[INFO0085] Compiler assignment

May 2014

L'Hoest Gaetan 1st *Master Computer Science*
Nix Julien 1st *Master Civil Engineer*

Université de Liège

Contents

Chapter 1

Introduction

In this project, we had to create a part of a compiler. We learned how to create a grammar for our own language which is quite pleasant but quite difficult as we had no idea where the main issues of creating a new programming language were. We also chose to implement this language in Python, this choice will be discussed in a particular section (2.7) as we could have misunderstood an advice given during the course.

Then, we had to deal with lexical analysis to produce tokens from expression of our language. With this token stream, we were able to parse those and fulfill the syntax analysis which were quite complex. Thanks to the parsing (depending on the different grammar rule - which were often changed) we retrieved an AST. This AST, composed of non terminal symbols for the nodes and terminals symbols for the leaves, will be very useful.

This AST can be displayed on a real graph and it helped a lot to represent ourselves what we had done. From this, we started the semantic analysis and the intermediate code generation by passing through the whole AST. Then, llvm-ir was generated, which was our final goal. Furthermore, we had to think how to manage the different errors and construct a better and more defensive implementation.

Chapter 2

Development

2.1 Grammar Definition

As we already said, creating a grammar for a new language was quite abstract for us. We mainly referred to the computability course and of course on the compilers first lessons. this being said, we finally created the NXLT language and its grammar which were described in our first report. Of course, our grammar had to change as the implementation of the further part highlights us about some problems (i.e. ambiguity, etc).

2.2 Syntactic Part

In order to do this part, we chose to use the package PLY. Here are some explanation about this and why we made this choice :

2.2.1 PLY Explanation

PLY is a module done entirely in Python. It uses all the parsing tools lex and yacc. Here are a few highlights:

- PLY is done in a very close and classical lex/yacc. Thus, the use of PLY is very similar to the lex/yacc use.

- PLY gives a lot and extensive error reporting and diagnostic information to assist in parser construction. The original aim of PLY was to give easier tools to loose less time in implementation then in understanding the way a compiler works. Thus, in order to help the user, the system tries to identify the most common types of errors made by novice users as we are/were.

- PLY provides full support for empty productions, error recovery, precedence specifiers, and moderately ambiguous grammars.

- As seen in the course, the parsing technique is based on Left-to-right Rightmost derivation which is fast, memory efficient, better suited to medium/large grammars so that we could extend our, and which deals great with syntax errors and other parsing problems. It facilitates the error handling. Currently, PLY builds its parsing tables using the LALR(1) - Look-Ahead Left-to-right Rightmost derivation - algorithm used in yacc.

- PLY uses Python introspection features to build lexers and parsers. This greatly simplifies the task of parser construction since it reduces the number of files and eliminates the need to run a separate lexyacc tool before running your program.

- PLY can be used to build parsers for "real" programming languages as our. Although it is not ultra-fast due to its Python implementation, functional languages as Haskell or Ocaml are more efficient. PLY can be used to parse grammars consisting of several hundred rules (as might be found for a language like C, Python, NXLT, etc). The lexer and LR parser are also reasonably efficient when parsing typically sized programs. Using PLY is convenient and could be used to build a much more complex language as we could do in a more thorough project.

PLY was firstly a pedagogic tools, it becomes used much more and, now, is a serious tools for compilers creation. It has been greatly documented and gives to new person in compilers world a great base and basic examples to fully understand the syntactic phase of a compiler. For all those reasons, we did this choice. It gave us the opportunity to begin this project in a right way which is really important to not to be overloaded after.

2.2.2 Token Generation

Firstly, we had to declare the different tokens that will be used (operators, delimiters, variables...) We had also to define reserved tokens in a different and explicit way, for example the word *if* is not a identifier but a particular and reserved words.

Then we did the binding between the representation of the token in our language and the token itself with its regular expression indeed (i.e. the operator '+' in our language and the regular expression (token) PLUS in the lexer).

Finally, we created particular functions for more complex token which are bound to more complex regular expressions and that we need to have a particular treatment on. For example, we had to define what to do when a comment block is found, what should be ignore, count the lines in case of error, etc. Moreover, when an error is detected here, we only warn the user and the illegal character is ignored and the process goes on.

