Custom Mini Programming Language Processor

Phase 3: Semantic Analysis

CMPE 458

Prepared By Group #2

Hendrix Gryspeerdt – 20337154

Monica Saad - 20348798

Ryan Silverberg - 20342023

Simon John – 20348233

March 30th, 2025

Overview of Modifications

List of Changes

Tokens

tokens.h, parse_tokens.h, and ast_types.h now only contain enums, #define macros and function declarations for enum conversion to string which are implemented in the src/enum_to_string directry.

Lexer

There were some significant structural changes to the lexer, however there were no core functional changes made to tokenization.

1. Use of global variables were replaced with the use of a Lexer struct to allow for access to the internal state of the lexer when printing compiler messages involving the line_start_positions.
2. print_token and print_token_compiler_message were moved from lexer.h into main.c since they don't form part of the lexer itself but instead form part of the error reporting scheme of the overall compiler.
3. is_keyword and record_if_newline functions are now qualified as "static inline" since they are small functions that are only used locally within lexer.c.

Parser

Some issues with the memory allocation were identified upon further review of parse_cfg_recursive_descent_parse_tree and ParseTreeNode_free and have since been corrected. As a result of this fix, some minor interface changes to the parsing function were made which are described in parser.h and reflected in parser.c. The only other changes to parsing were in introducing explicit functions for non-left-recursive parsing and default error recovery.

1. Moved definitions of ParseTreeNode and ASTNode into parser.h.
2. Removed the ParseTreeRoot_print function and created a generalized tree printing function (see tree.h and tree.c) that can work for a wider variety of tree structures.
3. Fixed memory allocation errors in parse_cfg_recursive_descent_parse_tree by ensuring only the children of a node are allocated dynamically.
4. Created two new functions, initialize_children_by_rule and default_error_recovery for parsing and error recovery to simplify the implementation of parse_cfg_recursive_descent_parse_tree.
5. ParseTreeNode struct now includes a pointer to the production rule that specifies the types of its children.

Semantic Analyzer

This phase of the compiler began the introduction of the semantic analyzer. There are two major components to our implementation of semantic analysis, (1) conversion from Parse Tree to Abstract Syntax Tree and (2) Semantic Validation of the Abstract Syntax Tree by type checking and variable scope checking.

Abstract Syntax Tree Creation

Creation of the Abstract Syntax tree was closely related to the program_grammar and ProductionRule definitions in grammar.h and is implemented by constructing an ASTNode from a given ParseTreeNode. The key operations of conversion from ParseTreeNode to ASTNode lie in removing unnecessary tokens and nesting from the parse tree and converting the ParseToken types into ASTNodeType. These type conversions are specified in the ProductionRules of program_grammar (see grammar.h) and the tree simplifications are specified using the three special ASTNodeType values AST_SKIP, AST_FROM_CHILDREN, and AST_FROM_PROMOTION along with promote_index and promotion_alternate_if_AST_NULL from the production rule. The grammar of the abstract syntax tree is specified below using the modified production rules.

Special Symbols:

- the ^ symbol following a token indicates that that token is promoted in-place of the left-hand side of the production rule.
- the @ symbol following a token indicates that its children are promoted to replace it.

Abstract Syntax Tree Grammar:

```
Program -> Scope
Scope -> StatementList@
StatementList -> Statement StatementList@ | epsilon

Statement -> Scope^
           | Declaration^
           | Expression^
           | Print^
           | Read^
           | Conditional^
           | WhileLoop^
           | RepeatUntilLoop^

Declaration -> Declaration_Type Identifier
Print -> Expression
Read -> Expression
Conditional -> Expression Scope Scope
WhileLoop -> Expression Scope
RepeatUntilLoop -> Scope Expression
```

```
Expression -> Operation^
            | int_const^
            | float_const^
            | string_const^
            | identifier^

```

```
Operation -> AssignEqual^
           | LogicalOr^
           | LogicalAnd^
           | BitwiseOr^
           | BitwiseXor^
           | BitwiseAnd^
           | CompareEqual^
           | CompareNotEqual^
           | CompareLessEqual^
           | CompareLessThan^
           | CompareGreaterEqual^
           | CompareGreaterThan^
           | ShiftLeft^
           | ShiftRight^
           | Add^
           | Subtract^
           | Multiply^
           | Divide^
           | Modulo^
           | BitwiseNot^
           | LogicalNot^
           | Negate^
           | Factorial^

