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:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

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.

A computer screen with white text

AI-generated content may be incorrect.

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:

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

# AST Error Tokens

A screenshot of a computer program

AI-generated content may be incorrect.

# Changes to Code Generation Output

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

A computer screen shot of a program

AI-generated content may be incorrect.

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.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

# 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

A screen shot of a computer program

AI-generated content may be incorrect.

Example print\_token\_compiler\_message:

A black background with white text

AI-generated content may be incorrect.

## Syntax Errors

A computer screen shot of a program

AI-generated content may be incorrect.

Example report\_syntax\_errors compiler message:

A black screen with white text

AI-generated content may be incorrect.

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.

A screen shot of a computer code

AI-generated content may be incorrect.

Example error reporting from ProcessConditional:

A black background with white text

AI-generated content may be incorrect.

## Runtime Errors

A screen shot of a computer program

AI-generated content may be incorrect.

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.