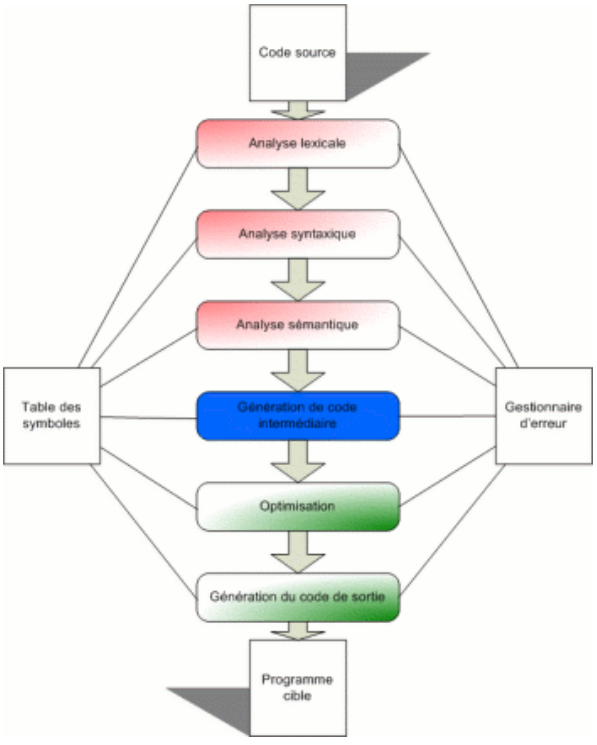
Université de Liège

FACULTÉ DES SCIENCES APPLIQUÉES
Ecole d'ingénieurs et d'informaticiens

LIÈGE université

Project report

INFO0085 : Compilers



Minne Adrien s154340
Maréchal Grégory s150958
Master 1 engineering

Teacher : J. Brusten
Assistant : C. Soldani

2018-2019

1 Choice of language

As we were free to choose the language in which to implement our compiler, the first step of the project was to decide which language we wanted to use. We have heard that the best languages to design compilers are the functional languages, but as we had too little experience with them (only one single course in Scheme), we rapidly gave up on this idea. For a big project like a compiler, we wanted to use an Object Oriented language, that would simplify our life. We were thus left with only a choice between C++ and Java, as we think that the Python language is a bit too script-oriented for such big projects (and we are not familiar with other OO languages). We finally chose Java because this was the language we were the most familiar with. Looking back at this choice at the end of the project, we believe that C++ may have been a better choice because of the availability of the Llvm Library for the code generation [1]. However, the Java language did not hinder our work in any way during the project, and we are quite happy with this initial choice. Moreover, we probably learned much more about Llvm than we would have if we had been using the library.

2 Broad overview of the implementation

Due to the organization of the course, our compiler is composed of 4 steps with little interactions, except that the results of previous steps are used during the next steps. We will here discuss them separately.

2.1 Lexical analysis

To create the Lexer, we used the JFlex library. This is by far the most popular lexer library for Java. It generates a class *VSOPLexer.java*, through the file *vsop.jflex* present in the folder *generator_files*. As this file simply implements the language given in the manual, there is not much to say. However, the following implementation details should be noted :

— Line terminators are not only $\backslash n$, but also $\backslash r$ and $\backslash r \backslash n$ to be more general.

— Line and column numbers are handled by JFlex, which count $\backslash t$ as being 1 character. Thus, files indented with tabulations may get not-so-precise column numbers. We implemented it this way simply because it was easier, we only had to choose where to report the errors.

— A hashmap is used to handle keywords. When an identifier is found, it is matched against the map. If a match is found, then this identifier is a keyword. Otherwise, it is simply a variable identifier. We made this choice to make adding new keywords as easy as possible.

— We also used a hashmap for the operators, for the same reason. In our lexer, anything that is not a literal, a comment or an identifier should be an operator. Otherwise, it is an error. This choice is due to the fact that abstracting operators in a simple DFA rule (i.e., a rule that would not need to be changed each time we want to add a new operator) is not that easy. The other elements (identifiers, strings and comments) are easier to abstract.

— Finally, a stack was used for the nested multi-line comments, in order to easily report the position of the error in case of nested comments that are not all closed.