```

```

AssignEqual        -> Expression Expression
LogicalOr          -> Expression Expression
LogicalAnd         -> Expression Expression
BitwiseOr          -> Expression Expression
BitwiseXor         -> Expression Expression
BitwiseAnd         -> Expression Expression
CompareEqual       -> Expression Expression
CompareNotEqual    -> Expression Expression
CompareLessEqual   -> Expression Expression
CompareLessThan    -> Expression Expression
CompareGreaterEqual -> Expression Expression
CompareGreaterThan -> Expression Expression
ShiftLeft          -> Expression Expression
ShiftRight         -> Expression Expression
Add                -> Expression Expression
Subtract           -> Expression Expression
Multiply           -> Expression Expression
Divide             -> Expression Expression
Modulo             -> Expression Expression
BitwiseNot         -> Expression
LogicalNot         -> Expression
Negate             -> Expression
Factorial          -> Expression
```

The ^ special symbol above is specified using the promote_index of a ProductionRule and optionally combined with the special ASTNodeType AST_FROM_PROMOTION and/or promotion_alternate_if_AST_NULL. The @ special symbol is implemented using the special ASTNodeType AST_FROM_CHILDREN. The ability to ignore nodes of the parse tree is implemented using the ASTNodeType AST_SKIP. This conversion is ultimately implemented in the function ASTNode_from_ParseTreeNode that makes use of ASTNode_get_promo to determine which nodes are promoted.

Semantic Validation of the Abstract Syntax Tree

The Semantic Validation was done by creating multiple functions to handle each part of the grammar. Semantic rules were defined along with their logic within these functions. The semantic validation stems from the ProcessProgram function which takes in the head of the AST and traverses through it recursively analyzing each AST node based on its corresponding AST type declared. ProcessProgram initializes the scope tracking, call ProcessScope, and cleans up scope tracking when the validation is complete. ProcessScope enters a new scope, calls ProcessScopeChild for each of its children, then exits a scope.

ProcessScopeChild will call the corresponding Process function on a given node. The only functions which ProcessScopeChild will call are ProcessScope, ProcessConditional, ProcessLoop, ProcessExpression, ProcessDeclaration, and ProcessIO. ProcessScope functions as previously discussed.

ProcessConditional confirms that the operation that the conditional statement is reliant upon returns an integer then processes the two scopes associated with the then and the else part of the conditional statement using ProcessScope.

ProcessLoop will process the scope of the loop, be it a while loop or a repeat-until loop and will check that the outcome of the expression does not return a string or null AST node. To check the outcome of the expression ProcessLoop uses ProcessExpression, similarly ProcessLoop uses ProcessScope for the scope in the loops.

ProcessExpression handles operators, constants, and variables. If the given AST node is of the type of some operator, then ProcessOperation is called and the type associated with the operation is returned as the type of the expression. If the AST node is a constant that the type associated with the code is returned. If the node is an identifier and if the identifier exists in the current scope, then the type associated with the identifier is returned, if the identifier does not exist in the current scope, then an error is returned.

ProcessOperation handles assignments, binary operations, and unary operations. For assignments, it verifies that the left-hand side is an identifier and not another assignment. It then evaluates both sides using ProcessExpression, converts variable types to their literal equivalents, and ensures type compatibility between the assignment target and value. Errors are reported if types are incompatible or undefined. For binary operations the function calls ProcessOperator which evaluates both operands and verifies type compatibility. For unary operations the function calls ProcessUnaryOperator which evaluates the single operand and return its type.

ProcessOperator uses ProcessExpression to evaluate the left and right sides of the expression, verifies compatibility and returns errors if needed.

Much like ProcessOperator, ProcessUnaryOperator uses ProcessExpression, verifies the type, and returns errors if necessary.

ProcessDeclaration handles variable declarations by first ensuring the node has exactly two children. It extracts the identifier and its type, then checks for redeclaration conflicts by searching the symbol table for matching identifiers that have conflicting scopes. If a redeclaration is detected, an error is recorded. If no conflicts exist, a new symbol table entry is created containing the variable's name, scope, and type.