We have bitten off more than he can chew as we declared much more tokens than what we used but it shows the complexity of a language.

2.2.3 Parsing Generation

Here, we retrieved the tokens and we defined the different rules of our grammar. As said in the computability course, we will create a state machine. From a initial state, we will go to another thanks to each token and depending on the rules till an accepting state is reached. We won't go into much more details as it is not the purpose of this course but it still important to have this in mind.[1] At this step, we transposed our grammar rules but we quickly realized that our grammar should be factorized and remove all the ambiguities. There is not much to explain except that it took a lot of time to finally obtain something satisfying.

After defining all the rules, we had to create our AST while parsing. To do this, we use the class Node which defines a Node and also its possible children (and thus a tree). From this, we had to define all the different Nodes that will appear in our AST. This is done thanks to inheritance in python, we declare class which inherits from Node with their own parameters. Here, we chose to define simple class without specialization in order to stay more abstract but of course, it could be a great idea to defines parameters in those classes (i.e. the while node has a condition and block node).
Finally, we add in the parser file the creation step o the AST by creating all the nodes when we needed too. There are mainly 3 types of Nodes: Intermediate Nodes without parameters only their types, Operation Node with the operator in parameter and the Final Nodes (TokenNode, NumberNode, ReturnNode...) with the token in parameter. [2] The result of the parser is an AST and ,if we print it, we obtain an ASCII-art representation which, for big AST, becomes messy, this is why we use tools to represent it in a better way. See more in the section 2.6

For the error management, we had two choices. The first one was to do some recovery and resynchronization with error rules. With this method, we had to declare rules with all possible errors inside. Indeed, the error token acts as a wildcard for any bad input text and the token immediately following error acts as a synchronization token. For example, if we have a if statement like this *if (condition)* , we have to specify those possibilities :

```
'''statement            :   if error condition )
                        |   if ( condition error
                        |   if error condition error
                        |   if ( error )
                        |   if error )
                        |   if ( error
                        |   if error'''
```

From this, we could know what causes the error but...This seems pretty inelegant. It requires a lot of time and it is likely to contain error. We could be more general but it would still looks like patching some errors and some not.
The second one was to do some panic recovery error. Simply, when an error occurs, the error recovery mode is set, and all the token will be discard until a synchronization point (in our case, it is a NEWLINE token). At a synchronization point, the stack is empty and the normal mod is set. The parsing restart. As this is not as accurate than the first technique, it is quite robust, fast and stay concrete enough. For those reasons, we chose to use the panic error recovery.

[1] Notice that using our Parser, a file names parser.out is created, it contains all the rules and some more information. It also creates the file parsertab which will be actually used for parsing (The file Parser is only use when the grammar rules change)

[2] those are leaves indeed but we say Node as they inherit from the class Node.

2.3 Semantic Part

We chose to do this part by hand as the tools offered were not well documented, not elegant and sometimes as tricky as llvm.
We defined a decorator [3] which will allow us to define methods depending on which node call it on. Thanks to this, we will perform two main things:

- Type checking
- llvm generation, see section 2.4

In order to do the type checking, we did two things:

- We retrieve all the variable in the code.

  It is quite simple, we implemented a function class which contains all the information we need about a function: name, return type, return variable (if any), number of arguments, the arguments and a dictionary, containing all the variable names as keys and types as values, contained in this function. From this, we declared a list which will contain all the function objects. At this step, we only retrieve those information and we set the values of the variables as unassigned. We check the order and the scope of the assignation of the variables (notice that we only have assignation in our language). In order to do that, we use two lists, one which will behave like a stack, another will be used to stock the indexes from where we enter a new scope. With those two lists, we can easily manage the variables we met and from where we have to forget them.

- We perform a type checking in a loop

  Now, we will mainly watch in the assignations and in the function calls to retrieve the variables type. We will loop until no changes in the list of function objects appear.

  Of course, we have to catch all the different errors that may occur. As our language has no type, a lot of different errors exist. We try to manage the most as possible. We check first with there is no function with the same name and with the same number of arguments. We check the order of assignation of the variables depending on the scope where they appear.

  Different features proposed have not been implemented, fortunately, but in the next section 2.5, the main idea will be explained.

