University of Liège

Project : the VSOP compiler

Compilers

Maxime Meurisse (s161278)
Valentin Vermeylen (s162864)

Master in Engineering and Computer Science
Academic year 2019-2020

1 Introduction

1.1 Overview

The goal of this project was to implement a compiler for the VSOP language. To do this, we decided to use the python language and the PLY library (for lexical and syntactic analysis).

Our compiler is divided into 4 tools : a lexer that performs the lexical analysis, a parser that performs the syntax analysis, a semantic tool that performs the semantic analysis and finally an LLVM tool that generates the executable file corresponding to the VSOP source file.

Each tool works with the previous one. A classical execution of the compiler is as follows : a lexical analysis is first performed. If no errors occurred, a syntax analysis is performed based on the tokens generated by the lexer. If no error occurred, the compiler then performs a semantic analysis based on the AST (*Abstract Syntax Tree*) generated by the parser. Finally, if still no error has occurred, the LLVM code and the executable are generated based on the annotated AST generated by the semantic analysis.

The precise operation of each tool as well as the data structures and methods used are explained in the following sections.

1.2 Error handling

For error handling, we started from the following general principle : *each tool displays as many relevant errors as possible and stops the compiler execution after displaying its errors.*

We chose to display as many relevant errors as possible (and not just one) because we think this system is more convenient for any developer who wants to work with this compiler. Indeed, it seems simpler to fix several potentially independent errors at once rather than having to fix them one by one each time the compiler is run.

Since this rule is not easy to implement, some tools derogate from it in certain cases. Details are explained in the sections of each tool.

We have also chosen to stop the compiler execution if a tool returns one (or more) error(s). Indeed, we believe that an error in one tool will almost systematically lead to an error in the next analysis tool (for example, an incorrect token will lead to an error in the syntax analysis since no grammar rule will be found for this token). We made this choice in order to avoid a snowball effect and thus not to display a large number of errors to the user, some of which making no sense.

1.3 Code organisation

The archive contains several folders, a `main.py` file and a `Makefile`.

Each tool is located in the folder with its name. The `main.py` file is the main compiler file. The `Makefile` is used to install all the tools needed by the compiler user (`make install-tools`) and to generate an executable version of the compiler (`make vsopc`).

After that, the compiler can be used via `./vsopc <VSOP-SOURCE-FILE>`. Several options are available and can be accessed via `./vsopc -h`.

2 Lexical analysis

For the lexical analysis, we used the python PLY library. This provides a `lex.py` module which is a pure python implementation of the well-known "lex" tool.

This tool first asks to define a list of tokens. Once this list is defined, a regular expression must be defined for each token (a python function containing the processing of this regular expression must be created for each token).

When the regular expressions are defined, we just have to launch the tool. It will read the VSOP source file and try to associate each element (each character or group of characters) to a regular expression. The tool uses the *longest prefix match* rule to avoid any ambiguity if a token matches several regular expressions.

If an element does not match any regular expression, a special error function is called.

2.1 String literal

Tokens "string-literal" were a difficult part of this tool. Indeed, managing special characters, escaped characters and invalid characters was a challenge.

We implemented a special function to handle this type of token. It works correctly except for one very specific case : when a line break is entered incorrectly and followed by the invalid character `"\n"` (not preceded by a \to be escaped) and is not on the first line of the string (because the string would be split on several lines), then the position of this invalid character is not correctly reported, but the error is still displayed with a meaningful message.

We tried to correct this case but we had some difficulties. In the end, we thought that being an extremely specific case, it was not too much of a problem.

2.2 Comment handling

To manage comments, we have implemented several states in our lexer: a "normal" state and a "comment" state.

When the lexer is in its "normal" state, it tries to match tokens with all encoded regular expressions (including comments). If the lexer matches a "comment opening" token, then it switches to the "comment" state.

When the lexer is in the "comment" state, it no longer considers the encoded regular expressions. It ignores all the elements of the file until it encounters either a line break if it is a one-line comment, or an "start/end of comment" token if it is a multi-line comment.

To manage comments on several lines that can be nested, we used a stack. Each time a comment opening (`"(*"`) is encountered, its position is added to the stack. Each time a comment close (`"*)"`) is encountered, an item is removed from the stack. If the stack is

empty when an item needs to be removed, or the stack is still full at the end of the VSOP file, an error has occurred.

2.3 Error handling

The error handling of this tool respects the previously stated rule : an error is reported for each wrong token in the VSOP file (and all these errors are displayed).

3 Syntax analysis

For the syntax analysis, we used the python PLY library. This provides a `yacc.py` module which is a pure python implementation of the well-known "yacc" tool. This tool is a *Look Ahead Left-to-Right (LALR) parser generator* and works in a bottom-up way.