ProcessIO validates print and read statements. It first confirms that these operations contain exactly one expression to operate on. It then processes this expression using ProcessExpression to ensure it's valid and of an appropriate type.

Verifying that identifiers are not redeclared in a way that is not allowed requires keeping track of the scope in which variables are declared. Scopes are represented by a string of numbers which represent the number of scopes deep the identifier is. The entire program is in scope 0, if a scope is entered then another number is added, this is best shown with an example. In the below figure x is in scope 0, a is in scope 0.0.0, and s is in scope 0.1.0.

```
int x;

{
    {
    int a;
    }
}
{
    {
        float s;
    }
    {
        int s; ?? valid redeclaration since scope does not conflict with previous identifier named 's'
        {
            string s; ?? this is an invalid redeclaration because of scope conflicts
        }
    }
}
```

A stack is used to represent the current scope, when s is declared the stack is holding [0, 1, 0]. When the current scope is exited, the top number is popped off and the stack is now holding [0, 1]. An array holds the highest number used at each nesting level, after entering the scope in which s was declared the first time the array holds [1, 2, 1], these are the next numbers to use for each nesting level. When the program enters a scope on the third level again it can reference the array and will know to add a 1 to the stack rather than a 0, this means that when s is erroneously redeclared the program is in scope 0.1.1.0. Variable a is declared in scope 0.0.0, at this point the array containing the next numbers to use contains [1, 1, 1]. When scope 0.0.0 is exited the array remains the same because if a new scope is entered then the array needs to be referenced. When scope 0.0 is exited the array can change. Since if a scope is entered it will start with 0.1 any further nesting levels can be reset to zero. To reiterate in another way, when scope 0.0 is exited, the array element representing one level down from 0.0 (the third element) can be reset to zero, it is this functionality that allows the first declaration of s to be in scope 0.1.0 and not 0.1.1.

main.c

Significant modifications were made here and are detailed in the list below:

1. grammar was renamed to program_grammar and was moved to grammar.h
2. grammar validation was moved to a function that is implemented in grammar.c and exported via grammar.h
3. All token-specific printing functions have been moved here (print_token, ParseTreeNode_print_head, ASTNode_print_head, print_token_compiler_message, and report_syntax_errors).
4. Debug flags have been introduced to conditionally print grammar validation/analysis, the token stream, the parse tree, the abstract syntax tree, and/or other messages from semantic analysis.
5. Any compiler errors are printed to stderr using print_token_compiler_message.
6. Both stages of semantic analysis (abstract syntax tree creation and validation) now occur following parse tree generation.

Description of new Tokens

Lexical Tokens

No new lexical tokens were added

Parsing Tokens

No new parsing tokens were added

Abstract Syntax Tree Tokens

The types/tokens introduced for the abstract syntax tree are specified in ast_types.h and are listed below:

```c
// node types for AST (Abstract Syntax Tree)
typedef enum _ASTNodeType {
    // used for null-terminated arrays of ASTNodeType
    AST_NULL,
    // Terminal nodes that have tokens associated with them
    AST_IDENTIFIER,
    AST_INTEGER,
    AST_FLOAT,
    AST_STRING,
    // Terminal Nodes that don't have tokens associated with them.
    AST_INT_TYPE,
    AST_FLOAT_TYPE,
    AST_STRING_TYPE,
    // special node types for grammar rules
    AST_SKIP,
    AST_FROM_CHILDREN,
    AST_FROM_PROMOTION,
    // Non-terminal nodes
    AST_PROGRAM,
    AST_SCOPE, // StatementList@ (variable number of children)
    AST_DECLARATION, // TYPE IDENTIFIER
    AST_PRINT, // Operation
    AST_READ,  // Operation
    AST_CODITIONAL, // Operation SCOPE SCOPE
    AST_WHILE_LOOP, // Operation SCOPE
    AST_REPEAT_UNTIL_LOOP, // Operation SCOPE
    AST_EXPRESSION, // Operation
```

```c
    AST_EXPRESSION, // Operation
    // Binary Operations
    AST_ASSIGN_EQUAL,
    AST_LOGICAL_OR,
    AST_LOGICAL_AND,
    AST_BITWISE_OR,
    AST_BITWISE_XOR,
    AST_BITWISE_AND,
    AST_COMPARE_EQUAL,
    AST_COMPARE_NOT_EQUAL,
    AST_COMPARE_LESS_EQUAL,
    AST_COMPARE_LESS_THAN,
    AST_COMPARE_GREATER_EQUAL,
    AST_COMPARE_GREATER_THAN,
    AST_SHIFT_LEFT,
    AST_SHIFT_RIGHT,
    AST_ADD,
    AST_SUBTRACT,
    AST_MULTIPLY,
    AST_DIVIDE,
    AST_MODULO,
    // Unary Operations
    AST_BITWISE_NOT,
    AST_LOGICAL_NOT,
    AST_NEGATE,
    AST_FACTORIAL,
} ASTNodeType;
```

