Documentation

Source code: https://github.com/caprapaul/flcd/tree/lab_05

Grammar

The grammar is read from a json file.

To check if it is context free we check if the left side of all production rules contains exactly one nonterminal.

g1

```
{
  "NonTerminals": ["S", "A", "B", "C"],
  "Terminals": ["(", ")", "+", "*", "int", "E"],
  "StartingSymbol": "S",
  "ProductionRules": {
    "S": [
      "A B"
    ],
    "A": [
      "( S )",
      "int C"
    ],
    "B": [
      "+ S",
      "E"
    ],
    "C": [
      "* A",
      "E"
    ]
  }
}
```

g2

```
{
    "NonTerminals": [
        "program",
        "statement_list",
        "statement",
        "declare_statement",
        "expression_statement",
        "expression",
        "block_expression",
        "block_content",
        "assign_expression",
        "if_expression",
```

```
        "while_expression",
        "print_expression",
        "read_expression",
        "index_expression",
        "array_expression",
        "array_elements",
        "unary_operator_expression",
        "unary_operator",
        "binary_operator_expression",
        "binary_operator",
        "arithmetic_operator",
        "boolean_operator",
        "comparison_operator",
        "group_expression",
        "type",
        "basic_type",
        "array_type"
    ],
    "Terminals": [
        ";",
        "let",
        ":",
        "{",
        "}",
        "=",
        "if",
        "(",
        ")",
        "while",
        "print",
        "read",
        "[",
        "]",
        ",",
        "-",
        "!",
        "+",
        "*",
        "/",
        "%",
        "&&",
        "||",
        "==",
        "≠",
        "<",
        ">",
        "≤",
        "≥",
        "i32",
        "char",
        "bool",
        "E"
    ],
    "StartingSymbol": "program",
    "ProductionRules": {
        "program": ["statement_list"],
        "statement_list": ["statement", "statement statement_list"],
        "statement": ["declare_statement", "expression_statement", ";"],
```

```
        "declare_statement": ["let identifier : type ;"],
        "expression_statement": ["expression ;"],
        "expression": [
            "const",
            "identifier",
            "block_expression",
            "assign_expression",
            "if_expression",
            "while_expression",
            "print_expression",
            "identifier",
            "read_expression",
            "unary_operator_expression",
            "binary_operator_expression",
            "group_expression",
            "index_expression",
            "identifier"
        ],
        "block_expression": ["{ block_content }"],
        "block_content": ["statement_list", "statement_list expression"],
        "assign_expression": ["identifier = expression"],
        "if_expression": ["if ( expression ) block_expression"],
        "while_expression": ["while ( expression ) block_expression"],
        "print_expression": ["print ( expression )"],
        "read_expression": ["read"],
        "index_expression": ["expression [ expression ]"],
        "array_expression": ["[ array_elements ]"],
        "array_elements": ["expression", "expression , array_elements"],
        "unary_operator_expression": ["unary_operator expression"],
        "unary_operator": ["-", "!"],
        "binary_operator_expression": ["expression binary_operator expression"],
        "binary_operator": [
            "arithmetic_operator",
            "boolean_operator",
            "comparison_operator"
        ],
        "arithmetic_operator": ["+", "-", "*", "/", "%"],
        "boolean_operator": ["&&", "||"],
        "comparison_operator": ["==", "≠", "<", ">", "⩽", "⩾"],
        "group_expression": ["( expression )"],
        "type": ["basic_type", "array_type"],
        "basic_type": ["i32", "char", "bool"],
        "array_type": ["[ type ; const ]"]
    }
}
```