# Custom Mini Programming Language Processor

## Phase 2: Syntax Analysis (Parser)

CMPE 458

Prepared By Group #2

Hendrix Gryspeerdt – 20337154

Monica Saad - 20348798

Ryan Silverberg - 20342023

Simon John – 20348233

March 8th, 2025

## Overview of Modifications

The project was refactored so that all code is in the folder “my-mini-compiler”. “my-mini-compiler” contains two folders, one to retain the code from phase1 separate from phase 2 under “phase1-w25”, and the second containing code brought forward from the lexer in phase 1 and all new work from phase 2 under “phase2-w25.” Lots of new code was added in phase2-w25 and the new files are detailed below.

The lexer in phase 1 was implemented in a way such that it was not modular and could not be combined with the future stages of the compiler. To address this, a significant refactoring of lexer.h and lexer.c took place to make this integration possible. Some modification had to be made to the underlying functionality of the lexer when it came to specifying the token types of operators and the “read” keyword.

New header files added to “phase2-w25/include”:

- grammar.h - contains data structure for representing context-free grammars and declares functions to check grammar rules for direct and indirect left-recursion.
- parse_tokens.h - contains full definition of all ParseTokens that can exist within the context free grammar. It also contains all definitions of ParseErrorTypes. It also contains a structural definition of what is present in a ParseTreeNode. It also contains functions that convert the relevant parse tokens into their string counterpart.
- parser.h - Contains relevant function definitions for generating a parse tree from a token stream using recursive descent parsing and additional functions for freeing and printing parse trees.
- ast_node.h - Contains relevant definitions for the tokens present within the Abstract Syntax Tree along with error definitions. It also includes a structural definition of an AST node present in the AST itself.

New source files added to the “phase2-w25/src”:

- parser/grammar.c – implementation of the two functions declared in grammar.h
- parser/parser.c – implementation of recursive descent parsing and ParseTreeNode printing and freeing functions.
- main.c – contains the grammar data structure for the programming language. Grammar validation logic to ensure that it can be parsed using recursive descent parsing. Loading input from file, instantiation of the lexer and tokenization of input, parsing input and creating parse tree. Also, a function to print the head of a parse tree node and show error messages.

## List of Changes

### Lexer

- Made original lexical tokens more descriptive and more specific rather than general. For example, TOKEN_KEYWORD is removed and replaced with a token value for each keyword allowed in the language.

- Added functions to allow for tokens to be mapped to the correct TokenType upon tokenization.
- Reordered lexical tokens to allow for 1-1 mapping to the parse tokens for ease of programming.

### Parser

- Implemented full functionality using recursive descent parsing.
- Defined a large array of grammar rules to allow for modular design and easy to modify/extend the programming language.

## Description of new Tokens

### Lexical Tokens

Added:

TOKEN_INTEGER_CONST, TOKEN_FLOAT_CONST, TOKEN_STRING_CONST, TOKEN_INT_KEYWORD, TOKEN_FLOAT_KEYWORD, TOKEN_STRING_KEYWORD, TOKEN_PRINT_KEYWORD, TOKEN_READ_KEYWORD, TOKEN_IF_KEYWORD, TOKEN_THEN_KEYWORD, TOKEN_ELSE_KEYWORD, TOKEN_WHILE_KEYWORD, TOKEN_REPEAT_KEYWORD, TOKEN_UNTIL_KEYWORD, TOKEN_FACTORIAL_KEYWORD, TOKEN_SEMICOLON, TOKEN_LEFT_BRACE, TOKEN_RIGHT_BRACE, TOKEN_LEFT_PAREN, TOKEN_RIGHT_PAREN, TOKEN_EOF, TOKEN_EQUAL_EQUAL, TOKEN_BANG_EQUAL, TOKEN_GREATER_THAN_EQUAL, TOKEN_LESS_THAN_EQUAL, TOKEN_LESS_THAN_LESS_THAN, TOKEN_GREATER_THAN_GREATER_THAN, TOKEN_AMPERSAND_AMPERSAND, TOKEN_PIPE_PIPE, TOKEN_EQUAL, TOKEN_PLUS, TOKEN_MINUS, TOKEN_STAR, TOKEN_FORWARD_SLASH, TOKEN_PERCENT, TOKEN_TILDE, TOKEN_AMPERSAND, TOKEN_PIPE, TOKEN_CARET, TOKEN_BANG, TOKEN_LESS_THAN, TOKEN_GREATER_THAN, TOKEN_ERROR