AST Error Tokens

```c
const char *ASTNodeType_to_string(ASTNodeType t);

// More errors should be determined and added by semantic analysis.
typedef enum _ASTErrorType {
    AST_ERROR_NONE,
    AST_ERROR_MISSING_TOKEN,
    AST_ERROR_TOKEN_ERROR,
    AST_ERROR_CHILD_ERROR,
    AST_ERROR_UNSPECIFIED_PRODUCTION_RULE,
    AST_ERROR_EXPECTED_IDENTIFIER,
    AST_ERROR_EXPECTED_ASSIGNMENT,
    AST_ERROR_INVALID_CONDITIONAL,
    AST_ERROR_EXPECTED_PROMOTION,
    AST_ERROR_REDECLARATION_VAR,
    AST_ERROR_UNDEFINED_ASSIGNMENT,
    AST_ERROR_UNDECLARED_VAR,
    AST_ERROR_INCOMPATIBLE_TYPES, // need to add to error -> string
    AST_ERROR_DIVISION_BY_ZERO
} ASTErrorType;
```

Changes to Code Generation Output

The output depends on the debug flags that are set (see image below and line 123 of main.c):

```c
// Debugging flags
You, 2 minutes ago | 2 authors (Hendrix Gryspeerdt and one other)
struct debug_flags {
    bool grammar_check;
    bool grammar_check_verbose;
    bool show_input;
    bool print_tokens;
    bool print_parse_tree;
    bool print_abstract_syntax_tree;
    bool print_semantic_analysis;
    bool print_symbol_table;
} const DEBUG = {
    .grammar_check = true,
    .grammar_check_verbose = false,
    .show_input = true,
    .print_tokens = false,
    .print_parse_tree = false,
    .print_abstract_syntax_tree = true,
    .print_semantic_analysis = true,
    .print_symbol_table = true
};
```

Example output of printing the parse tree and abstract syntax tree with DEBUG.show_input, DEBUG.print_parse_tree, and DEBUG.print_abstract_syntax_tree set to true on input file “phase3-

w25/test/SimpleTest1.cisc” are shown in the following figures. As you can see in those figures, even for a very simple input, the parse tree is very verbose, whereas the abstract syntax tree is much more concise.

phase3-w25 > test > output1.txt

```
Processing input:
```
int x;
x = 42;

```
Parse Tree:
PT_PROGRAM
|-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_EXPRESSION_STATEMENT
|     |   |-PT_EXPRESSION
|     |   | \-PT_ASSIGNMENTEX_R12
|     |   |   |-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_IDENTIFIER -> TOKEN_IDENTIFIER "x"
|     |   |   \-PT_ASSIGNMENT_REST
```

```
|     |   |   \-PT_ASSIGNMENT_REST
|     |   |     |-PT_ASSIGNMENT_OPERATOR
|     |   |     | \-PT_ASSIGN_EQUAL
|     |   |     |   \-PT_EQUAL -> TOKEN_EQUAL "="
|     |   |     \-PT_ASSIGNMENTEX_R12
|     |   |       |-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_INTEGER_CONST -> TOKEN_INTEGER_CONST "42"
|     |   |       \-PT_ASSIGNMENT_REST
|     |   \-PT_STATEMENT_END
|     |     \-PT_SEMICOLON -> TOKEN_SEMICOLON ";"
|     \-PT_STATEMENT_LIST
\-PT_EOF -> TOKEN_EOF ""

Abstract Syntax Tree:
AST_PROGRAM
\-AST_SCOPE
  |-AST_DECLARATION
  | |-AST_INT_TYPE
  | \-AST_IDENTIFIER -> TOKEN_IDENTIFIER "x"
  \-AST_EXPRESSION
    \-AST_ASSIGN_EQUAL
      |-AST_IDENTIFIER -> TOKEN_IDENTIFIER "x"
      \-AST_INTEGER -> TOKEN_INTEGER_CONST "42"
```

