Laurence Stolzenberg (260657154)
Maxence Hull (260706895)

Final Report

Due April 10th 2017

Compiler Design (COMP520)
Presented to Alexander Krolik

McGill University, School of Computer Science

GoLite Project : Final Report

1. Introduction

To start off, we believe it would be beneficial to give a brief recap of the Google Go language, and the differences with GoLite, the subset of Go we have implemented in our compiler throughout the course of this session.

According to the website dedicated to this programming language, *“Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.”* To be a bit more specific, GoLang is, in our opinion, a high level programming language that can create very complex programs with very little code. Additionally, it uses an import system to use a large number of complex and specialized modules giving excellent functionalities with very little additional code, which displays our first major resemblance between GoLang and Python. Next, GoLang does not force the use of semicolons at the end of an instruction, they are optional, which is a very interesting, if hard to implement feature.

Now that we have given a brief introduction to GoLang itself, it is important to mention that it would have been incredibly difficult to create a valid and functional compiler for the entirety of the features in this language, given the time lapse that we have at our disposal. Instead of taking on this herculean task, we only selected certain features to implement, making this project somewhat more possible with our time constraints.

Now, we will go a bit more into the specifics of the features we implemented. First of all, we implemented all reserved keywords from the full Go language, including those we will not actually implement, to ensure our code would actually be fully compilable in a reference GoLang compiler. In particular, the Interface and Select keywords are reserved, but not implemented. We also added a few keywords that are not included in the original GoLang including print, println and “append” to facilitate testing without using optional go packages. To go into further detail about our previous statement, we add the “print” and “println” statements to be able to print strings, integers and other basic types without using GoLang imports (i.e. “FMT”).

Next, we included all unary and binary operators in our scanner. We also support all forms of comments supported in the original language. Next, for literals, some minor differences are to be observed. First of all, for a floating point, we can leave off either the integral part or the decimal part of the number, and it would still be a valid floating point literal. Also, we do not have to support scientific notation, which is quite a timesaver for us. Also, for rune literals, we do not have to support the more complex character escape codes, only the more basic escape codes (i.e. ‘\n’). The same basic rules go for the “Raw” string.

Next, for identifiers, our subset of Go only handles identifiers using the ASCII charset, meaning we are able to skip the entire headache behind all the different character sets like UTF-8, UTF-16, UNICODE etc. and all the associate problems.

Additionally, we only handle some cases of the “semicolon insertion rules”. That is to say that adding semicolons on the code we are scanning is simplified for GoLite. the full Go language has two rules to add semicolons after code, in our case, we only need to implement the first of these two rules. The rule we do not implement is simply the possibility to close an accolade “}” or a parenthesis without a semicolon to allow complex lines of code to “fit” on a single line.

Next, GoLite will not support importing packages, so we do not have to implement these features, which are, in our opinion, quite heavy. We do need to keep the package declaration to ensure our code can be compiled by a reference compiler. While on the subject of package declarations, in GoLite there must be exactly 1 package declaration per program (file).

Next, for the top level declarations, we do not have to support constant or method declarations in our subset of GoLang, which is another point that makes this project somewhat easier for us.

For variable and type declarations, nothing was removed from GoLang to GoLite, although there was nothing incredibly difficult to implement, some difficulties were faced here. We will go into more details about this a bit later in this report.

Now, on the subject of function declarations, much bigger variations between GoLang and our subset exist. Namely, we do not support named return variables, we do not support an arbitrary number of return values, we do not support an optional body, we do not support an optional comma after the last parameter and finally we do not support variadic (...) input parameters. All these factors together make our job quite a bit simpler here.

For basic types, GoLite only supports five of them, making our work much simpler. For slices and arrays, we do not support literals, which is another point making this a bit easier for us.

Next, our compiler must support structs, but only a simplified version of what exists in GoLang. To be more specific, we do not support anonymous members, struct literals and a few other specific features.

On the topic of statements, in an assign statement, both sides (assign statement, =, and expression) must have the same number of arguments in GoLite.

There are a few other points that are specific to our GoLite subset of GoLang that are designed to complement what was already previously mentioned and to make this project possible in the time frame we are given.

1.1 Structure of our report:

Now onto the structure of our report.

Previously, we went into a bit more detail about GoLang and the subset of which we implemented in our compiler. Next, we will go into more detail about our language and tool choices, that is to say the language our compiler outputs and which tools we used to develop our compiler as well as any other tool we used during our project.