2.4 Intermediate Code Generation

The final step was the Intermediate Code Generation. The intermediate language generated is the LLVM IR. This langage is very low langage interpreted by a very low virtual machine. As a homework we were assigned to was to create a quicksort implementation in LLVM IR, we saw some of the rules that were quite systematic. That helped us to understand how to create an intermediate code: find generic ways to generate the code of different situation with the same technique. The intermediate code generation follows the same philosophy as the rest of this project: go through the AST and react adequaly when the different types of nodes are

[3]We use the decorator to perform the visitor pattern : Use the Visitor pattern when you want to add capacity to a composite set of objects and where encapsulation is not important.

encountered. Our intermediate representation may not be the best as the LLVM IR language is quite hard to understand, and some problems like the numbers of the intermediate variables labels forced us to implement techniques working in the biggest number of case as we could think of. We also generate automatically the code of functions allowing us to print an array of integer. We asked ourselves if it was the best way to proceed, and arrived to the conclusion that this is not a problem to define extra function in the intermediate representation because as long as they are not called, the execution of the program will not be affected by them.It can be seen as a "inside library".

2.4.1 Results

An implementation of the quicksort algorithm (see Listing 2.1) has been done. With the function print, we ensure ourselves that the results were correct. We also did the factorial function and the prime predicate. Note that you can ask our program to take several files as input.

2.5 Possible Improvements

- Improving the language possibilities

  Some features are finally not present in the final version due to a lack of time. Multiple assignment is not done (almost the same idea as parallel assignment). We use if and while, no loop for. The possibilities are so large that an exhaustive list would be useless but as we take our inspiration from Python and MATLAB, feature from those language would be great.

- Heuristics in the type checking

  We won't enter into to much details as it is a very specific field of research but we will simply say that in order to compile huge file in acceptable time, heuristics are mandatory.

- Error handling

  It is a big piece of cake... As we have static type, we have different error that are not manage as a function with void as return type could not be called in an assignation. The parameter inside an array expression (i.e. vector[i]) could be longer than the length of the array. Moreover, we do not use the colon to point where a syntax error appear but we counted the colon in the lexer for that, it was not mandatory, thus is not present in the final version.

- Multithread compiler

  As more and more processor have multiple core, multi threading would be a nice way to deal with different file to compile each in a different thread. Then we should ask ourselves how to set up this, how to handle the different part separately...

2.6 Tools to represent the AST

In order to represent the AST, we use Graphviz and PyDot, a python module which is necessary to use Graphviz.
We create an ASCII-art AST and it is transformed thanks to those tools into a readable and beautiful pdf. We let all the graphs generated in the same folder as the report. Typically, from this implementation of the quicksort (see Listing 2.1) , we obtain this representation of the tree on the Figure 2.1

Listing 2.1: Quicksort Function

```
quicksort (vector[], left, right)
{
        if (left < right)
        {
                i ,last = left + 1, left
                while (i <= right)
                {
                        if (vector[i] < vector[left])
                        {
                                last = last + 1
                                swap (vector, last, i)
                        }
                        i = i + 1
                }
                swap (vector, left, last)
                quicksort (vector, left, last)
                quicksort (vector, last + 1, right)
        }
}
```