Removed:

TOKEN_IDENTIFIER, TOKEN_INTEGER, TOKEN_FLOAT, TOKEN_KEYWORD, TOKEN_OPERATOR, TOKEN_IDENTIFIER, TOKEN_STRING_LITERAL, TOKEN_PUNCTUATOR

### Parsing Tokens

Added:

Terminals:

PT_TOKEN_ERROR, PT_IDENTIFIER, PT_INTEGER_CONST, PT_FLOAT_CONST, PT_STRING_CONST, PT_INT_KEYWORD, PT_FLOAT_KEYWORD, PT_STRING_KEYWORD, PT_PRINT_KEYWORD, PT_READ_KEYWORD, PT_IF_KEYWORD, PT_THEN_KEYWORD, PT_ELSE_KEYWORD, PT_WHILE_KEYWORD, PT_REPEAT_KEYWORD, PT_UNTIL_KEYWORD, PT_FACTORIAL_KEYWORD, PT_SEMICOLON, PT_LEFT_BRACE, PT_RIGHT_BRACE, PT_LEFT_PAREN, PT_RIGHT_PAREN, PT_EOF, PT_EQUAL_EQUAL, PT_BANG_EQUAL, PT_GREATER_THAN_EQUAL, PT_LESS_THAN_EQUAL, PT_LESS_THAN_LESS_THAN, PT_GREATER_THAN_GREATER_THAN, PT_AMPERSAND_AMPERSAND, PT_PIPE_PIPE, PT_EQUAL, PT_PLUS, PT_MINUS, PT_STAR, PT_FORWARD_SLASH, PT_PERCENT, PT_TILDE, PT_AMPERSAND, PT_PIPE, PT_CARET, PT_BANG, PT_LESS_THAN, PT_GREATER_THAN

Non-Terminals: (PT_PROGRAM is the start symbol)

PT_PROGRAM, PT_SCOPE, PT_STATEMENT_LIST, PT_STATEMENT, PT_EMPTY_STATEMENT, PT_DECLARATION, PT_EXPRESSION_STATEMENT, PT_EXPRESSION_EVAL, PT_PRINT_STATEMENT, PT_BLOCK, PT_CONDITIONAL, PT_WHILE_LOOP, PT_REPEAT_UNTIL_LOOP, PT_OPTIONAL_ELSE_BLOCK, PT_STATEMENT_END, PT_TYPE_KEYWORD, PT_EXPRESSION, PT_BLOCK_BEGIN, PT_BLOCK_END

Expressions:

PT_ASSIGNMENTEX_R12, PT_ASSIGNMENT_REST, PT_OREX_L11, PT_ANDEX_L10, PT_BITOREX_L9, PT_BITXOREX_L8, PT_BITANDEX_L7, PT_RELATIONEX_L6, PT_SHIFTEX_L5, PT_SUMEX_L4, PT_PRODUCTEX_L3, PT_UNARYPREFIXEX_R2, PT_FACTOR, PT_FACTORIAL_CALL

Operator Tokens:

PT_ASSIGNMENT_OPERATOR, PT_OR_OPERATOR, PT_AND_OPERATOR, PT_BITOR_OPERATOR, PT_BITXOR_OPERATOR, PT_BITAND_OPERATOR, PT_RELATIONAL_OPERATOR, PT_SHIFT_OPERATOR, PT_SUM_OPERATOR, PT_PRODUCT_OPERATOR, PT_UNARY_PREFIX_OPERATOR

Operation Nodes:

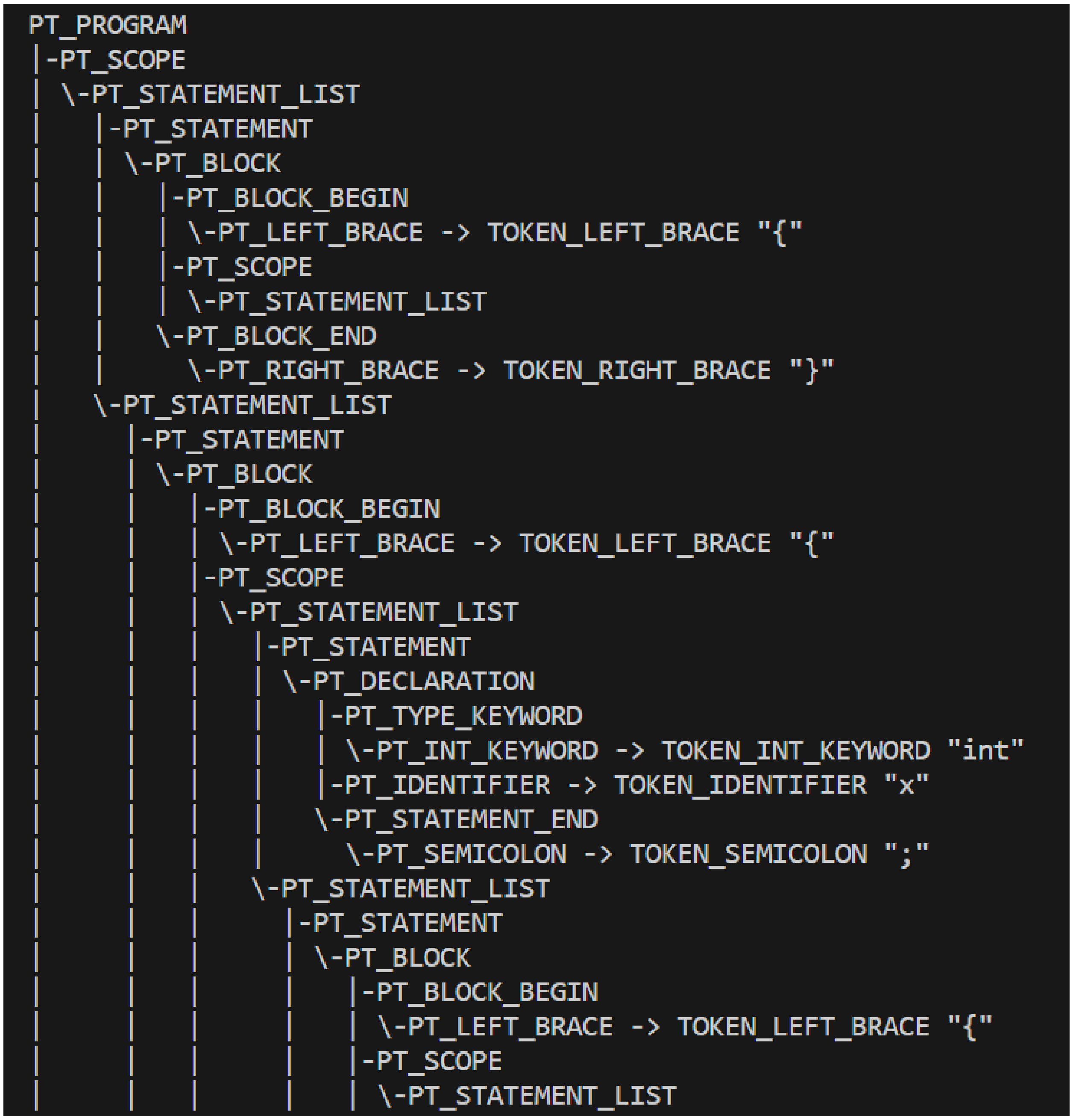
PT_ASSIGN_EQUAL, PT_LOGICAL_OR, PT_LOGICAL_AND, PT_BITWISE_OR, PT_BITWISE_XOR, PT_BITWISE_AND, PT_COMPARE_EQUAL, PT_COMPARE_NOT_EQUAL, PT_COMPARE_LESS_EQUAL, PT_COMPARE_LESS, PT_COMPARE_GREATER_EQUAL, PT_COMPARE_GREATER, PT_SHIFT_LEFT, PT_SHIFT_RIGHT, PT_ADD, PT_SUBTRACT, PT_MULTIPLY, PT_DIVIDE, PT_MODULO, PT_BITWISE_NOT, PT_LOGICAL_NOT, PT_NEGATE,

Used to simulate null termination of ParseToken array.