Following the tool section, we will go into more detail about each phase of our compiler, from the scanner we developped for our first milestone, to our code generator which we just “completed”. We will give you a brief overview of how we implemented these, how we tested them as well as specify our major design decisions.

Next, we will move onto our conclusion, were we will mention any major decisions we regretted and would have changed given the chance, as well as some more feedback over this entire experience.

Finally, we will have a small section outlining the contribution of each of the two members of the team for the project.

2. Language and Tool Choice

We chose SableCC instead of flex/bison for several reasons:

- One of the team member had already worked with SableCC for a prior school project.
    - Although that team member later found out that he had worked with an earlier version of SableCC, SableCC 2, which had some pretty significant differences with the latest version.
    - Nonetheless, we feel as though this prior knowledge gave us a small advantage to actually develop our compiler.

- We both have a deeper knowledge of Java than C or C++.

- We both thought that Java would be easier to debug (compilation errors are often detailed enough to be useful) and use in the context of a large project (no memory management, lots of native complex functionalities…)
    - Also we avoided the possibility of getting seg faults, which we heard plagued the other teams.

In fact, SableCC would still be our first choice today, not because it is perfect tool (more on that later), but because having a strong knowledge of Java was indispensable in this project given its scale.

Next, we used a Java Integrated development environment (IDE), NetBeans, for part of our project. We chose to use NetBeans as it has an excellent built-in debugging engine which allowed us to find and correct a huge number of problems that would otherwise have been very difficult to track down. In particular, the capacity to look inside our data structures in real time while stopped in our code allowed us to find some pretty strange issues with our symbol table and type checker.

Next, We chose Python 2.7 as the target language. It seemed like a good choice for multiple reasons:

- Like Go, Python natively includes type inference which makes our job easier for some statements like variables declarations. Moreover, Python syntax allows us to group some complex cases (like variable declaration, assignment and short variable declaration) into a single solution (since it does not make distinctions between those cases).

- Python's system library natively includes many of GoLite's operators and expressions (like append, print…).

- Many GoLite statements (if/else, for loops…) and expressions can be easily expressed in Python. In fact, Python and GoLite syntax are very similar.

- Debugging Python programs is easy: error messages are often very clear and no compilation is needed.

However, choosing Python as a target language also have some drawbacks:

- Explicit typing and type declarations are not allowed in Python. However, given the fact that many type declarations refer to basic types (string, integer…), some parts of this problem can be easily resolved.

- Structures do not have a strict equivalent in Python (but they can be simulated through objects).

- Arrays do not have a strict equivalent in Python (but can be simulated through lists).

- Since Python is an interpreted language, it is slow (compared to compiled languages such as C).

- Some statements (like switch) or operators (like binary bit-clear) are not present in Python 2.7

In conclusion, Python seems like a reasonable choice for 80% of GoLite. But some cases (like type declaration and structures) will be much harder to handle.

3. Scanner

The Scanner is of course the first phase of our compiler. Its role is to transform a text file (strings/chars) into tokens with a predefined set of regular expressions forming a "grammar". Regular expression syntax in SableCC looks a lot like flex, that is to say its syntax is very similar, as you can see below.

Symbol	**Meaning**
c	character
	empty string
[ ]	Range
|	Alternation
*	Kleene closure (0 or more)
?	Optional (0 or 1)
+	1 or more

We extensively used the helper section of our sableCC grammar in order to make our regular expressions easily readable. That is to say we defined as many basic rules as possible in the helper section, that we could then use in the token section, making the rest of our grammar much cleaner and shorter than if we had skipped that part. As an example, we were able to make a whitespace helper that contained space, tabs etc., to avoid having to name each of these later on when we meant a whitespace.

We took advantage of matching rules (as defined by Étienne Gagnon is his thesis: “For a given input, the longest matching token will be returned by the lexer. In the case of two matches of the same length, the token listed first in the specification file will be returned.”). We first declared all the keywords, operators and separators. Then we declared runes, the different types of integers (octal, decimal..), the different types of string and finally identifiers. Basically, we ordered the tokens from the simplest ones to the more general (the “identifiers” regular expression is a “catch all” token).

In order to only keep useful tokens, we ignored some of them. There is a dedicated section for these “ignored” token in sableCC, fittingly called the ignore section, where we included the following: comments, end of lines and blanks.

Tokens are represented by Java classes and in SableCC are all derived from one general token class.

We did not have major difficulties in this phase, except for multi-lines comments (but we fixed the problem after milestone 1, however it cost us several points).