1. There is no official release for the llvm Library in java, and unofficial releases aren't always fully trustable.

2.2 Syntax analysis

We implemented our parser with the help of Cup. We chose Cup because its known to go well with JFlex that we used for our lexer. This framework generates two files : a symbol interface defining all the symbols that compose our program and the parser class. We also needed to implement a scanner that will return Cup symbols by wrapping the tokens returned by our lexer. All these 3 files can be found in the package *be/vsop/parser*.

In terms of structure, the parser *VSOPParser.java* takes its input (a Symbol) from the scanner *VSOPScanner.java*, which itself takes its input (a Token) from the lexer. The parser turns the sequence of tokens into an ensemble of nodes (*ASTNode.java*) linked together to form a tree, whose root is a program (*Program.java*). The abstract tree structure is described more in depth below. This is summarized in Figure 1.

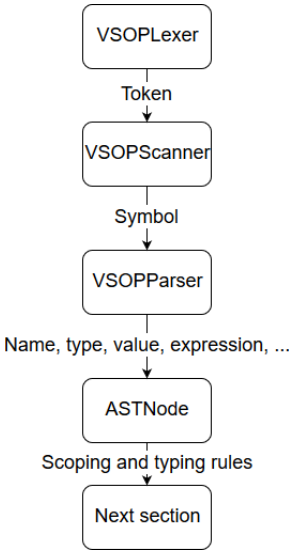


FIGURE 1 – Structure of the parser analysis

The parser is generated from the file *generator_files/vsop.cup*, which translates the grammar given in the manual. It does not require much explanations, here are just a few notable details :

— The line and column numbers used for the errors are the same one than the one generated from the lexer, and thus present the same defect with respect to tabulations.

— In order to describe the errors precisely, we used the *expected_token_ids* of cup. However, it does not seem to report all the tokens that could have correctly been found at some given place, thus sometimes giving empty results which should never happen (if at some point no tokens are possible, that means that the error is somewhere before). The errors given by our parser are usually not that helpful for the programmer (beside for the line number) and can be improved.

— The unary minus is implemented as advised in the (not so) theoretical course, using a normal minus with a right associativity.

— The else keyword is also implemented with a precedence rule. This is needed as otherwise, the grammar would be ambiguous because of the possibility of If-Then with no Else, for instance in *If a Then If b Then c Else d* one need to know with which If the Else is matched. To do this we used the *precedence left* operator provided by Cup.

The *Scanner* does not do anything complicated. It simply takes a token, compute useful information such as start / end location and type, and create a symbol from these, thanks to the ComplexSymbolFactory class provided by cup.

The structure of our AST is represented in Figure 2, and detailed below. In order to implement the different passes, we decided to use a simple OO approach. We heard about other techniques such as the visitor pattern or the functional one, but we believed that by using inheritance, we would be able to avoid code explosion / duplication and to easily implement the needed functionalities. After having done the project, this opinion has not changed. We have learned more in-depth the visitor pattern in other courses, and we still don't think it would have been much easier. The biggest problem of our approach is that the code is spread across all the classes, which is not the easiest to maintain.