PT_NULL

## Changes to Code Generation Output

The parse tree is now being printed to the console containing all relevant grammar rules and what tokens the non-terminals derive from.



```
PT_PROGRAM
|-PT_SCOPE
| \-PT_STATEMENT_LIST
|   |-PT_STATEMENT
|   | \-PT_BLOCK
|   |   |-PT_BLOCK_BEGIN
|   |   | \-PT_LEFT_BRACE -> TOKEN_LEFT_BRACE "{"
|   |   |-PT_SCOPE
|   |   | \-PT_STATEMENT_LIST
|   |   \-PT_BLOCK_END
|   |     \-PT_RIGHT_BRACE -> TOKEN_RIGHT_BRACE "}"
|   \-PT_STATEMENT_LIST
|     |-PT_STATEMENT
|     | \-PT_BLOCK
|     |   |-PT_BLOCK_BEGIN
|     |   | \-PT_LEFT_BRACE -> TOKEN_LEFT_BRACE "{"
|     |   |-PT_SCOPE
|     |   | \-PT_STATEMENT_LIST
|     |   |   |-PT_STATEMENT
|     |   |   | \-PT_DECLARATION
|     |   |   |   |-PT_TYPE_KEYWORD
|     |   |   |   | \-PT_INT_KEYWORD -> TOKEN_INT_KEYWORD "int"
|     |   |   |   |-PT_IDENTIFIER -> TOKEN_IDENTIFIER "x"
|     |   |   |   \-PT_STATEMENT_END
|     |   |   |     \-PT_SEMICOLON -> TOKEN_SEMICOLON ";"
|     |   |   \-PT_STATEMENT_LIST
|     |   |     |-PT_STATEMENT
|     |   |     | \-PT_BLOCK
|     |   |     |   |-PT_BLOCK_BEGIN
|     |   |     |   | \-PT_LEFT_BRACE -> TOKEN_LEFT_BRACE "{"
|     |   |     |   |-PT_SCOPE
|     |   |     |   | \-PT_STATEMENT_LIST
```

See the list of sample input files in the directory "my-mini-compiler/phase2-w25/test."

## Details of New Error Handling

There are currently 5 different types of errors that are handled by the parser as outline in the ParseErrorType enumeration (PARSE_ERROR_NONE is used to indicate that no error has occurred). The list below details the different errors and how they arise.

1. PARSE_ERROR_WRONG_TOKEN
2. PARSE_ERROR_NO_RULE_MATCHES

3. PARSE_ERROR_CHILD_ERROR
4. PARSE_ERROR_PREVIOUS_TOKEN_FAILED_TO_PARSE
5. PARSE_ERROR_MULTIPLE_LEFT_RECURSIVE_RULES

PARSE_ERROR_WRONG_TOKEN will occur when trying to parse a single terminal token and the token in the input does not match. This error cannot occur in the first child of a non-terminal token because if the first child of the production rule could not start with the input token, then an alternate production rule would have been tried, and if no production rule could start with the input token the non-terminal would have the error of PARSE_ERROR_NO_RULE_MATCHES.

PARSE_ERROR_NO_RULE_MATCHES occurs in a non-terminal when the first available token cannot be a prefix of any of its production rules.

PARSE_ERROR_CHILD_ERROR occurs in a non-terminal token when one of its children failed to parse and had an error type other than PARSE_ERROR_NONE. When a child fails to parse, position in the input token stream is not advanced.

PARSE_ERROR_PREVIOUS_TOKEN_FAILED_TO_PARSE occurs in any non-terminal or terminal token in a production rule when parsing stops due to a previous token resulting in an error.

PARSE_ERROR_MULTIPLE_LEFT_RECURSIVE_RULES is an error identified during parsing, however, it is not a "parse error" per se. This error would occur if the grammar rule for a particular token has associated with it two or more left-recursive production rules prior to identifying a viable production rule that starts with the input token.