INFO0085 : Compilers

Final Compiler

LOWETTE Quentin - SCHLEGEL Maxime
s150853 - s187511

AUGUST 2019

1. Introduction

For this project, we were asked to write a compiler for the VSOP language that whould handle the lexical, syntax, semantic analysis of input code, that will be able to generate LLVM intermediate representation of that code and generate a native executable. We choose to implement our compiler with the C++ language. We choose to use C++ because the different tools that we used to create the compiler (lex and bison) are both implemented for C but are easily compatible with C++. Moreover C++ allows to manage objects and classes more easily.

2. Implementation

2.1. Code organization

Our code is organized into 5 different folders :

— The `ast_node` folder contains all the files related to the abstract syntax tree.

— The `exception` folder defines the different custom exceptions that our compiler may return.

— The `symbol_table` folder holds all the files that defines our implementation of the symbol table.

— The `debugger` folder define a simple object used to debug our compiler during its development.

— The `compiler` folder contains the basic files that form our compiler, including the `flex` and `bison` files.

2.2. External tools

As already mentioned in the previous section, we use the `flex` and `bison` tools to handle the lexical and syntax analysis.

The `vsop_lexer.l` file is used by `flex` to create the lexer for the VSOP language which is used for the lexical analysis.

The `vsop_parser.y` file is used by `bison` to create the parser for the VSOP language. The parser is used to create the abstract syntax tree of the parsed program. Unlike the first iteration of this project, we have now defined different kind of nodes that all inherited from the abstract class `Node`. The usage of these nodes is also defined in `vsop_parser.y`.

2.3. Data structures

We define two main data structures for this project an Abstract Syntax Tree (AST) and a symbol table.

The AST is represented by a series of nodes that holds pointers to their children. We defined as much class as there are possible theoretical nodes in an abstract tree of a VSOP program. This allows us, unlike the previous iteration of the compiler, to have simpler methods because they are specialized for each kind of nodes. It remains easy to go through the tree and search it as every nodes is a subclass of `Node`. The semantic analysis and the code generation are thus made easier to perform.

We defined a symbol table using a mapping between strings and `SymbolTableScope` objects. This mapping allows us to have named scopes. The symbol table holds also 3 pointers to differents scopes : one for the root scope, the topmost one, one for the current scope to reach it quickly and one for the previous one to restore it when we leave the current scope. A `SymbolTableScope` object contains 2 mapping from strings to `SymbolTableEntry` objects. However one use `VariableEntry` and the other one `MethodEntry`. Those two classes are subclass of the `SymbolTableEntry` one. The first mapping is thus responsible of holding a list of all the variables defined in the scope and the second holds a list of all the methods defined. Each scope has also a pointer to its parent scope.

3. Compiler

3.1. Lexer

The lexer role is to decompose the input file in different tokens, it must also detect lexical error such as invalid name format or the use of reserved identifier. When it detects such errors it stops the token decomposition immediately and reports the error to the user. This part is managed by the external tool `flex`. We defined what are the tokens of the VSOP languages and what is the set of acceptable characters and it generates for us a lexer that we can call in our main function with `yylex` to obtain the next token in the given input source.

3.2. Parser

The parser is used to create the Abstract Syntax Tree from the token stream of the lexer. According to a given grammar, it will turn token into AST node and define which ones are the children of the others. It detects syntax error and report them to the user. The parsing is done by the external tool `bison`. We give it a grammar and the rules according to which the parser will build the AST. Thereby the right subclass of `Node` is inserted in the tree when facing a given token. The parser generated by `bison` is used by calling the function `yyparse` inside our main function. This parser will use the lexer described in the previous version as `flex` and `bison` are made to work well together.

The parser uses left recursive syntax to define its rule. This allow to parse the sequence in a bounded stack space (as describe in the bison manual. To accommodate this, we transform some definition from right to left recursive, particularly the block are defined without the brace to ease the recursion, they are added around the block when it is used. Other definition atr defined left recursive such as the class body or the formals.

3.3. Scope checker

The scope checker take the Abstract Syntax Tree generated by the parser and detects type and scope errors. It will, as it searches for errors, decorate the tree with the different types it encounters. To do this semantic analysis, we have defined a `ASTProcessor` object that will perform 2 passes of analysis.

The first one is called "preprocess" and it registers all the classes definition of the given program. It records these inside the symbol table along with their fields and methods. At this step, it also checks that there is no circular inheritance and that the `Main` class is defined properly with its `main` method. To find all classes, we simply look among the children of the root node, the `ProgramNode`, the topmost one. If a class is found, the analysis continue inside it looking for methods and fields. If a node different from a class is found among the children of the program node, if there are nodes different than methods and fields in a class or if there are definition errors (as a redefinition of a field), the `ASTProcessor` will throw an exception.

The second pass is the semantic analysis strictly speaking. Beginning in the root node, the `ASTProcessor` will go through all nodes of the tree and check if there is no type or scope error. For the scope checking, it will look that the used names (of variables and methods) are defined in the current scope. This task is perform by the symbol table which will search in the current scope and then recursively in its parents until it reaches the root scope. The type checking is done by checking that the node uses the proper type. To do so, the `ASTProcessor` check recursively the type of the children of a node and then according to those it sets the type of the current node or it throws an exception if for example the types of the two operands of an operator doesn't match.

3.4. LLVM code generator

We implemented the LLVM code generation in a similar way that for the semantic analysis, i.e. each node has a `codeGen` method that generates an intermediate representation in LLVM.

The generation begins in the root node, the `ProgramNode`. It then travel the tree by recursively calling the `codeGen` method of the nodes. To generate the appropriate LLVM code, we didn't write it directly. We used instead the advantages given by the LLVM library and its `IRBuilder` class.

At this stage, we were able to implement 23 of the 32 `codeGen` methods of the different nodes. We were not able to implement the object-oriented part of the VSOP language.

4. Extensions

We did not implement any extensions of our compiler.

5. Limitations

First of all, our compiler doesn't actually compile file due to the incomplete LLVM generator.

However, even complete, the compiler would have some limitations :

— Only one file can be compiled. Unlike modern compiler as gcc, it is not possible to compile multiple files.
— Our error handling can be improved with better error messages or by displaying more than one error at the time.
— During the semantic analysis, we go through the tree several times and that might surely be improved in order to gain in performances.

6. Feedback

The main difficulties of this project was the time and the understanding. There is quite a gap between the theoretical course and the project and it takes us some time at each step to figure out how to rightfully implement the different parts of the compiler. Specially with the IR code generation where we didn't understand how to properly encode classes.