CS335 : Milestone2 Documentation

Roll no.- 210750, 210557, 210543

April 4, 2024

1 Introduction

This document provides an overview of the tools used, compilation instructions, execution instructions, and output files generated for the project.

2 Tools Used

The following tools were used in the project:

- **Bison (parser.y)**: Bison is a parser generator used to generate the parser from the grammar file `parser.y`. It converts the grammar rules into a C++ parser (`parser.cpp`) along with the header file (`parser.hpp`).
- **Flex (lexer.l)**: Flex is a lexical analyzer generator used to generate the lexer from the lexical specification file `lexer.l`. It converts the regular expressions defined in `lexer.l` into a C++ lexer (`scanner.cpp`).
- **GNU Compiler Collection (g++)**: The GNU Compiler Collection is used to compile the source files into an executable. It compiles the generated parser (`parser.cpp`), lexer (`scanner.cpp`), and any additional source files (e.g., `ast.cpp`) into the executable named `parse`.

3 Compiler Implementation

The compiler implementation consists of the following phases:

1. Lexical Analysis: The lexer (`scanner.cpp`) generated by Flex tokenizes the input Python program.
2. Parsing: The parser (`parser.cpp`) generated by Bison constructs an abstract syntax tree (AST) from the token stream.
3. Semantic Analysis:
   - The AST is traversed to perform type checking and scope resolution.
   - Symbol tables are constructed to store information about variables, functions, and their properties (type, scope, line number, etc.). The symbol table implementation is in the `symtab.cpp` file, which includes the following main components:
     - `MAPVAL` struct: Represents an entry in the symbol table, storing details such as identifier, scope, line number, type, size, and additional information for arrays and functions.
     - `SYMTAB` class: Represents a symbol table, which is a hierarchical structure with a parent-child relationship. It contains an unordered map to store symbol table entries (`MAPVAL`) and methods to manipulate the symbol table.
     - `WriteSYMTABToCSV()` and `WriteAllSYMTABsToCSV()` functions: Write the contents of the symbol table to a CSV file.
4. Code Generation:
   - The AST is traversed again to generate the corresponding 3-address code (3AC) for the input program.
   - The 3AC representation is a sequence of instructions, each with at most three operands, capturing the semantics of the input program.
   - The 3AC code is stored in an appropriate data structure (e.g., a vector or list) for further processing or optimization.

The primary data structures used in the implementation are:

- Abstract Syntax Tree (AST)
- Symbol Table
- 3-Address Code (3AC) Representation

4 Supported Language Features

The compiler supports the following Python language features:

- Variable declarations and assignments (including support for basic data types like integers, floats, and strings)
- Basic arithmetic and logical expressions
- Control flow statements (if-else, while, for)
- Function definitions and calls
- Compound statements (e.g., nested if-else, loops)
- Scoping rules and variable resolution
- Type checking and error handling for type mismatches

5 Output Files

The compiler generates two output files:

1. **symtabs.csv**: This file contains a comma-separated value (CSV) representation of the symbol tables for each function in the input program. The columns include the syntactic category, lexeme, type, and line number.
2. **3ac.txt**: This file contains a text representation of the 3-address code (3AC) generated for the input program.

6 Compilation Instructions

To compile the project, use the `make` command. It involves the following steps:

1. Generate the parser using Bison:
   `bison -d -o parser.cpp parser.y`
2. Generate the scanner using Flex:
   `flex -o scanner.cpp lexer.l`
3. Compile the source files:
   `g++ parser.cpp scanner.cpp ast.cpp symtab.cpp -o parse -std=c++11`

7 Execution Instructions

To execute the compiled program, following command line tools are supported (Note: Two files need to be provided always, input and output. There is no default file.) By default, we can execute using
`./parse path-to-input-file path-to-output-file`

1. -input:
   `./parse -input path-to-input-file path-to-output-file`
2. -output:
   `./parse -output path-to-output-file path-to-input-file`
3. -help:
   Gives instructions to execute the file

8 Testing

The compiler implementation has been tested with a suite of test cases covering various scenarios, including:

- Correct programs with different language features
  - Variable declarations and assignments
  - Arithmetic and logical expressions
  - Control flow statements (if-else, loops)
  - Function definitions and calls
  - Compound statements (nested if-else, loops)
- Programs with syntax errors
- Programs with semantic errors
  - Type mismatches
  - Undeclared variables
  - Scope violations
  - Incorrect number of arguments in function calls
- Edge cases and corner cases
  - Empty programs
  - Programs with large input sizes
  - Programs with deeply nested structures

The test cases were manually created and executed, and the outputs were verified for correctness. Additionally, the generated symbol tables and 3AC code were inspected for accuracy and correctness.