We basically included the test phase of this section into the tests we ran for the next phase of our compiler, the parser.

4. Parser

Here, we can see that the SableCC syntax used to write a grammar is very straightforward:

Terminals	Tokens declared in the dedicated section
Non-terminals	Productions declared in the “Production” section
Productions	represented by the following pattern: name = production
Start Symbol	The first production

We also used some other symbols like the Kleene closure *, alternation between rules | or the optional ? (all very useful to make the grammar more readable).

During this phase, we really followed the Go documentation: https://golang.org/ref/spec. For most of the cases we encountered, it offered a good approximation of what productions rules we had to implement, often giving something very close to the actual regular expression that was required for the production in question. However, we encountered our very first major difficulty: optional semicolons.

In fact they are completely optional in GoLite, but the grammar needs to have some in order to avoid conflicts even if they never appeared in the Go official specifications: “*To reflect idiomatic use, code examples in this document elide semicolons using these rules.*”. We included too many of them in our grammar (more on this problem in the “weeder” section), which gave us quite a bit of trouble.

Since all basic types can be overridden in GoLite, they appear as identifiers in our grammar. From the same reason, casts and function calls have exactly the same syntax. What we did to separate function calls from casts was keep a list of the basic types used in GoLite, and if a function call had exactly the basic type as a name and a single parameter, we considered it a cast. If a user declares a function named float64 with a single parameter, well then he will obviously face some issues. At this point, we could have added an optional check to our “cast detector” to see if a function had been declared with the same name and a single parameter, or if the basic type was no longer a basic type, and disregard the cast, but we pretty much ran out of time for the edge cases and kept our current approach, which in our opinion, should handle 99.99% of cases.

Even if we finally resolved many of the conflicts after milestone 1 (basically, we break some productions into many in order to force SableCC to choose the right one: a good example is the statements rule who had been divided into several productions rules), one of them is still problematic: the empty statement. We never figured out how to perfectly handle it (we still have a shift-reduce conflict), but making optional (“?”) most of the calls to the “statements” rule resolved a part of the problem.

Comparing to flex/bison, SableCC does not have precedence properties (I.E. expression statements were broken into many pieces and “levels” to respect the given precedence). This made our grammar larger than the ones produced with Flex/Bison (but it was excellent practice for the midterm exam).

As for tokens, SableCC converts productions into classes. These production classes are composed by other productions classes or token classes. Each alternation of a production rule is a class derived from the “main” production rule. To give an example, the statement production rule has many alternations: if statement, declaration statement… Each of these cases are represented by a class which derives from the “statement” class. This hierarchy will be extensively used throughout our next phases.

SableCC 3 introduced the possibility to create an abstract syntax tree (AST) directly from the SableCC file. That is to say, we can greatly simplify our grammar and link each original production rule to a new one. Although this functionality has been more than useful in order to the type checking and code generation phases, we found it quite difficult to apply. The original grammar became more difficult to read and much longer. Moreover, error messages due to AST construction are very abstract: debugging them was a painful task given the size of the grammar.

Additionally, while changing our grammar to an AST form made our productions much easier to work with in the later phases, it broke certain functionalities from our earlier phases. In particular, the pretty printer had to be completely redone, as it was entirely different at this point.

Next, to test our parser, and at the same time our scanner from the previous phase, we made a large series of tests, designed to check each, or at the very least most, of the functionalities described in the Go Language specifications and included in our GoLite subset. These tests were designed to be scanned, parsed, and have our compiler output either a valid if the input file was valid, or invalid with a specific error if an error was contained. As we had mentioned above, we had some issues with semicolon as well as empty statements which was reflected in our tests results (and the grade for our milestone 1). Nonetheless, we feel like we did a pretty decent job for these first phases.

5. Weeder

Switch case:

A first weeding phase was mandatory in order to refuse a program if more than one default case was observed in a switch case. We used the SableCC “depth first adapter”, which is a module that offers an elegant way to traverse the AST. For each node three actions are possible: make a stop when you first visit it, make a stop when you leave it or stop the depth traversal at this point.

In our AST grammar, a switch statement is composed by an optional statement, an expression and many “switch bodies”.

```
switch_stmt = [opt_stmt]:statement? [condition]:expression? [body]:switch_body* ;
```

A switch body can be a general case or a default case:

```
switch_body =
   {case} [expressions]:expression* [statements]:statement* |
   {default} [statements]:statement* ;
```

Basically, for each switch statement we checked if at most one default case was present using Java’s “instanceof” (each production is represented by a class).

