Master in applied sciences
Computer science and engineering
Year 2018-2019

Compilers

Project report

INFO0085-1
BRUSTEN Julien

HOCKERS Pierre
MOITROUX Olivier
ROEKENS Joachim

Liège

May 2019

Abstract

In this project, we had to implement a compiler for a custom language called VSOP. This compiler had to be implemented in 4 steps : First a lexer with Flex to implement the lexical analysis. The second step was the implementation of a parser, using Bison, to perform the syntax analysis. The 3rd step was to perform the semantic analysis. Finally, the input code had to be translated into the intermediate representation language LLVM. For this project, we decided to use the language C++.

Contents

1 Project Organization

1 Sub-parts

The project is organized in 4 main entities : the lexer, the parser, the semantic analyzer and the code generator. `main.cpp` initializes global variables, handles the given command and arguments to execute whatever parts of the code are needed to perform the requested task. It also handles the incorporation of the VSOP librairies that we have implemented. Another file that is used by many sub-parts of the project is `extern_var.hpp`, that holds global variables that will be used across several parts of the program.

The lexer's role is to read the input raw code and process it to extract tokens, or to trigger lexical errors if what it read was incorrect according to the rules of VSOP.

These tokens are used as an input by the parser that will perform a syntaxic check on them, and trigger syntaxic errors whenever an error is read. In order to do so, it will build an Abstract Syntax Tree (AST) that will be our main tool all across the rest of the program (cf. Section 3).

The semantic analyzer will get this AST and use it to perform a semantic analysis, and trigger semantic errors if need be. It will at the same time collect important information concerning the types and return type of variables and blocks.

Finally, again by using the AST as support, the code generator will convert all the input code in LLVM.

2 Lexer

The lexer code can be found in the file `lexer.lex` in the root directory of the project. This code reads the input code and will extract tokens from the input. This is done thanks to *Flex*, which enables us to define regular expressions that specify how to represent the different elements one can encounter in VSOP. Writing these regular expressions down enable flex to recognize the sequences of characters and match them to a token, or to write an error. Let's note that the allowed tokens are not directly defined in `lexer.lex` but instead in `grammar.y`. These definitions can then be imported in the lexer with `grammar.tab.h` that is generated by *Bison* from `grammar.y`. This technique allows the lexer and parser to share information in a convenient way later on.

In order to track the position of each token, we use the macro `YY_USER_ACTION update_loc()` so that each position of token is automatically registered for later use. However, in some specific cases, it is necessary to set it manually. We define `location_step` and `location_line` to perform operations on the position by settings the relevant fields of the structure `yyloc`.

Several macros representing useful entities such as digits, object ID, etc... have been declared at the beginning of the lexer.

The lexical analysis is launched by `main.cpp`. The lexical analysis will stop reporting errors if the global variable not2MuchLexErr is set to false.

3 Parser

3.1 Overview

The parser is called through the function `yyparse()` generated by *Bison* in the file `grammar.tab.c`, that is created when compiling the file `grammar.y`. The grammar recognizes the tokens extracted by the lexer and defines grammar rules to check if VSOP syntaxic rules are respected.

As the grammar recognizes the different global element, (classes, methods, fields, if, variables, assignements, etc...) it will build the *Abstract Syntaxic Tree*. Each time an entity is recognized, the grammar will create the matching AST node (ProgrammNode, ClassNode, FieldNode, AssignNode, etc..) and potentially link it to children or parent nodes if needed.

The class implementing these nodes are lay down in the folder `AST`. A generic `AstNode` class is defined with some virtual methods. This allows specific nodes to extend it while sharing the same base code. It is possible to identify the type of an AST node via the enumeration `NodeType` defined in the folder of the same name.

The whole tree starts from a root that is the programNode (representing the program) and holding all the classes. A global variable, `rootProgramNode` is used throughout the whole project to track the root node of the AST tree. That way, after the tree is created, it is accessible anywhere in the rest of the program. The nodes of the AST tree contain informations about the elements represented by the node. A class will for example hold

the methods and fields it contains. All nodes also know at which row and column they were encountered in the input VSOP program, which will be useful to output errors later. The key word `error` helps us trap errors that would happen at expected locations, as for example a lacking brace at the end of a class body, and therefore display a more accurate error message.

3.2 Implementation specific details

We decided to use a *Bottom up* approach to perform the syntaxic analysis. While the grammar rules are expected to be easier to write, the downside is a more complex building of the AST tree. One of the most important building block is the `blockNode` that contains a vector of `ExprNode`s. Due to the nature of our grammar, it is sometimes required to merge two nodes together. This can be achieved by using `fuseXNodeBack()` that will merge the calling node with the one given as input. We use the same technique with formals and arguments.