Details of New Error Handling

New error handling is described in main.c by the functions print_token_compiler_message, report_syntax_errors, and ProcessProgram which is the semantic phase entry point, and returns an array of semantic errors. Those functions are included below. print_token_compiler_message is virtually identical to what it was previously but now it includes arguments for the output stream and lexer. report_syntax_errors uses print_token_compiler_message to print an error message indicated what kind of syntax was expected in the event of the wrong token.

Semantic errors are stored globally in an array called semanticErrors, which is initialized with the InitializeErrors function. Each ASTNode has an error field which gets assigned one of the following error codes: AST_ERROR_UNDECLARED_VAR, AST_ERROR_REDECLARATION_VAR, AST_ERROR_INCOMPATIBLE_TYPES, AST_ERROR_UNDEFINED_ASSIGNMENT, or AST_ERROR_INVALID_CONDITIONAL. When an error is detected, the node's error field is set to the appropriate error code and the node is pushed onto semanticErrors and an informative message is printed.

Lexer Errors

```c
void print_token_compiler_message(FILE *const stream, const Lexer *const l, const char *input_file_path, const Token *const token, const char *const error_message)
{
    const int line_start_pos = *(int *)array_get(l->line_start_positions, token->position.line - 1);
    const char *const line_end = strchr(l->input_string + line_start_pos, '\n');
    const int line_length = line_end == NULL ? (int)strlen(l->input_string + line_start_pos) : line_end - (l->input_string + line_start_pos);
    // tildes is supposed to be as long as the longest token lexeme so that it can always be chopped to the right length.
    static const char *const tildes = "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~";
    fprintf(stream,
        "%s:%d:%d: %s\n"
        "%.*s\n"
        "%*s%.*s\n",
        input_file_path, token->position.line, token->position.col_start, error_message,
        line_length, l->input_string + line_start_pos,
        token->position.col_start, "^", token->position.col_end - token->position.col_start, tildes);
}
```

Example print_token_compiler_message:

```
<input-file-name>.cisc:3:5: error: invalid character
int ?;
    ^
```

Syntax Errors

```c
// Enhanced syntax error reporting function using new print function
void report_syntax_errors(FILE *const stream, const Lexer *const l, const ParseTreeNode *const node, const char *const filepath) {
    // print error message if the node has an error and it has a token that was not already reported as an error by the lexer
    switch (node->error) {
        case PARSE_ERROR_NONE:
        case PARSE_ERROR_PREVIOUS_TOKEN_FAILED_TO_PARSE:
            break;
        case PARSE_ERROR_CHILD_ERROR:
            for (const ParseTreeNode *child = node->children; child != node->children + node->count; ++child) {
                report_syntax_errors(stream, l, child, filepath);
            }
            break;
        case PARSE_ERROR_NO_RULE_MATCHES:
        case PARSE_ERROR_WRONG_TOKEN:
            if (node->token) {
                const unsigned int MESSAGE_SIZE = 100;
                char *message = malloc(MESSAGE_SIZE);
                snprintf(message, MESSAGE_SIZE, "error: expected a %s", ParseToken_to_string(node->type));
                print_token_compiler_message(stream, l, filepath, node->token, message);
                free(message);
            }
            break;
    }
}
```

Example report_syntax_errors compiler message:

```
<input-file-name>.cisc:3:5: error: expected a PT_IDENTIFIER
int ?;
    ^
```

The syntax errors are identified in the function parse_cfg_recursive_descent_parse_tree. PARSE_ERROR_NONE indicates that there was no error and that parsing was successful. PARSE_ERROR_NO_RULE_MATCHES occurs in the event of parsing a token where non of its production rules may start at the current position in the token stream. PARSE_ERROR_WRONG_TOKEN occurs in the event of parsing a terminal token which does not match the start of the input token stream. PARSE_ERROR_PREVIOUS_TOKEN_FAILED_TO_PARSE is set during default_error_recovery for the remaining tokens of a production rule after an error has occurred. PARSE_ERROR_CHILD_ERROR is set at a node when one of its children incurred an error type other than PARSE_ERROR_NONE. As you can see from the switch case above, PARSE_ERROR_NO_RULE_MATCHES and PARSE_ERROR_WRONG_TOKEN are the only syntax errors that are reported as the other error types are simply indicative that one of these errors occurred elsewhere in the parse tree.

