsilon
TheRestOfIntConstant -> Digit TheRestOfIntConstant

String -> Char String
String -> epsilon

Char -> Letter
Char -> Digit

```
class Grammar:
    def __init__(self, filename):
        self.nonterminals = set()
        self.terminals = set()
        self.productions = {}
        self.start_symbol = None
        self.read_grammar(filename)

    def read_grammar(self, filename):
        with open(filename, 'r') as file:
            lines = file.readlines()

        for line in lines:
            line = line.strip()

            if not line:
                continue

            lhs, rhs = line.split('->')
            lhs = lhs.strip()
            rhs = [prod.strip() for prod in rhs.split('|')]

            # Add nonterminal (lhs)
            self.nonterminals.add(lhs)

            if self.start_symbol is None:
                self.start_symbol = lhs  # First nonterminal is the start symbol

            # Add productions for this nonterminal
            if lhs not in self.productions:
                self.productions[lhs] = []

            for prod in rhs:
                self.productions[lhs].append(prod.split())

                # Add symbols to terminals/nonterminals
                for symbol in prod.split():
                    if symbol.isupper():
                        self.nonterminals.add(symbol)
                    else:
                        self.terminals.add(symbol)

    def print_grammar(self):
        print("Nonterminals:", self.nonterminals)
        print("Terminals:", self.terminals)

        print("Productions:")

        for nonterminal, prods in self.productions.items():
            for prod in prods:
                print(f"{nonterminal} -> {' '.join(prod)}")

    def is_cfg(self):
        # Simple check for CFG (productions must have a single nonterminal on the left-hand side)
        for lhs in self.productions:
```

```
    if len(lhs) != 1 or not lhs.isupper():
        return False

return True
```

```
class RecursiveDescentParser:
    def __init__(self, grammar):
        self.grammar = grammar
        self.current_token = None
        self.index = 0
        self.input_string = []
        self.parse_tree = []
        self.node_counter = 0

    def parse(self, input_string):
        self.input_string = input_string.split()
        self.index = 0
        self.parse_tree = []

        # Start the parsing process with the start symbol
        success = self.parse_nonterminal(self.grammar.start_symbol, None)

        if success and self.index == len(self.input_string):
            print("Parsing successful.")
            self.print_parse_tree()
        else:
            print("Parsing failed.")

    def parse_nonterminal(self, nonterminal, parent):
        # Store parent-child relationship in parse tree
        node_id = self.node_counter
        self.node_counter += 1
        self.parse_tree.append((node_id, nonterminal, parent))

        # Try all productions for this nonterminal
        for production in self.grammar.productions[nonterminal]:
            saved_index = self.index

            if all(self.parse_symbol(symbol, node_id) for symbol in production):
                return True

            self.index = saved_index

        return False

    def parse_symbol(self, symbol, parent):
        if symbol in self.grammar.nonterminals:
            return self.parse_nonterminal(symbol, parent)
        elif self.index < len(self.input_string) and symbol == self.input_string[self.index]:
            # Match terminal
            node_id = self.node_counter
            self.node_counter += 1
            self.parse_tree.append((node_id, symbol, parent))
            self.index += 1
            return True

        return False

    def print_parse_tree(self):
        print("Parse Tree (ID, Symbol, Parent):")
```

```
for node_id, symbol, parent in self.parse_tree:
    print(f"Node ID: {node_id}, Symbol: {symbol}, Parent ID: {parent}")
```