Final Project Documentation: Balanced Parentheses Language

Part 1 – Design a Formal Language

Language Name: Balanced Parentheses

Introduction and Intention

The purpose of this language is to model and recognize balanced sequences of parentheses. This seems like a small problem, however I learned the importance of syntactic correctness required by compilers, interpreters and parsers in programming languages, arithmetic expressions, and document structures where proper nesting is required. And here I try to have this language implemented as straightforwardly as possible.

This language consists of strings composed only of the characters (and). A string is valid in this language if every opening parenthesis (has a corresponding closing parenthesis) and the pairs are properly nested.

Alphabet

Σ = { (,) }

Semantics

A string belongs to this language **L** if: For every prefix of the string, the number of closing parentheses) does not exceed the number of opening parentheses (. At the end of the string, the total number of (equals the total number of).

Valid: "", "()", "(())", "(()(()))"
Invalid: ")(", "(()", "())(", `"((())"

Purpose

This language is context-free language—a class of languages recognized by pushdown automata. It is often used to validate the structure of expressions before they are further processed. For instance: Programming: Matching brackets in source code. Mathematics: Ensuring that nested expressions are correctly grouped. Markup languages: Ensuring that tags are properly closed.

Part 2 – Grammar

This language is defined by a **context-free grammar (CFG)**:

Non-terminal symbols: S
Terminal symbols: (,)
Start symbol: S

Production Rules:

S → (S)
S → S S
S → ε

The first rule allows a single balanced pair with possible nesting inside.
The second rule allows sequences of valid expressions.
The third rule allows termination and represents the empty string.
This grammar generates all strings of balanced parentheses and only those strings.

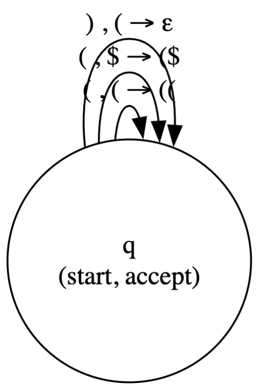
Part 3 – Automaton

We construct a Pushdown Automaton (PDA) that recognizes the balanced parentheses language. The PDA uses a stack to track unmatched opening parentheses.

PDA:

States: { q }
Input Alphabet: { (,) }
Stack Alphabet: { (, $ } ($ is a bottom-of-stack marker)
Start State: q
Start Stack Symbol: $
Accept States: { q }

Transitions:
On (, push (onto the stack.
On), pop (from the stack.
Accept if the input is consumed and the stack contains only the marker $.



Part 4 – Data Structure

We represent the PDA using a structured text file (paren.pda) and an in-memory C++ data structure.

File Format:

```
Line 1: q                # states
Line 2: ( )              # input symbols
Line 3: ( $              # stack symbols
Line 4: q                # start state
Line 5: $                # start stack symbol
Line 6: q                # accept state
Line 7+: transitions (from input pop push to)

q ( ( (( q
q ( $ ($ q
q ) ( ε  q
```

Data Structure Implementation in C++:

```
struct PDA {
vector<string> states;
vector<char>   input_alphabet;
vector<char>   stack_alphabet;
string         start_state;
char           stack_start;
unordered_set<string> accept_states;
map<tuple<string, char, char>, pair<string, string>> transitions;
};
```

Each transition is indexed by (current_state, input_symbol, pop_stack).
The transition result is (next_state, push_stack).

Part 5 – Testing

Function Specification

```
bool accept(const PDA & A, const string& w, vector<StackTransition>& path);
```

A: the PDA
w: input string
path: list of state/input/pop/push transitions taken

PDA load_pda(const string& filename). Reads a PDA from a file and populates the in-memory structure.

Sample Test Case:

```
string w = "(()())";
vector<StackTransition> path;
bool result = accept(pda, w, path);
```

Output:

```
accept
q --( / $ --> q [push=($]
q --( / ( --> q [push=((]
q --) / ( --> q [push=ε]
q --( / ( --> q [push=((]
q --) / ( --> q [push=ε]
q --) / ( --> q [push=ε]
```

Download and Run the Program

The full implementation is available in the GitHub repository

1: Clone the Repository

```
git clone <repository_url>
cd <repository_folder>
```

2: Build the Program

Ensure you have a C++ compiler (e.g., g++). Then compile using: g++ -std=c++17 src/project.cpp -o paren_pda.

3: Run

```
./paren_pda paren.pda "(()())"
```

Expected Output

```
accept
q --( / $ --> q [push=($]
q --( / ( --> q [push=((]
q --) / ( --> q [push=ε]
```