Université de Liège

Compilers: Final report

L'hoest Julien - s150703
Schils Cédric - s152233
Stassen Théo - s150804

1 Choice of tools

1.1 Chosen language

We were free to chose the language we want to implement the compiler project. At first, we were thinking about using Java for its relative simplicity for development and for its object-oriented structure. But after some reflection, it was more appropriated to use a low level language which is performances oriented. Because the conception of a compiler is close to the architecture, it seems natural to consider such a language. We finally considered C++ and the fact that LLVM has an official library in C++ comfort us in this decision. As a retrospective, we do not regret this choice. It was relatively simple to use and it does not prepossess the data we wish to manipulate. The only difficulty were few avoidable segfault appearing due to distraction and were hard to find back.

1.2 Lexer

The lexer chosen was Flex due to the fact that it is a widely used lexing tool in C++. It is also well documented and allow us to use the regex notion we have learned in Theory Of Computation lessons for generating the corresponding automata which will accept or reject the regex. It will execute a given C++ block for each longest recognise define regex. This tool is oriented towards regex but it is possible for it to handle expression such as $\{a^n b^n | n \geq 0\}$ with a mechanism of state stacks to be able to recognise concatenate comment blocks or literal strings (it generates automata similar to push-down automata). Used alongside the Bison tool, it can send back to the parser defined tokens in order to identify the grammar and more.

1.3 Bison

GNU Bison is a C/C++ LALR(1) parser tool (not only). It is able to handle context-free grammar and is delivered with useful functionality such as a fault handler. GNU Bison was chosen because it can work alongside flex. It was sufficient to express the grammar from the vsop manual into a Extended Backus-Naur form notation coupled with correct predecessing values map to each concerned

token. A special attention has been apply to the if/then/else which is a classic problem in parser. Additional grammar rule should have solved it but predececense attibute on the concerned tokens was enough to solve shift/reduce conflicts.

1.4 LLVM

It is possible to implement an IR llvm code generator by manipulating string with our AST. But it is far more adequate to use the LLVM library because we use C++ and that we can just considered and manipulate elementary blocks instead of worrying about the syntax of llvm and implementing type/value/insertion points/registers names etc.

2 Organisation of our solution

First we have implemented a driver for managing the execution flow of the lexer, parser and the treatment of our AST. Our Lexer and our parser work together. The driver will call the parser which will then call the lexer with its defined function in order to get the tokens needed in the analysis of the grammar.

In order to get the lexer working alone, we keep in the driver a value to know in which execution we are (-lex or else). The lexer execute itself with the implementation written for the first deadline and it will send back unique Tokens (SUCCESS/FAIL). Depending of the returned token, the parser will return respectively 0 or 1.

If the execution flow should correspond to the parsing of the source and the generation of our AST (-parse), we make the lexer sending back final and intermediate tokens defined in the parser. Once done, the bison tools proceed to a bottom-up analysis. During its analysis, we map recognised rules to constructors of our implemented structure to construct the AST. Thanks to the stack mechanism of Bison, we are able to keep track of our pointers on object and use them as argument for our nodes until it reach the starting rule where we assign the root pointer in one of the driver fields.

When the AST is constructed and kept in the driver, we apply a visitor pattern to the root in order to either:

- Make it appeared on the standard output (-parse)
- Perform a semantic analysis.

When the semantic analysis is done and successful, the driver perform a visitor pattern to either:

- Print the annotated AST (-check)
- Generate the intermediate code

Once the intermediate code is generated, the driver can either

- Dump the llvm code to stdout (-llvm)
- Generate an executable

3 Semantic analysis

For the semantic analysis a class "prototype" is used to record all the information about the interfaces of the classes in the class. This class record the prototype of each method, field of all classes. We first use a visitor to fill in the class prototype. Then we use a function to check that the prototypes are correct: no redefinition, coherent overwrite, cyclic inheritance. Finally, we use another visitor to check the type and the scope of the variable in the source file.

This visitor is very exhaustive. It checks a lot of specific cases. Some checks are easy to do (for example the two members of a binary expression must have the same type : you obtain the left and right type recursively and compare them) must other need the implementation of some structures. The main is the Symbol Table. It is a list of hashtables containing the variables. Each time you enter in a new scope (in a class, a method, a let instruction, a block of instruction) we add a hashtable to the list. When you exit the scope the front hashtable is pop. The lookup of a variable is simply a check of all the variables in the table. The scope

exit is a destructive method but in the way the visitor is implemented (bottom - up) there are no situation where we lost useful information. The prototype class is also very useful and practical for the type / scope check.

4 Intermediate code generation

Our goal was to implement thanks to the LLVM librairy a code able to dump on the stdout. Sadly, we lack time and mastery of the tool to be able to achieve a working solution on the plateform. The compiler is able to dump the llvm code, but not to generate a native program. Thus, we weren't able to execute the programm of the source. That negatively impact our capacity to debug and assess it correctness

First we instantiate a llvm Module from a context and we instantiate a IRBuilder which will be our main object to generate the intermediate code. Because everything in vsop is expression oriented, it was clear to see how we should implement a new vistor for our AST returning llvm::Value* instead of std::string. Most of the binary expression (add, mul, ...) and unary one (minus, not, ...) directly have their function available in the IRBuilder methods, so it was easy to write down. Only the pow function was harder to find, we send back a call to the llvm intrinsic powi function done with the correct argument. For the If Then Else, a simple PhiNode trick has been used because it was already available in the miscellaneous methods of the IRBuilder. For the while generation, two different basic blocks (loop, loopend) and a branch couple with the (re)evaluation of the condition were used.

In the implementation, each class is represented by a llvm structure. This structure has one field per class field plus one to store the dynamic type of the object. The method of a class is represented by a global funtion in llvm with the name "<class ID>+<method name>". For each class, we also add a llvm function "malloc<class ID>" to allocate on the heap the memory needed for that object.

To handle dynamic function dispatch, we add a field "class" in each class; This field is fill in with an integer when the object is created. This integer uniquely

identify a class. With that, we can know the dynamic class of an object and which method to call. That will be also useful for a key word "instance of" in java or for adding a base method in object to retrieve the dynamic type of the object.

5 What we would have done differently

A thing we should have done differently is to work directly with the environment put in place on the submission platform. On Arch linux for instance, the available version of flex, bison and llvm with pacman are more recent than the ones available on the apt-get, it could lead to some problem in compilation on submissions. Compiling on this environment would have give us a better idea of the actual execution of our solution. For instance, the g++ version of Arch linux include the Boost C++ library but not the version from Debian 9. But has Cyril told us, it is also a good way to consider other environment and not overfit our solution for one environment.

Another things that would help us is to set up an IDE for implementing our code instead of text editors. This could gained us some times in writing the code. But it is not trivial for us to set up a IDE with the integration of Flex and Bison.

We also have made poor choice of name for our AST fields and methods adaptation of the vsop grammar into EBNF notations. This led to some confusion while writing the code but it does not impact the system itself. But this could be easily solved with an IDE for a safe refraction of the names or directly find and replace all with a text editor.

A point which does not impact the results but it's a sign of bad programming is the lack of free/delete commands to delete recursively our AST and the different structures we have initialise during our treatment before the last return. The use of languages with garbage collector (java/python) made us overlook this feature.

6 Remarks about the course

In overall, we appreciate the course and we have not so much to say about it. Just maybe a lack of vsop samples directly available for the students, for the fourth part of the project to see how the code is generated so far.