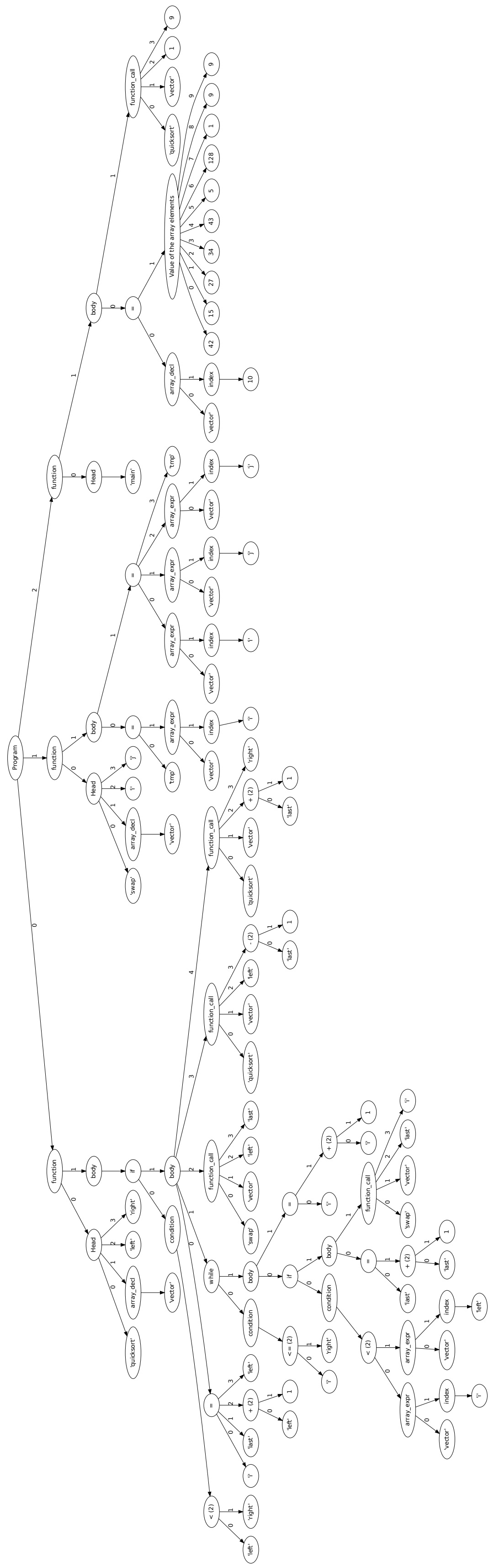
Listing 2.2: Swap Function

```
swap (vector[ ], i, j)
{
        tmp, vector[i], vector[j] = vector[i], vector[j], tmp
}
```

Listing 2.3: Main Function

```
main ()
{
    new vector[10] = 42,15,27,34,43,5,128,1,9,9
    quicksort(vector, 1, 9)
}
```

Figure 2.1: quicksort AST



2.7 Python choice

For this project, we heard that we were advised to use an unknown programming language (or almost). Apparently, some other students told us the contrary. Anyway, as we never had done Python before (or just the basics, until managing a while...), this project had been experienced as a challenge. We learned a lot about python and here are some remarks about it :

- Python is a dynamic language which is Metaclass-based metaprogramming (i.e. oriented object).
- The different types of variable as lists or dictionaries (i.e. hash tables) cannot be copied by a simple equal (it only creates a pointer to the same structure). They have to be called with constructor but, as it is a bit hidden in python (high level of procreation), it is a trap, often
- As Python is oriented object, we used some features from this like: inheritance, decorator, object handling...
- The python documentation is great, thus it facilitates the work but , as we are new, all the different examples in python needed a bit more process to understand
- Python has a strong list use. We use them a lot even for string (i.e. list of character) in order to define a stack explained before.
- ...

As it is not really related to the project itself, we will just add to the project one example of an interpreter without developing. As we were working, after the AST was done, we begin to create an interpreter in python (just to figure out how to go through the AST and make some interesting trials). When we talked about to other students, they found it "fresh" as it gave direct results by performing two steps : the sewing step and the interpretation step. Thus, we let it in the project in order (maybe) to explain that for the presentation.

Chapter 3

Conclusion

This project was fun, we had to be creative, to think on our own and be persevering facing all the challenges. It is a big piece of work but, as we are not fully blocked, it does not seem unbearable. We learned a lot about python, parsing and the different mechanisms which are present in a compiler.

As we have a lot a freedom, it is also hard to do the project in a "right" way and thus to evaluate the work behind those hundreds lines of code. Moreover, there is no "real" end to this project which is disturbing (because no specific applications where asked - except the quicksort).

Finally, we are aware of the tones of work behind tools (i.e. gcc) that we used commonly, there is always work to do. It forces humility. Thanks for reading, hope you enjoyed it.

Bibliography

[1] *Python documentation.* `https://www.python.org/`.

[2] Matthieu Amiguet. *Conception d'un compilateur en python avec PLY.* `http://www.matthieuamiguet.ch/pages/compilateurs`.

[3] David Beazley. *PLY : implementation of lex and yacc parsing tools for Python.* `http://www.dabeaz.com/ply/`.

[4] University of Illinois at Urbana-Champaign. *Kaleidoscope: Code generation to LLVM IR.* `http://www.llvm.org/docs/tutorial/LangImpl3.html`.