Semicolons

As mentioned before, semicolons are optional in Go but our grammar needs them in multiple cases to correctly distinguish some production rules (ie: to avoid conflicts). To do this, we enforce the following property from the Go Language specifications:
“When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is

- *an identifier*
- *an integer, floating-point, imaginary, rune, or string literal*
- *one of the keywords break, continue, fallthrough, or return*
- *one of the operators and delimiters ++, --,),], or } “*

Using Laurie Hendren code as an example, we added a new weeding phase to introduced a TSemicolon token each time it was needed. This weeding phase occurs before the CST construction and is represented in SableCC by a filter tool, present in the lexer class. However, in this phase, we must keep the ignored tokens (like end of line, comment and blank) to correctly apply the above rules. Because of this, for the first milestone, we missed some of the semicolons insertions which caused some pretty major issues with the test sets used to correct our milestone.

As an example: *TIdentifier Tequal Tidentifier Tcomment* TEol was not correctly seen as a case where a semicolon is necessary. In order to resolve this problem, we relaxed our grammar removing all unnecessary semicolons but we did not find a general solution. We supposed SableCC has some tools to precisely taking care of ignored tokens but we did not find it.

The tests for this section were included with those for the previous phases. We simply added a few specific test cases to check if we invalidated programs containing a switch case with more than one default case. Additionally, we did some basic, if obviously insufficient tests for our semicolon insertion cases to ensure we correctly distinguished the cases that would otherwise have had collisions.

6. Symbol Table

Next, we will move on to the “Symbol Table” generation phase of our project. the symbol table itself is a class consisting of:

- A stack of hash tables containing the types and variables contained in a scope and all variables and parents from its parent scope.
- Each Hash table represents a symbol table as seen during the course
- A Multitude of methods (add a new variable, a new type, check if a type exists, RT function...)

All of these Hash tables have, of course, the same structure:

- a name as a key (as an example a variable name)
- a SymbolTableEntry as a value

This SymbolTableEntry contains many useful information:

- a category to know if the entry is a variable, a type or a function
- a string that represents the type of a variable or the referring type of a new type.
- a boolean field to know if an entry can be shadowed or not by another entry with the same name.
- a hashmap with the same structure that represents a struct fields (if the entry is a structure)

One of the most challenging part of the SymbolTableEntry class to design was the type field. It is a string with specific nomenclature:

Type	**Nomenclature**
Basic type	identifier (example: int)
Array	(”[“ digit+ “]”)+ identifier (example: [4][5] int)
Slice	(”[“ “]”)+ identifier (example: [] float64)
Struct	(type “,”)+ (example: int, [3]float64, bool)

With this structure, it is very easy to know if a type is a slice, an array or struct with regular expressions. We can also retrieve the size or the number of dimension of an array.

Each time a new hashmap is needed (as an example, when we entered a new block) a new one is created. It is filled with the entries of the most recent hashmap (ie: the hashmap at the top of the stack). This new symbol table is removed of the stack at the end of the block. A initial hashmap is created at the object construction with all basic types (float64, rune, bool …).

RT is an important function: for a given type we check in the current symbol table if the entry exists and then retrieve its type. If this type is a basic type, an array type, a slice type or a struct type we return this value, otherwise we recursively call RT with this type.

7. Typechecker

Next, we will delve into the subject of our typechecker and the type checking phase of our compiler. Just like the weeder, we used SableCC tools to do a depth first traversal of the AST. We extensively used the symbol table, especially these functions:

Function	Description
addVar(String name, String type)	Add a new variable into the symbol table. Check if the name does not already exist in the table as a key, otherwise reject the program. If the entry already exists but can be overridden (ie: it can be shadowed) then update the entry in the symbol table, otherwise reject the program.
addType(String name, String type)	Same behavior as addVar, but obviously for a type instead of a variable.
addTypeStruct(String name, String type, HashMap<String, SymbolTableEntry> hm)	If the new type is a struct, then we used this function. It has the same behavior as addType but create a special “struct” entry with a hashmap representing all its fields.
addVarStruct(String name, String type, HashMap<String,	Same behavior as addTypeStruct

SymbolTableEntry> hm)	

This brief overview of the major functions inm the symbol table class gives more insight into how we handle different cases. We access the symbol table data structure through well defined methods in order to take care of all possible different cases.

