University of Liège

Engineer - Master

INFO0085 Compilers

Project

MADUREIRA Guilherme - 20175603

CARION Julien - 20174231

1 Lexer

We started by using the *RPLY* library in Python because it seemed to be, on some forums and blogs, one of the best libraries to implement lexers with a good API. However, after some hours of work, we found out that the lack of documentation on this particular library, and decided to go on a lower level library, *PLY-Yacc*, which will also be useful for parsing.

We then created a class *Lexer* where we implemented all of our code. The definition of tokens was fairly easy. We just copied the tokens reported inside the VSOP manual, and created the right regular expressions to detect them in a file string. Nevertheless, we had to use some techniques to correctly handle literal strings and comments. Indeed, to detect strings and comments, we need to keep track of illegal strings and if they stop before the EOF, which would be a error.

For that purpose, we have variables that keep track of the number of opened comments, closed comments and the number of string's double quotes. All of these variables are mainly used to detect Lexer errors surrounding comments and strings. Another technique we used to keep track of strings and comments was the usage of states in *PLY*. Sates have 2 modes: exclusive and inclusive. An exclusive state completely overrides the default behaviour of the lexer. The Lexer will only return tokens and apply rules defined specifically for that state. An inclusive state adds additional tokens and rules to the default set of rules. We only used the exclusive mode. This allowed us to add new regular expression rules inside strings and comments to detect for example nested comments.

In order to keep track of column and line positions for each token, we used internal functions of the lexer, specially for line positions, which we increment at each new line feed, and computed the column position by simply finding inside the file, as a string, the position.

2 Parser

As we used the *PLY* library for the lexing part, we used the *Yacc* parser which comes along and is a bottom-up parser that uses LALR(1) parsing.

Most of the grammar rules, and precedence rules used to avoid shift/reduce conflicts, were copied form the VSOP manual, except grammar rules used to detect parsing errors. Until the code generation part, we were only storing the result of each grammar rule inside a string, which was the standard in the documentation. However this was an awful idea and we had to change all of our implementation of the parsing and semantics part before doing code generation. Finally, we used a tree to store the result of grammar rules.

Each node of the tree contains in this order:

1. name, name of the node
2. children, also nodes
3. values, values important to keep as object identifier names for example
4. position, position (line, column) in the file
5. node_class, name of current class
6. type, type of the node
7. parent (optional), node parent

For every grammar rule visited, we store all of the information inside a new node and return the newly created node to their parent. We also used lists to momentarily store methods, fields, formals, classes and the current class as a simple string. Those lists (and the current class string) are then saved inside the tree at specific nodes. This information will be precious during semantics check and code generation.

The printing of the final tree is done by creating a function for every type of node, which is correlated to the name of the node. By creating a function for each node, be can just travel through the tree and it will print itself.

3 Semantics

Our implementation of the semantics part has 2 parts. The first one (*classCheckerFile.py*) is based on our first parsing implementation where we would return a string. It does a first pass using the old parser implementation with the detection of some semantic errors, but more important, creates dictionaries to store crucial information. We created five Python dictionaries: classes, extends, methods, fields and formals.

The classes dictionary stores all class' names as key and a tuple with their position as value. The extends stores a pair of child class and their parent. the methods dictionary stores a list per class. Each element of this list contains a tuple for every method, containing the name of the method, its returning type and its position. The same idea is applied for the fields dictionary but we store the field's name instead of the method's name. Finally, the formals dictionary has a key a tuple containing the class' name and the method's name, and as value a list of tuples which contains the name of the formal, its type and its position.

We decided to use dictionaries because of its fast access time (O(1)) and its flexibility to store any type of structure as value.

After creating all dictionaries, we used them to already check some errors. It checks if the Main class and main method exist and are well defined, it they are no cycles or problems with heritage, if there are field or method overrides and finally if every type in field exist. After performing these first checks, this part of the semantics returns the dictionaries that will be used in the second part.

The second part (*semanticsFile.py*) will perform type checking using the dictionaries provided at the previous step while doing the parsing. The parsing implementation is the same as the one at the previous section, but we add the type checking and potentially detect semantic errors related to type mismatches.

4 Code generation

For the last part of the project, code generation, we decided to use the library *llvmlite* because it allows to do a complete binding to LLVM using Python, and since we have been using Python for the entire project, it is the more natural choice.

4.1 Data structures

To implement code generation, we need first to store some elements inside a structure. As we have been using dictionaries for the past parts, we will continue to use a dictionary to store information about classes. The first dictionary we will use will be the *classes* dictionary, which will store information about classes. Here is the structure of the dictionary:

- Class name :
  - 'pointer' (stores pointer to class structure in LLVM)
  - 'VTable' (stores VTable of that class)
  - 'fields' :
    - Field name : [index, fieldType] (stores information about a particular field inside the class)
  - 'methods' :
    - method name : [index, methodType, methodFunction] (stores information about a particular method inside the class)
  - 'new' (Stores New function of that class)
  - 'init' (Stores Init function (or constructor) of that class)

Every type and function inside this dictionary are LLVM types and functions, coming from the inner *llvmlite.ir* library.