Another detail to notice is the fact that *dotOp* are like *fctCall*. By default, calls are self.method() but if the call is detected to be performed from another object later on, "self" is replaced by the calling object via the method `changeObject` of `callNode`.

4 Semantic analyzer

The semantic analyser relies heavily on the `AST` and is implemented in the `semantic` folder in `semantic.cpp`. It can be seen as made of 3 parts :

1. the `classCheck`
2. the `ScopeCheck`
3. the `typeCheck`

These 3 checks are done iteratively in this order. In this way, our semantic analysis is performed externally of the AST tree while still extensively using it. The result is an annotated AST tree.

The semantic analysis also relies on other files. `Types.cpp` defines a dictionary of names/pointer to class nodes. It is used to track all the types define is the program being analyzed, the built in and library types and to load our vsop library (IO). In the scope directory, other classes have been defined and are discussed in section 4.2.

4.1 classCheck

`classCheck()` will look after any error in the declaration of the classes.Is there any inheritance cycle ? Do all classes from which other classes claim to herit actually exist ? Is there any double declaration ? Is there a (unique) main class ?

In order to do all this, the `classCheck` will take from the `rootProgrammNode` the vector of classes used in the program and store them in a vector called `remaining`. It will then iterate on it and check all classes to see if they are correctly declared, and store them in the vector `declared` if they are. If during its analysis the `classCheck` can assert that a class is badly declared, it will delete it from the `remaining` vector and display an error. If we validate or delete a class, the iteration has to start over to avoid creating a segfault by using the iterators.
During these iterations, we will check if the classes are inheriting from themselves, have already been declared earlier. If the class currently being examinated passes these tests, we have to check for its inheritance. We will check if the class inherits from an already declared object (so the classes that inherits from no other homemade class will be added first as they inherit from `Object` which is obviously already considered validated), and if yes add them to `declared` vector. This is also the point where we will set the parent classes in the `classNodes`.

At the end of these checks, we should have deleted or validated a class. If not, and if the vector declared is not empty, it means that we have classes that are not double declared, that inherits from classes that exist, and that do not inherit from themselves. Its seems to us that the only case where that is possible is in case of inheritance cycles. In order to be sure not to miss any case, we simply stated that there was an inheritance error for these remaining classes.

4.2 ScopeCheck

Once we have gathered all the correctly declared class, we will start performing the ScopeCheck. This is the step where we check if all variables, functions and classes used at any point in the VSOP code are indeed available at said place. In order to do so, we will create several classes `Scope`. These classes have the ability to check if a variable, function or class is accessible from the element they represent. A scope is an entity containing other scopes (child Scopes) and itself contained in another scope (called parentScope). They will always conatins the variable that were declared within them, and nothing more. We can find 4 different `Scopes` :

1. `ProgramScope` represents the program and keeps track of all the different classes implemented in it under the form of a vector of `ClasssScope`. It is the only scope that isn't contained in a parent Scope
2. `ClassScope` Represents a class and keeps track of the methods and fields from the corresponding class by storing their names. It will also store a vector of `MethodScope` and have a pointer on its eventual `parentClassScope` and on the `parentProgramScope`. This is the only Scope that can have 2 parentScopes.
3. `MethodScope` represents a method. Contains the names of the different variables held in the corresponding method in the VSOP program. Has a pointer toward its `parentClassScope`.
4. `BodyScope` Represents a body. We call a body all blocks of instructions : the block initializing a field, the block executed in a while, the block executed for a if, etc.. It stores the names of all variables declared in the corresponding block and has a pointer on its `parentScope` that can be either a `MethodScope`, a `classScope` or another `bodyScope`.

All Scopes have a `inScope` function that allows them to check if an element of a given type (we here call type "class", "method", "variable") is in the scope of the element they represent. That way whenever a name of variable, function or class is used, the scope can check if this name is legal here.

`inScope` works by checking for the name of the element in itself if the type is one it can hold(for ex a variable in a `methodScope`). If the type can't be found in the scope (for example if we ask for a class name in a body) or if the name isn't found in the current scope, `inScope` will call the `inScope` function of its `parentScope`. This will allow us to recursively look through our scope tree for the requested name. If we find it, everything is fine. If not, we trigger an error.

We build the Scope tree by creating a `ProgramScope` and filling it with `classScopes` built by going through the vector of declared class we got from `ClassCheck`. We will then go through all these classes and create their `methodScope`, and then in these `methodScope` the `bodyScope` etc...

4.3 TypeCheck

Typechecking start by analysing individually each class nodes and its fields and their body. Then, it proceeds to the analysis of the methods and their body. While it is first easy to check the existence of the types, thanks to the `Types` class, we then need to dive deeply in the AST node to analyze the dynamic return types.