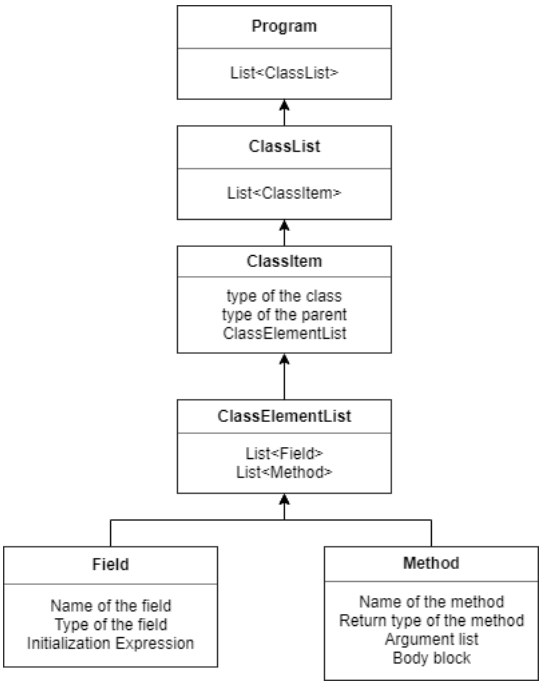


FIGURE 2 – Structure of Abstract Syntax Tree

In our AST, each node is a subclass of *ASTNode*. The AST handles every steps until the Intermediate Representation generation, inclusive. Here we will just describe the general structure of the tree and the important information kept in each type of node, and then have a few words about expressions. Each node keep information about its location (line and column) and a list of children that is used to forward a pass when a particular node is not involved in this pass. All node classes are in the package *be/vsop/AST*.

Our AST structure starts with a program, which represents a list of list of classes. In this project, we only handle single-file compilation thus we always have only one list of classes, however we wanted to be more general as it was not costly. A list of classes is represented by a *ClassList* object, which does not do anything hard neither.

A *ClassList* contains classes, each represented by a *ClassItem*. A class contains information about its type, the type of its parent, and a *ClassElementList*, which contains the fields and methods of this class.

A *Field* contains its type, its name and an optional expression used for initialization. A *Method* contains its name, its return type, its arguments (a *FormalList*, similar to the *FieldList*) and obviously its body, represented by a list of expressions.

Expressions are numerous and there is no point describing all of them in details, as they are quite similar in concept. Syntactically speaking, an expression is not more than an operator with operands, that needs to check some conditions on associativity and precedence. Expressions become more important when checking semantic and, obviously, when generating and running the code.

2.3 Semantic analysis

Our semantic analysis is implemented with the KISS (Keep It Simple and Stupid) principle. We didn't care at all about performances or memory usage [2]. We thus did several passes through the AST, checking some errors and setting some useful variables each time. In a general way, we decided not to run subsequent steps if errors where found during the current step. This was an easy way to avoid some *NullPointerException* and problems of this type.

The first pass is used to generate a *classTable*, i.e. a *HashMap* mapping *String* (names) to *ClassItem*. We needed to do this separately in order to link more easily the scope tables in the next step. Indeed, with the class table filled it is easy to get the scope table of the parent (getting it through the corresponding class item), as any class item knows the name of its parent (from the syntax analysis). The errors checked during this pass are duplicated class (classes with the same name), and inheritance cycles.

During the second pass, we fill the *scopeTable*. This class is a bit more developed than the *classTable*. It is used to store all the methods of the current scope and all the variables, and it obviously contains a pointer to the parent scope table, as discussed above. It also provides facilities to search for a *Method* or a *Formal* (variable), either globally, or in local scope only, or in outer scope only. The last one may seem surprising, but it is an easy way to avoid getting twice the current variable / method when we check for variable already declared or for method overriding. During this pass, we check that fields and methods are not defined multiple times, but in local scope only. We decided to check first that methods and fields are not duplicated in local scope, so that we are then able to check for the outer scope only, without missing an error. This avoid creating false errors by comparing a variable or a method with itself.

During the third pass, we typecheck the types of the expressions arguments, initializers,... The type rules of VSOP being strong, there is not much to say about it. All the needed details are in the VSOP manual. The order in which we go through the tree is not well defined (or at least not on purpose), we compute types as we need them. However, it should not be far from top-down, due to the OO approach.

The fourth and last pass is used to do a final scope-checking of the variables and methods, checking not only duplication but also usage of undefined variables. It also checks for the self variable, which needs to be handled differently.

2.4 Code generation

In Java, there is no official Llvm library. There exist the public Github repositories implementing it, but nothing official. We could not afford the risk (in terms of time) to use unofficial and not always trustworthy releases. This is why we decided to generate the llvm code as a big String. Before generating any code within our compiler, we decided to implement the IO class. The code is in the *language/llcode.ll* file. It also contains other declarations of useful llvm intrinsics. This code is always appended to the generated .ll file. We also need to append a little main function that calls the VSOP main function, as well as initializing the main class. This is done at the root of our AST, in the *Program* class. The idea is simply to call the Main constructor, which will be generated later, and then to call the main function of this newly instantiated object. Then we return the return of the main function as exit code.

