Compiler Frontend Project Documentation

Overview

This project was made as part of the final project for CSCI 2115 - Discrete Math in Computer Science. It implements the front end of a basic compiler with JSON as the input language.
It comprises of two major components:
1. Scanner: Performing Lexical Analysis of input JSON into a tokenised input stream using a DFA-based approach
2. Parser: Taking the tokenised input stream and performing syntactic and semantic analysis and providing an output parse tree using a recursive descent parser.

Scanner

The scanner tokenizes input JSON strings into a sequence of tokens using a Deterministic Finite Automaton (DFA) based approach and processes the input character by character (excluding whitespace).

It converts JSON elements like strings, numbers, boolean, structural symbols and null into tokens and stores them in an output file, while handling any errors gracefully.

Usage

How to use the scanner is detailed in the scanner readme file, but to sum up:

1. Place your JSON input in a file (e.g., test01.txt by default).
2. Run the scanner file (ensure the input file name is correct).
3. The tokens will be saved to an output file [inputfilename]_output.txt

Key classes and methods:

Token: Represents JSON Tokens
JSONScannerDFA: The main scanner class with DFA transitions and tokenization logic.
ScannerError: Handles invalid character errors gracefully.

Parser

The parser reads in the scanner output as an input stream, validates the tokenized input against JSON grammar rules, builds parse trees, and performs semantic analysis to detect errors such as duplicate keys or inconsistent list types.

It validates JSON grammar using recursive descent parsing, detects semantic errors like duplicate keys, lists with inconsistent element types, reserved keywords being used as dict keys, etc and generates readable parse trees as the output (given valid input).

Usage

How to use the parser is detailed in the parser readme file, but to sum up:

1. Put the tokens output from the scanner (or manually create input token streams) into the tokenstreams folder.
2. Run the parser.py file
3. Parse trees (or errors) will be saved to the parsetrees folder, or otherwise throw an error.

Key Classes and methods:

ParseTreeNode: Node class for constructing parse trees.
Parser: Implements recursive descent parsing with semantic checks
SemanticError: Handles semantic rule violations

Example Workflow:

1. Input JSON:

{"name": "schlawg", "age": 30, "is_student": false}

2. Scanner Output (test01_output.txt):

<LEFT_BRACE, {>
<STRING, name>
<COLON, :>
<STRING, schlawg>
<COMMA, ,>
<STRING, age>
<COLON, :>
<NUMBER, 30>
<COMMA, ,>
<STRING, is_student>

<COLON, :>
<BOOLEAN, false>
<RIGHT_BRACE, }>

3. Parser Output (error file):

```
object
  pair
    key
      name
    value
      schlawg
  pair
    key
      age
    value
      30
  pair
    key
      is_student
    value
      false
```