This tool requires to define a series of grammar rules. These rules may involve other rules or tokens. Once these rules are defined, the tool can be launched to parse a VSOP file.

It works as follows : the parser takes as argument the lexer at its instantiation. It then uses the lexer to generate the tokens one by one. As the tokens are generated, the parser tries to match a grammar rule. If no grammar rule could be matched, a special error function is called.

For each grammar rule matched, the corresponding python function is called with the different elements of the grammar rule as arguments. Each function called creates a new node in the AST.

3.1 Abstract Syntax Tree

We have created a customized structure to generate the AST. For this, we used the object-oriented paradigm : each node of the tree is an object and contains references to its child nodes. The root node of the tree is an object called `"Program"`. It is this object that will symbolize the AST.

For example, when the AST will have been generated, the `"Program"` object will contain a reference to all the `"Class"` objects representing the different classes composing the VSOP program. Each `"Class"` object will contain references to `"Field"` and `"Method"` objects representing respectively the fields and methods defined in the class. Each `"Method"` object will contain references to `"Formal"` objects, etc.

We have also taken advantage of the inheritance mechanism : each node inherits from a `"Node"` class and each expression inherits from an `"Expr"` class (which itself inherits from the `"Node"` class).

When a grammar rule is matched, we instantiate a new object representing that rule and return it. Each rule returns the instantiated object (except the root node rule), so the AST is built recursively as we parsed.

3.2 Precedence rules

The grammar defining the VSOP language is ambiguous. To remove this ambiguity, precedence rules have been defined.

The PLY "lex" tool allows us to encode these rules simply in a tuple. Within this tuple declaration, tokens are ordered from lowest to highest precedence.

3.3 Error handling

Handling errors in the parser is a real challenge. By default, when no grammar rule could be matched, an error function is called. This function displays an error in `stderr`. However, his message is not very clear; it is simply a "syntax error" message with the faulty token.

We have therefore tried to implement clearer and more personalized error messages. For this, we have encoded false grammar rules. When one of these rules is matched, we can then display a personalized error message according to the matched rule.

For example, if a semicolon is required, we encode the grammar rule with the semicolon but also the one without the semicolon. The first rule will be tested and not matched. The second will then be tested and matched. We will then know that we are in the case of a forgotten semicolon and we will be able to display a corresponding error message.

We have obviously not encoded all the possible wrong grammar rules because there would be too many of them. We only encoded the frequent errors (forgetting a semicolon, forgetting the type, etc).

The parser, contrary to the general principle stated above, shows only one error. The display of several errors must be meticulously managed in order to be relevant. We didn't implement it at the beginning and we didn't have the time to go back and do it properly. So we decided to display only one error at a time.

4 Semantic analysis

For the semantic analysis, we did not use any tool or library; we implemented all the analysis ourselves.

For this, we used symbol tables (implemented using the object-oriented paradigm) and an Eulerian traversal of the AST.

4.1 Passes in AST

Our semantic analysis tool takes the AST as an argument when it is instantiated and returns this same AST where all the expressions have been annotated of their type.

For this, we run through the AST several times in order to make all the necessary checks and annotations.

First, we partially scan the AST to check the class declarations (we iterate several times on the list of classes in the `"Program"` object). For each class encountered, we check if it has not already been declared and we associate a table of symbols with it. This table

contains information such as the name of the class and its position as well as references to the symbol table of its parent class (either a custom class or the `"Object"` class), the symbol tables of its fields and the symbol tables of its methods. At the end of this first pass in the tree, we are sure that all the class declarations are valid (no re-definition or cycle for example).

Second, we continue with two partial passes : a first to check the declarations of fields and a second to check the declarations of methods. These passes are very similar to that concerning the classes : checks are made (to be sure that a field has not been re-defined or that a method is re-defined correctly, etc.) and a symbol table is created for every element. The symbol tables created are linked to the corresponding class.

Finally, we make a last pass in the tree to check the type of each expression. This pass in the tree is complete and is in the form of an Eulerian route. The analysis is done recursively : when an expression is analyzed, all the expressions making it up are first analyzed. So we analyze the deepest expressions in the AST first. If an expression is valid, it is annotated with its type, otherwise an error message is displayed.

Remark. Sometimes, when we do a partial pass (for example, only on the classes), we actually iterate several times on the classes. Maybe we could have iterated only once, but we preferred to separate the actions : one iteration to retrieve the parents of each class, another iteration to check the cycles, and so on. We call this set of iterations a "pass" although in reality it is several passes. Being simply iterations on a list (the list of classes for example) and not the whole AST, we think that this does not induce huge inefficiency.