Semantic Errors

AST_ERROR_UNDECLARED_VAR is raised when an ProcessExpression look for an identifier and does not find

Semantic errors are reported in different functions based on the cause of the error. The ProcessExpression function reports errors for uses of undeclared identifiers. ProcessDeclaration reports errors for redeclaration and conflicts across scopes. ProcessOperation, ProcessOperator, and ProcessUnaryOperator report errors for invalid assignment targets (non-identifiers), chained assignment (e.g. a=b=c), incompatible types, and identifiers or assignment symbols that are missing. ProcessConditional and ProcessLoop report errors for invalid conditions, such as not an int compatible.

```
void ProcessConditional(ASTNode *ctx, Array *symbol_table) { // If statements
    assert(ctx->type == AST_CODITIONAL);
    assert(ctx->count == 3); // A conditional should have a condition and two scopes
    ASTNodeType outcome = AST_NULL;

    printf( Format: "Conditional Analyzing -> %s\n", ASTNodeType_to_string( t: ctx->type));

    // TODO: operation returns int, 0=false, non-zero=true
    outcome = ProcessOperation( ctx: &CHILD_ITEM(ctx, 0), symbol_table); // Process the comparison
    if (outcome != AST_INTEGER){
            printf( Format: "Error Reported -> Incompatible Conditional\n");
            ctx->error = AST_ERROR_INVALID_CONDITIONAL;
            array_push( a: semanticErrors,  e: (Element *)ctx);
    }
    ProcessScope( ctx: &CHILD_ITEM(ctx, 1), symbol_table); // ThenScope
    ProcessScope( ctx: &CHILD_ITEM(ctx, 2), symbol_table); // ElseScope
}
```

Example error reporting from ProcessConditional:

```
Conditional Analyzing -> AST_CONDITIONAL
Error Reported -> Incompatible Conditional
```

Runtime Errors

```
ASTNodeType ProcessOperator(ASTNode *ctx, Array *symbol_table){
    assert(ctx->count == 2); // Binary operator
    printf( Format: "Operator Analyzing -> %s\n", ASTNodeType_to_string( t: ctx->type));
    ASTNodeType LHS = ProcessExpression( ctx: &CHILD_ITEM(ctx, 0), symbol_table);
    ASTNodeType RHS = ProcessExpression( ctx: &CHILD_ITEM(ctx, 1), symbol_table);

    if (LHS >= AST_INT_TYPE && LHS < AST_SKIP) LHS = VarToLiteral( varType: LHS);
    if (RHS >= AST_INT_TYPE && RHS < AST_SKIP) RHS = VarToLiteral( varType: RHS);

    // Check for division/modulo by zero if RHS is a literal 0
    if ((ctx->type == AST_DIVIDE || ctx->type == AST_MODULO) &&
        (CHILD_TYPE(ctx, 1) == AST_INTEGER || CHILD_TYPE(ctx, 1) == AST_FLOAT) &&
        strcmp(CHILD_ITEM(ctx, 1).token.lexeme, "0") == 0) {

        ctx->error = AST_ERROR_DIVISION_BY_ZERO;
        array_push( a: semanticErrors,  e: (Element *)ctx);
        printf( Format: "Error Reported -> Division or Modulo by Zero\n");
    }
```

Runtime error detection is handled for division or modulo by zero. This is done in the ProcessOperator function. After evaluating the left-hand side (LHS) and the right-hand side (RHS) expressions of a binary operator, the analyzer checks if the operation is a division (AST_DIVIDE) or a modulo (AST_MODULO). If the RHS is a literal value (AST_INTEGER or AST_FLOAT) and it’s lexeme is “0”, it is flagged as a semantic error. The error is stored in the AST node as AST_ERROR_DIVISION_BY_ZERO, the node is added to the semanticErrors array, and a message is printed to the user.