To do so, `getAndCheckTypeBody` checks that a block node satisfies its expected static type. If no static type is provided, this method just returns the dynamic return type of this block. As a block node is nothing else than a wrapper of a vector of expression nodes, it calls a method called `getAndCheckTypeXpr` to analyze an expression in the same way. The analysis is then performed based on the node type, sometimes by exploiting recursion. Some more specific methods have been defined (for callNodes, ...) if required. Body scopes are able to check themselves recursively by calling their method `checkYourself`.

As the dynamic type may change from a scope to another, we use a vector of `scopeLocation`, to track the different scopes and the variables they contain. `getVarTypeWithinScope()` can then be used get the scope that the node is in for the given node row and column. Then with the scope, it is possible to retrieve the dynamic type of a variable via id. Finally, when the dynamic type has been retrieved, the dynamic type is set to the variable node.

5 Code generator

For this part of the project, we took the decision not to use the LLVM librairy. This is driven by a lack of time. As there was already a new language to learn (LLVM), we feared that learning how to use the library would cost us too much time to hope bring convincing results for due date. Moreover, without a good understanding of the LLVM language, understanding the library would tend to be difficult but without the library, if we wanted to

use it, we could not have coded anything. Thus it would have taken more time to get result. Instead, not using the library allowed us to freely dive into the llvm to directly get results. This was also to avoid the previous case where most of the code was done but because of one missing module no tests was working which cost us 2 null grades out of the 4 deadlines.

We therefore used our AST tree again and added a method `codeGen` to each node. These methods create the LLVM code corresponding to the VSOP code they represent. The full generation of the code is done recursively by calling `codeGen` on the `rootProgramNode`. The `codeGen` in `rootProgramNode` will then generate the code related to the program, then append to it the code related to the classes held in it by calling `codeGen` on these classes. In these class, `codeGen` will generate the code related to the class then append to it the code related to its field and method by calling their own `codeGen`. Etc... At the end we get a string representing the whole VSOP code in its LLVm representation.

In order to support this implementation, we have introduced global variables in the global_extern.hpp file in the LLVM folder. Some of them are map that link VSOP names to current LLVM name, or variable that tells us when we need to remember if we changed the LLVM name of a variable, etc... We also widely use a variable called `vsn` that is basically the count of variables used in LLVM. We decided to use mainly numbers to represent our variables in LLVM and we therefore needed to keep track of the number corresponding to the next variable. Hence `vns` : every time we create a new variable in LLVM, we name it after `vns`, then increment vns, and pass it to the next `codeGen` when called.

The idea to go from the object oriented class to the LLVM was to create a structure type containing the fields of the class and the fields of its parents. Then the functions of the class are defined as global function with the name 'method name'+'class name'. Any method inherited from a parent which is not overwritten is also defined as such. The LLVM function will take the same arguments than the VSOP one, at the single difference that the LLVM function has one more argument which is a structure type of the class of the function. This way, if the function needs to get or update a field of the class, it can take it from this argument. Unfortunately doing this has some drawbacks which will be described in Section 1.

If understood correctly one of the support given by the library is the naming of variable. In our code, we used a global dictionary containing a mapping between the VSOP name of the variable and its LLVM name. Moreover, a counter called vns (VariableNameStart) was used to keep track of the next name to use (all LLVM names are numbers in our code). If a variable is not contained in the mapping , it means it is a field of the class which hasn't been used before. Thus it need to be collected from the structure of the class given in argument of the function. Using the dictionary proved to work very well with the let which allowed to easily change the mapping to the variable if the let 'overwrite' locally a variable.

6 VSOP extension

Only one extension has been added. It is the possibility to add some build-in class to the VSOP language. It already has the IO and Object class. But any other can be added and be used as any other build-in class. To do so, the only thing to do is to add the VSOP implementation of the class in the file IOlibrary.vsop.

2 Difficulties

1 Lexer

Linkage of main, lexer and parser : when implementing the lexer, a lot of time have been lost to make the `main.cpp` able to launch externally the lexical analysis. Then, we faced issues to make the link between the `.y` and `.lex` file. Indeed, our lexer was not able to recognize the TOKENs defined by the parser and the parser couldn't retrieved the information filled by the lexer.

Numbers: Quite some time have been spent on the processing of numbers (hexadecimal, ...).

2 Parser

Creating nodes in the grammar rules : For the second deadline, we didn't succeed to instantiate our nodes from the grammar rules. It took a bit of time to understand how to use *Bison* correctly in this regards.

The IO library: While the IO library was first hardcoded in the code, it soon showed its limitation. We thus defined its signature inside an external file. It was normally planned to import the library (file) within *Types.cpp*. However, we faced again issues to perform a lexical/syntax analysis from another file of `main.cpp`. As resolving external linkage took a lot of time for the lexer, we gave up this idea and decided to parse the IO file in `main.cpp` which isn't a problem after all.