We also use another dictionary to store basic types such as *int32* or *bool* but in LLVM types. This dictionary will be immutable.

4.2 Implementation steps

When launching the code generation, the first step is to declare and define the intermediate representation of all classes in the VSOP code we want to compile. First of all, we start by always declaring and compiling parents class before their children. It is during this first step that we will build and enrich our dictionary. Then, we will get the LLVM code for every method. During all the process, the LLVM code will be stored inside a single module to which we will add the *object.ll* at the end.

4.2.1 Intermediate representation and declaration of classes

The first class we need to declare it is the Object class. We just need to declare its structure, Vtable and its methods. Llvmlite requires first to define the types of functions (the types of arguments and return value of the function) before defining the function. When all of this is done, we simply add those methods and structures inside our dictionary. We don't need to define the body (instructions) of these functions because there already exist in *object.ll*.

However, for other classes, it is more complex. After declaring a class' structure and VTable, we need to perform a deep copy of its parent's fields and methods. When the new class has inherited of its parent's structure, we need to add its own methods and fields inside the structure, the Vtable and our dictionary. Afterwards, the class also requires a New and Init functions. Compared to the Object class, we need this time to initialise the class' fields, even if the code as no assign attributed to a field, we initialise the field with the type's default value (i.e. int32 type's default is 0).

4.2.2 Body of methods

The process to get the intermediate representation of all bodies is the same. We first create a new scope and a new IRBuilder, which is required for every function. Then we allocate space for the arguments of the method and store then inside our new scope. Finally, we will get the return value by travelling across the tree starting from the method's node. This "tree walk" will also allow to get the LLVM code for the internal instructions inside the method.

4.2.3 Saving the intermediate representation

In the end, we get the module as a string, add the object IR to it, and save the final string into a .ll file. The file can then be compile with the command *llc-xx filename.ll* and *clang filename.s -lm -o filename*, where xx is the llc version we want to compile to.

4.2.4 Trick for *unit* type

When a method has a return value of type *unit*, it is impossible to get the LLVM code for the given module. Hence our strategy to get around the problem. We simply say that the returning type of is a boolean and replace the last operation of the method, which is normally a *()* by a 0. By doing this, we actually give a returning value to the function without messing with the other instructions.

5 Comments

5.1 Limitations of our compiler

Our compiler seems to be working with every basic VSOP needs. One potential weakness of our compiler is the extensive usage of *unit* type inside our code. As llvmlite uses *VoidType* for representing *unit*, which by definition doesn't have a type, when trying to get the string (LLVM code) of the module, we can get some errors. However, we passed all tests in the submission platform, hence it is difficult to know if our implementations has errors regarding the usage of the *unit* type.

5.2 Changes

As we said before, we started by returning a simple string when performing the parsing and the semantics instead of a tree. Not only it wasn't a good idea for the code generation, the step afterwards, that is also made us fail one test during submissions, resulting in some stupid lost points. We would have immediately used a tree and gained a lot of time, which could have been useful to make some improvements.

5.3 Improvements

If we had more time, we would had tried to test all possible usages of the *unit* type in VSOP, which is currently our compiler's most probable biggest weakness. We would had also done some easy extensions such as adding the upper and upper-equal comparison operations, adding new types (int64, float types,...) and finally 2 control-flow operations: for and switch, which are quite useful for programmers.

5.4 Time and suggestions

Per student, we may have worked around 100 hours.

As for suggestions, we do think that the course is well-given, specially with the hands-on project complemented by the theoretical lectures that allows us to understand where to go and what we are actually reading when searching for documentation. Doing this project gives us a good insight on how a compiler works, what are the steps and why some compilers are better than others. The only 2 suggestions we can have, and it comes directly from our experience, is to check the implementation, maybe by having a 5 minutes meeting after the parsing part, before the semantics and also give 2-3 codes to test, other than the factorial and the list codes.

6 Bibliography

- Theoretical course INFO0085 by Pascal Fontaine
- https://blog.usejournal.com/writing-your-own-programming-language-and-compiler-with-python-a468970ae6df
- https://ply.readthedocs.io/en/latest/
- https://pypi.org/project/rply/
- https://www.gnu.org/software/bison/manual/
- https://releases.llvm.org/11.0.0/docs/tutorial/OCamlLangImpl1.html
- https://llvmlite.readthedocs.io/en/latest/index.html