INFO0085-1 Project - Compiler

Elie Azoury s226446 - Cédric Seron s175048

May 2024

1 Overview

This report provides a broad overview of our compiler implementation for the VSOP language. It details the tools used, the organization of the code, the responsibilities of each module, the data structures and algorithms employed, and the decisions made throughout the implementation. It also addresses conflicts, limitations, and possible improvements.

2 Tools Used

- **Flex**: Used for lexical analysis to tokenize the input source code.
- **Bison**: Used for parsing to generate the syntax tree.
- **LLVM**: Used for intermediate representation (IR) and code generation.
- **C++**: The primary language for implementing the compiler.

3 Organization

The compiler is organized into several modules, each responsible for a specific task:

1. **Lexical Analysis (Lexer)**
   - **Tool**: Flex
   - **Responsibility**: Tokenizing the input source code into a stream of tokens.
2. **Syntax Analysis (Parser)**
   - **Tool**: Bison
   - **Responsibility**: Constructing the Abstract Syntax Tree (AST) from the token stream.
3. **Semantic Analysis**
   - Ensuring the correctness of the program by checking types, scope, and other semantic rules.
4. **Intermediate Representation and Code Generation**
   - **Tool**: LLVM
   - **Responsibility**: Converting the AST into LLVM IR and generating the final machine code.

4 Key Modules and Responsibilities

- **ASTClasses.hpp**: Defines the structure of the AST nodes.
- **ASTClassesSemanticChecker.cpp**: Implements the semantic checks for AST nodes.
- **ASTClassesCodeGenerator.cpp**: Implements the code generation logic for AST nodes.
- **ClassSymbolTable.hpp**: Manages symbol tables for classes and functions.
- **CodeGenerator.hpp**: Contains definitions for the code generation process.
- **ProgramScope.hpp**: Manages the program scope and nested scopes.
- **lexer.lex**: Defines the lexical analysis rules and token definitions.
- **parser.y**: Defines the grammar rules for parsing VSOP and constructing the AST.
- **main.cpp**: Main function that drives the compilation process.

5 Data Structures and Algorithms

- **Abstract Syntax Tree (AST)**: A hierarchical tree structure representing the parsed program.
- **Symbol Table**: Stores information about identifiers, their types, and scope levels.
- **Program Scope**: Manages scopes in the program. It uses a map to store symbol-to-type mappings and supports nested scopes via a parent pointer. .

6 Implementation Choices

Each phase of compilation is handled by a distinct module, ensuring that the responsibilities are well-separated and the overall structure remains manageable.

6.1 ClassSymbolTable

The ClassSymbolTable class is essential for managing the class definitions in the VSOP language. Here's how it works:

1. The ‘ClassSymbolTable‘ maintains a map that associates class names with their respective class definitions. This map ensures that each class name is unique within the program.
2. The ‘addClass ‘method adds a class definition to the symbol table. If a class with the same name already exists, the method returns false, preventing redefinition errors. Otherwise, it adds the class and returns true.
3. The ‘getClass‘ method retrieves a class definition by its name if it exists.

The ClassSymbolTable plays a role in semantic analysis by ensuring that all class references are valid and that inheritance hierarchies are correctly resolved.

Program Scope

The ‘ProgramScope‘ class is crucial for managing variable and function scopes during compilation. Here's how it works:

Each ‘ProgramScope‘ instance can have a parent scope, creating a hierarchy that represents nested scopes. This hierarchical structure allows the compiler to resolve identifiers in the current scope and, if necessary, recursively in parent scopes. For example, in a function, the scope hierarchy would be:

$$\text{class scope} \rightarrow \text{method scope (arguments)} \rightarrow \text{block scope}$$

Each scope contains a symbol table, which is a map associating variable names with their respective types. This map ensures that each symbol is correctly typed and helps in detecting redefinitions within the same scope.

The ‘inFieldInitializer‘ flag indicates whether the compiler is within a field initializer. This is important for enforcing rules about using class fields during their initialization.

Symbols (variables or functions) can be added to the current scope. If a symbol with the same name already exists in the current scope, the addition is rejected to prevent redefinition errors.

To resolve a symbol, the compiler starts from the current scope and traverses up through the parent scopes until it finds the symbol or exhausts all scopes. This ensures that nested scopes can access symbols defined in outer scopes.

Each scope can be associated with an AST node, allowing the compiler to reference the part of the AST that created the scope. This is useful for context-specific checks and error reporting.

The ‘ProgramScope‘ class is extensively used in semantic analysis to ensure that variables and functions are used correctly according to the language's scoping rules. It helps manage the hierarchical structure of scopes within classes, methods, blocks, and other constructs. Additionally, it is helpful during code generation to determine where and what to store.

7 Conflicts and Resolutions

- **Shift/Reduce Conflicts**: We decided to fix the shift/reduce and reduce/reduce conflicts late in the project, we were able to fix them all but one. The only shift/reduce conflicts left happens when we call a method of an object in parentheses, such as "(new Cons).init(0)".
- **Reduce/Reduce Conflicts**: There no Reduce/Reduce conflicts in the parser.

8 Limitations

The compiler works well with most of the test cases. However, we can clearly see that some field's storage is duplicated, making the compiler use more memory than necessary. This could have been avoided if we designed more the compiler architecture before starting coding.

9 Improvements

- **Improved Error Messages**: Adding more descriptive and context-aware error messages.
- **Refactoring**: Further modularizing the codebase for better maintainability and readability.
- **Extended Testing**: Increasing the test coverage to include more edge cases and complex scenarios.

10 Conclusion

In conclusion, the project provided a comprehensive learning experience in compiler construction, despite the challenges faced. Future iterations could benefit from more advanced features and refined error-handling mechanisms.