Project 1 Report

ml language to C translator

Name: Leran Peng, Runzhi Zhao
StudentID: 23909531, 24022534
Unit: CITS2002

The University of Western Australia

September 19, 2024

Abstract

In this project, we developed a C11 program to translate, compile and execute a mini-language(ml) program. The ml language supports basic syntax rules, real numbers as a single data type, and variables that do not require prior definition. Our *runml.c* program first validates ml programs, then translates them into C11 code, compiles the C code using a system-provided compiler, and executes the resulting program, and remove the files produced by our C program. The transpilation process leverages C's role as a high-level assembly language, ensuring efficient compilation and execution of ml programs. The project was completed with a focus on standard utility behavior, command-line argument processing, syntax validation, and file management, and dynamic memory allocation.

Introduction to our runml Program

The program is designed to:

1. **Preprocess the ml file**: Remove comments and empty lines.
2. **Lexical Analysis (Tokenization)**: Convert the input into a sequence of tokens.
3. **Parsing**: Check the syntax of the ml code.
4. **Symbol Table Construction**: Build a symbol table to track identifiers and their scope.
5. **Translation to C Code**: Convert the ml code into equivalent C code.
6. **Compilation and Execution**: Compile the generated C code and execute the resulting program.
7. **Cleanup**: Remove intermediate files after execution.

This program addresses the following functionalities:

1. Translates ML language statements, expressions, and functions into equivalent C code, followed by compilation and execution.
2. Supports variable reassignment.
3. Automatically infers the return type of functions based on their return values.
4. Correctly ignores both blank lines within code blocks and comments at the beginning or end of lines.
5. Resolves conflicts between ML identifiers and C language reserved keywords.
6. Accurately passes command-line arguments to the ML program, using ‘arg0‘, ‘arg1‘, ‘arg2‘, etc.
7. Automatically distinguishes between integer and floating-point values when printing, with floating-point numbers rounded to six decimal places.
8. Properly recognizes the minus sign as a negative operator when placed before a variable or number, rather than interpreting it as a subtraction.
9. Correctly handles function parameters during invocation.
10. Identifies whether variables within functions are local (if assigned within the function) or treats them as global variables by default.
11. Supports nested function calls.
12. Detects and reports syntax errors in ML code, halting execution if any errors are encountered.

In this C programming project, we define constants for maximum limits, such as the number of lines, characters per line, total length, symbols, and identifier length to follow our project requirements.

To adhere to the project requirements, we begin by defining constants for maximum limits in our program. These constants include the maximum number of lines, maximum characters per line, total maximum length of all lines, maximum number of symbols, and the maximum length of identifiers. These definitions ensure that the program can handle the input within specified bounds and prevent issues such as buffer overflows.

In the lexical analysis phase, the program enumerates all possible token types that can be identified in the ml language. We define an enumeration TokenType that includes tokens such as identifiers, numbers, functions, return statements, print statements, assignment operators, arithmetic operators, parentheses, commas, tabs, line feeds, negatives, and unknown tokens. This enumeration allows the tokenizer to categorize each piece of the input code accurately, which is essential for the parsing process.

Each token identified during lexical analysis is represented using a Token structure that holds its type and value. The type field indicates the category of the token, while the value field stores the actual string representation from the source code. This structure facilitates the handling of tokens throughout the parsing and translation processes.

The program maintains a symbol table to keep track of identifiers such as variables, functions, and parameters, along with their attributes like type and scope. We define a Symbol structure to represent each symbol and a Symbol Table structure to manage the collection of symbols.

During the translation phase, we represent functions using a Function structure that stores their return type, name, parameters, and the number of parameters. This function structure which is used when generating C code from the ml code, providing necessary information to construct function definitions and calls.

Our preprocessing module reads the ml source file, removes comments (denoted by #), and skips empty lines. This cleans the input and prepares it for lexical analysis.

Our lexical analysis module converts the preprocessed input into a sequence of tokens, categorizing each piece of code according to the defined token types.

After that, we implement a function get_next_token that reads characters from the input string and identifies tokens based on the ml language specifications. It detect numbers, including floating-point numbers, and handle negative numbers by context, and identify identifiers and distinguish them from reserved keywords like function, return, and print. It also has operator and punctuator recognition, which recognize operators such as +, -, *, /, and punctuators like (,), and ,. And it differentiate between subtraction and unary minus based on the preceding token. This accurate tokenization is crucial for the subsequent parsing phase.

The syntax parsing module analyzes the token sequence to ensure that the code conforms to the ml language grammar. We employ recursive descent parsing, implementing functions that correspond to grammar rules:

- Expressions: The expression, term, and factor functions parse mathematical expressions, handling operator precedence and associativity.
- Statements: The statement function parses individual statements such as assignments, function calls, print, and return statements.
- Program Structure: The program_item function parses either a function definition or a standalone statement.

At each stage of parsing, we check for syntax errors. If an unexpected token is encountered, we output an informative error message and terminate the parsing process. We also differentiate between global and local scopes by tracking the current function context. Parameters and local variables are scoped within their functions, while global variables are available throughout the program.

The code generation module translates the parsed ml code into equivalent C code, generating a complete C source file ready for compilation. We organize the generated code into three main parts:

- Header Part: Includes necessary headers and utility functions.
- Function Declarations: Contains translated function definitions from the ml code.
- Main Function: Represents the entry point of the program, translating global statements.

To avoid conflicts with C keywords and standard library functions, we prefix all identifiers with a specific string. And we generate C function definitions by translating ml functions, including their return types and parameters. Statements and expressions are converted into their C equivalents, ensuring that the logic of the original ml code is preserved.Our program include a custom print_iof function in the generated C code to handle printing integers and floating-point numbers according to the ml language specifications.

After generating the C code, we compile it using the system's C compiler and execute the resulting program. To maintain a clean environment, we delete any intermediate files and free allocated memory. Then, our program free all dynamically allocated memory used during preprocessing and symbol table construction and remove temporary files generated during the compilation process.

However, the program has the following limitations:

1. It does not verify whether a function has been defined before it is called, resulting in a compilation error in the translated C code if an undefined function is invoked.
2. If the ML program references command-line arguments but insufficient arguments are provided during execution, the translated C code will produce a compilation error (e.g. if the *xxx.ml* program contains the statement *print arg0 + arg1* but is executed with the command *./runml xxx.ml 2*).

Conclusion

In this project, we inspired by a real compiler and successfully designed and implemented an interpreter and translator for the ml mini-language using C11. By meticulously following compiler design principles, we created modules for preprocessing, lexical analysis, syntax parsing, symbol table management, code generation, compilation, and execution. The program meets the project requirements and provides a logical, coherent implementation that demonstrates fundamental concepts in compiler construction.