Before talking about our IR generation passes, we need to present three classes. All of them are present in the *be/vsop/codegenutils* package. The first one is *InstrCounter*. This class allows to count the instructions, more precisely we can get from it a temporary llvm id (%1, %2, %3,

2. Well, at least a little bit so that we can compile several times the same day !

...) which is automatically incremented between the calls. However, it also allows to count the If instructions and the While instructions and get new labels for them. For this, it returns a *HashMap* containing the label id of the conditional or loop statement, as well as the label ids of important blocks of these statements (if/else/start of the loop/end). By passing this counter as argument of the recursive calls in the AST (and sometimes creating a new counter when entering a new method), we can generate temporary variables easily which are unique and follows the counting rules of llvm.

The second important class is *ExprEval*. This class is used where we need to evaluate expressions. It contains the llvm code that generates this expression, generated with the help of an *InstrCounter*, and the id of the llvm register in which is stored the result of the expression evaluation. This allows to use the results of the expression evaluation from higher in the tree.

The third class that helps in generating IR is *MethodCounter*. This class in fact implements the first of the 4 passes that we do for generating the intermediate representation. The idea is to go through all classes, thanks to the class table generated during semantic analysis, and to setup their methods. By "setup", we mean giving the methods an index in the vtable, taking into account inheritance, and to tell to the methods whether they override or not, by giving them (or not) a pointer to the method they override.

Our second pass is pretty simple, at least in its concept. It has two main objectives : first, creating a vtable (a table containing all the methods of an object) for each class that we will fill later. Second, adding self as argument of each function of each class. Then, and before getting to the core of the IR code generation, we do a third pass to generate the strings literals, that are more easily handled statically in llvm. As there is no operation on strings in VSOP, we decided to generate all strings as global constants, at least for the strings declared in the VSOP code. We also generated a few strings, used for error reporting [3], as local variables, and this is done during this last pass.

So now that the code is prepared, we will be able to start our fourth pass which simply defines all the Structures in llvm that represent our Objects and their vtable. These structure are implemnted with first the fields of the parent class, so that we can cast the object into its parent without problem. The following and last pass generates almost all the llvm code. Once again, we won't detail how all operation / keywords are implemented as they are numerous, here is a list of what is important to note :

— To make the code a bit clearer, we implemented three enumerations, *LLVMTypes*, *LLVMKeywords* and *LLVMExternalFunctions*. These enumerations contains the Strings representing various types, keywords and external functions (such as malloc, powi, printf) in LLVM, it is also useful to avoid typos. We also implemented a class, *LLVMWrappers* which is in some way our own llvm library. It contains numerous functions to generate llvm code for some useful operations. These classes are present in the *be/vsop/codegenutils* package.

— To instantiate Objects, we defined two new methods for each class : a "new" method that allocate memory for the object, and an "init" method that initialize the fields and the vtable of the class. Separating this in 2 parts allow us to chain initializers when there is inheritance. The name of these methods are .New.<Class Name> and .Init.<Class Name> (it starts with a dot to avoid naming conflicts).

— We always use the store and load operators of llvm to load and store varaibles before each expression. While being sub-optimal, it prevents some difficulties (like having to use the phi function in conditionals), and it's something that can be optimized later.

3. In VSOP dispatch on null is the only possible runtime error.

— When generating the code for an expression, the last thing we do is always to cast the result of the expression in llvm into the type we expected. It prevents problems when dealing with classes implementing inheritance.

— The dynamic dispatch is implemented by following the methodology explained in the slides of Cyril Soldani.

— It is not forbidden in VSOP to create a field or variable with type unit (and thus value ()). As it was not detailed in the manual, what we decided to do is basically to ignore those variables when they are defined or set [4]. When a variable of type unit is read, it is simply replaced by its only possible value ().

3 Possible improvements of our compiler

— As discussed in the lexical analysis section, one could better handle the tabulations in terms of column counting.

— As described in the syntax analysis section, one could better handle the parser errors as as the parse error returned by our parser aren't the most helpful.

— Efficiency : as mentioned several times, our compiler is probably slow (even if we don't see it with our little tests) and could be improved much in terms of efficiency.