Tracking location of tokens: we didn' have the time to implement the location tracking of tokens for the second deadline. However, it didn't take a lot of time to fix.

toStr() : many tests failed for stupid reasons while printing the AST node. Some segfaults slowed down the development. The reason was often an ommision of the return keyword (which is not detected by the compiler!). Printing the relevant type was not easy to manage because we were all working on the same part of the code all together.

3 Semantic Analyzer

Frequent modification of the grammar: While implementing the semantic analysis, we often had to change the behaviour of the grammar. At first, we relied too much on std::string while we needed later more information on a given token. That way, we had to change/delete/add some nodes to use more heavily pointers. It took a bit of time to fuse nodes together when designing the grammar. Just a little modification of the grammar resulted in a change of some nodes or the tweaking of the order of precedence of the grammar rules. Some bugs took a lot of time to be found, just to resolve one test on the submition platform.

4 Code Generation

Compared to the other parts, what has been implemented went pretty smoothly. Of course, it is not perfect and the code most probably the most ugly to look at with all the strings constructed however given a bit of work everything which we set our mind into has been done. It has been somewhat challenging to come up with a solution to code the LLVM corresponding to a loop, as modified variables had to be taken into account when we would brench back to the condition. But after giving it some thought and giving ourselves the tools to go around some difficulties, we managed.

3 Limitations

1 Limitations

1. Nested while do not work well yet for the code generation.
2. If an if-else statement can return either a class or its parent, we do not have the ability in our current form to return one or the other. We will have to cast the child as its parent class if we come to this scenario. This can be seen in the test 10 of codeGen, where we print "parent" for both print, as we will always return a parent object and therefore always call the parent method.
3. Casting a child to a parent class will lose the information of the overwritten methods. This is because the function are declared globally and are not part of the structure type. Thus if a variable of type 'child' is casted as a 'parent' calls a method which is overwritten by the type 'child' the method call is the one of the parent because this is the current type of the variable.
4. We still fail the semantic test 71. It seems like assigning to a variable a value coming from the result of a function defined in the parent class is not allowed but we honestly have no idea why. From our understanding of the vsop language, the input is valid and the compiler does a good job.
5. The grammar is not general enough which causes some tests to fail, for instances fancy initializations with blocks and code inside (if-else-...).
6. The "and" of a condition will always compile both side of the and even if the left hand side is false.

2 If more time

If we had more time we would have made some changes in the grammar to enable it to handle situations we didn't think of when we started the project. It was in general not general enough and we had to modify on the run some rules to be able to handle some very specific situations, and some of our remaining problems are still coming from that. We would probably also learn how the LLVM librairy works and either use it for the trickiest part of the code generation, or the one that we simply cannot solve with our current implementation, or we would simply start from scratch and do it all over again using the library. Since we didn't really dig into the library and how it's used, its hard to tell exactly what we would change concerning the code generation if we used the librairy. We would also probably document the code more and make it more reader friendly. Finally, we could try to implement some extensions of the vsop language.

3 If we could start again

As stated before the main weakness in our project might be the grammar that limits us in some situations. We believe that if we were to start again, with the same time, we would mainly improve this section. Indeed, the grammar was designed to be too precise. It is only afterwards that we realized that the vsop language was more permissive. Unfortunately, some flaws of the grammar were revealed only at the semantic milestone. We would perhaps also change the interaction between some nodes and make the `ExprNode` more versatile and the parent of nodes that are somewhat related to it.

Moreover, some idea of structures in our code such as the scope tree which was initially thought and designed to help us ended up restricting us instead which is why we had to add the scope location system. In a way, it sometimes really felt as if we were slaves to the structure we choose. Thus, some choice such as this would not have been made if we had to do it all over again.

4 FeedBack

1 Time spent

On average 5 days per milestone, accounting for average days of work of 8h : 5*4*8 = 160h per student

2 Improvement suggestions and general feedback

student 1 : The project was interesting but it is hard to correct it if something was done badly at the early stages of the code, such as our grammar. Due to time issues mostly (many projects in many courses).

student 2 : I found the implementation of such a big project to be an interesting challenge, as it had to be built in the long run, and satisfying to see the whole take shape. I just hope that the grade of each milestone will not only be dependant on the tests passed, as we worked a lot on syntaxic and semantic analysis but passed no tests. I hope the work submitted will be taken into consideration.

student 3: from my point of view, the real benefit of this project was to work with my team mates on a long-run and more complex project. It changes the usual workflow and forces ourselves to perform more unit tests. I spent too much time debugging at my taste though and I agree with student 1 on the downside of having a project that is hard to get back on the right path.

References