4.2 Symbol tables

We used the object-oriented paradigm to implement the symbol tables. We have created a symbol table for each element that can define a new scope, namely classes, methods and `"let"`. We also created a similar object for the fields.

Each table contains references to the tables in its scope. For example, the symbol table of a class will contain references to the symbol tables of its fields and methods.

To check if an element is well defined in a scope, we used a stack in our expression analysis : when the tool walks in the tree, it adds and removes the symbol tables from the stack according to the elements that he crosses. For example, when working in a class, the symbol table for that class will be added to the stack. If a `"let"` is defined in this class, its symbol table will be added to the stack. Thus, when checking a scope, the most recent element on the stack will be analyzed first. In this example, the tool will look first in the symbol table of the `"let"` and then in the symbol table of the class.

4.3 Error handling

In this tool, it is easier to display clear and personalized error messages because all the analysis is done by us. In addition, this consists of analyzing each type of expression with well-defined functions; it is therefore simple to know in what context the error was raised.

Again, although perhaps easier to manage than parsing phase, we did not have time to implement a mechanism to display multiple errors. So we limited the display to one,

preferring not to implement something bad in the rush.

5 Code generation

This part was, we felt, the most difficult part of all. To do it, we used the python library llvmlite. This one works in LLVM 8 but did not pose a problem for LLVM 9.

To generate the code, we proceeded in two phases : initialization and analysis.

5.1 Initialization phase

In this phase, we create a large symbol table containing information about each class (its VTable, its fields, its methods, etc) as well as the `"new"` and `"init"` methods of each class.

The symbol table keeps, for each class, a reference to the LLVM objects (created with llvmlite) concerning it (structure, VTable, methods and their type, arguments, etc). This makes the next step possible : analysis.

5.2 Analysis phase

During this phase, we analyze each expression of the program according to the same principle as the semantic : we make an Eulerian traversal of the annotated AST. To each node corresponds a method to generate the corresponding LLVM code based on the elements previously stored in the symbol table.

To manage the different contexts (when entering a method, or one or more `"let"`) we used a stack on which we add and remove the symbol table of each context (just like for semantic analysis).

5.3 The unit type

The `unit` type is represented by the `void` type in LLVM. However, the latter can only be used as the return type of a function. It is therefore not possible to directly use an element (field or argument of a method) of type `unit` (at the risk of causing compilation errors).

To overcome this problem, we have simply ignored all `unit` elements in the LLVM generation and manually returned a `void` (in LLVM) when a `unit` element is encountered in the VSOP code. This avoids storage and allocation operations (`store` and `alloca`) on `void`, which is not allowed in LLVM.

5.4 Error handling

In view of the various analyses carried out previously, this part is no longer supposed to generate errors. We have therefore written the code for this part without specifically implementing verification mechanisms (for example, when we retrieve the type of an element, we assume that this type exists and is valid, we do not verify it).

We have also not implemented a mechanism to prevent certain errors at runtime (for example, spotting infinite loops or memory leaks). We found this quite difficult and

thought that an error would be generated at runtime anyway. We were also inspired by modern languages that do not implement such mechanics either.

6 Extensions

We didn't have time to implement extensions because we started a bit late with the generation of the LLVM code. We are a bit disappointed about that.

If we had had the time, we would have started by adding the `">"` and `"or"` operators and the `"double"` type. We would also have tried to implement a `"for"` loop.

We had also thought about adding warning messages (for example, if an `"int32"` is used instead of a `"double"`). Just like in C, a warning message would not stop the creation of an executable.

7 Limitations and retrospective analysis

We are quite satisfied with the result obtained for our compiler. This seems to work in the vast majority of cases and is robust to fairly specific tests.

However, we are a bit disappointed in the error handling. In the majority of cases, we failed to meet the principle we had set, which was to display several relevant errors. We realized that this was not an easy task and we didn't find the time to go back over it properly. Nevertheless, we are happy that we were able to implement relevant error messages in the majority of cases.

We spent a considerable amount of time on the last part : the generation of the LLVM code. The library being quite undocumented, we had a lot of trouble getting through it. In the end, we got a satisfactory result, but if we had to do it again, we would have done it sooner !

8 Conclusion

This project allowed us to discover and learn a lot of new things. We were able to discover the main principles of a compiler and deal with the different difficulties of implementing such a tool.

The VSOP language for which we had to create a compiler was well defined, complete and affordable enough to create a first compiler.

We spent a large amount of hours on this project, while staying within the reasonable limits defined by the course workload. We estimate that we each spent approximately 80 hours on this project.