— Dispatch on null : we implemented customized error message but they are not printed any more because a segmentation fault occur when loading the VTable (error message not customized). This could be improved.

— Identifiers : it would have been clearer to implement two identifiers classes, one for the variables and one for the methods, here we needed to use a boolean isVar instance variable which is a bit ugly when needing to add conditions and so on

4 Possible improvements of VSOP

— Nested comments : the way multi-line nested comments are handled is really strange and counter-intuitive. A comment should be able to contain as much as possible (i.e., everything except a closing string). This choice [5] is probably due to the fact that we want to generate an error if a comment is not closed, but another comment is opened later. For this, the python way (same string to open and close comments) is in our opinion better.

— Let operator : the "let" instruction to declare variables seems a bit strange and counter intuitive to us (at least we never saw something that reassembled this in our (little) programming experience). A classic declaration of variables may be more intuitive to use (but maybe a little more difficult to parse and analyze).

— Unit in variable : there is no point storing a variable of unit type. It has only one possible value and can always be replaced by (). It should be a semantic error to create a variable with type unit.

— Dispatch on null : why enforcing any dispatch on null to generate an exception ? It will happen in any cases if the object is used, but could work when the object is not modified inside the called function. In particular, the IO class contains only functions thus no instance will ever be modified. Calling IO functions without checking null dispatch works thus, in a similar way as static dispatch in Java. We don't think this is something bad.

4. Side effect : a variable of type unit which is not initialised by the user will automatically get the value (), and not null or some special value indicating that the variable is not initialised.

5. This one : (* (* *) I am a comment *)

— Testing script : if the VSOP program does not return 0, this is considered as "an unexpected error" while it is not a problem of the compiler and should thus be OK.

5 Possible improvements of the course

Here our opinions on the course strongly diverge, se we're going to each explain what we thought about the course separately.

5.1 Grégory

I think that there is only 2 possibilities. Either we put more credits on learning compilers (described after), either we have no course at all. Indeed, building a true compiler from scratch, even for a simple language, is a big project. It takes time (around 100h per member including learning tools, submitting). Thus, with only 5 credits, we have little time for the theoretical course and so even less for sessions dedicated to the tools. I believe this is a shame, because the project looks much like SPEM 2, which is interesting but should not be here. It should, in my opinion, look like the course OOP 2, i.e. we should have a complete theoretical course (no project or little projects), with an exam and an in-depth understanding of the algorithms. VSOP could be introduced by explaining how are built the grammars used for lexing and parsing. Here, they were given and we did not really care about how they were created, we simply translated them into flex/cup files. During the following semester, a course containing only the project with sessions dedicated to the tools (again, more in-depth). In this way, one could build the first parts of the compilers with already knowledge about the last parts, which creates better work. Moreover, we would have more time to implement extensions, which is very interesting but not possible for us, unfortunately.

If this is not possible, I just got an idea : why don't present and give the reference implementation to the students, and ask them (for the project) to implement extensions ? I think it would require a deeper understanding of the course (honestly, the grammar being given in the manual for projects 1 and 2, there is nothing to understand except jflex / cup) while requiring less time, and it would probably be more interesting.

Completely unrelated, the scripts on the submission platform should be more compliant. Indeed, we lost a non negligible amount of time implementing a runtime call in our java program during the last part of the project last part, while replacing native execution ./ by a lli call in the submission test script would have solved the same problem. We also needed to get the current directory (not the working directory, the directory of the file) in order to correctly read the language specific files. It took a lot of time while issuing a simple cd command in the test script would have again solved the same problem. It would be nice for the next students to simplify the usage of the platform.

5.2 Adrien

For my part, i am quite satisfied with this course. We didn't get too in depth in the theory, but i don't feel that it's really needed. The only two remark that i can make to improve the course are :

1. Maybe guide a bit more the student on the initial choice of language. A bad choice here can ruin the whole project, its something that should be prevented at all cost
2. As Gregory said, it's sometime more painful to make the code actually *run* on the platform than making our compiler works

6 Time spent on the project

We spent around 10h on the first project, 10h on the second, 30h on the third and 30h on the fourth. Plus 10h for the report and 10h for cleaning and writing some final tests. Around 100h in total per person.