We also created several methods to handle typechecking correctly including getType(String identifier) to retrieve the type of an identifier and many utilities functions like isArrayType(String type) or isStructType(String type) etc.

As an example, we will take a look at the type checking for expressions, as it is one of the most difficult cases in our opinion. First, we created a recursive method: getExpressionType(PExpression expression). PExpression is the node representing an expression, generated by SableCC.

If the node is:

- A literal expression such as a float64 or a string:
    - Then return a string representing this literal
- An identifier:
    - Then retrieve its type in the symbol table (the symbol table object will return an exception if the identifier is not present)
- A parenthesis expression:
    - Then call “getExpressionType” again with the expression inside the parenthesis.
- A binary expression:
    - Then retrieve the types of the two expressions. Then check, given the operator, if the operation is possible. Finally, return the type of the whole expression given the operator.
    - We used many helper functions for case like this one, including isNumerice(String type) or isComparable(String type).

- An access to a struct field:
    - (example: identifier.x)
    - Then we check if the identifier refers to a struct. If it is the case, we get the entry corresponding to this struct and get the corresponding type of the field inside the hashmap (as mentioned before, struct entry are special: it contains a symbol table with all these fields inside).
- etc.

The others cases follow the same strategy. Each time we encountered an expression (as an example in a if statement) we used this function to get its type. At the bottom line, type checking is always a string comparison.

Next came another fairly complex part of type checking, function declarations and function calls.

When we have a function declaration, we have a separate data structure that contains the identifiers for our list of function call, as well as a special list for its parameter types. To be more specific, if a function declaration has five parameters with different types, we have a list of these five types associated to the name of the function in question. If types repeat, we still keep the type twice in our list to further facilitate function calls. We also memorise the function declarations return type, so in our type checking routine described above, we can return this return type, if it is not void, and correctly verify if our function call is used correctly.

To give an example to clarify our previous statement, if we assign the result of a function call that returns an int to a variable, that variable must be an int as well.

Any time a function is called, we verify if the correct number and type of parameters are included in that function call.

Finally, for a cast, using the methodology described above to differentiate a cast from a function call, we keep in mind in our main type checking routine that a cast results in the type specified in the cast "name", and return that type when our routine receives a cast as input.

To test this phase of our compiler, we also obviously tested our symbol table at the same time. We created a fairly large number of checks designed to verify as many cases as possible of valid and invalid programs testing the different scenarios we might face when type checking. From invalid casts, to casts to structure types, which are not allowed in our code, to casts using an alias of an alias of a basic type, which are allowed, we tried to think of as many edge cases as possible here.

Once again, judging from the results of the test sets used to correct our milestone 2, I would say that we forgot a decent number of cases. Nonetheless, we were able to fix most, if not all of these errors.

8. Code Generator

Python as a target language

As we mentioned above, we chose Python 2.7 as our output language for a range of reasons that made us think it was the optimal choice in the scope of our project. We did learn, during the course of the last two milestones, that this choice had some complications attached to it as we have briefly mentioned in the previous sections.

AST traversal

Our code generator adds a full pass through all of our Abstract syntax tree, generated throughout our previous phases.

We go through the entire thing, node by node, and attempt to generate the equivalent of each statement, declaration etc. from our source GoLite Code to our output python code. Obviously, some things have larger differences that made the translation much more complicated, while some concepts from Go simply did not exist in Python.

A good example of this is the package declaration, which is something that simply does not exist in python. Python uses the actual filename to "name" or identify a class, but in our case the filename is defined by the input filename, as defined by our project, so our package declaration was simply discarded.

Some other differences between languages that we encountered, and toiled to solve, are specified below.

Amongst other problems, we cannot declare a new type in Python. In GoLite, each type can refer to:

- a basic type (int, string, bool…)
- an array or a slice (detailed in next section)
- a struct (detailed in next section)
- another type

Type and variable resolution

In numerous cases, we needed to be able to resolve the basic type behind an alias, verify the existence of a variable previously declared or any other number of things. At this point, we had the option of including all the code from our type checker in our code generator, which initially seemed like a very bad idea. After some thought, we found another interesting solution.

We decided to keep all of the "Scopes" created in our previous type checking phase, and identify it by hashing the parent node responsible for the scope creation. That is to say we found the first parent outside the scope creation, so either a switch, an if instruction, a function declaration etc., hashed the node in question and used it as a key for a hashmap to contain our scope. This, surprisingly, worked very well to access the correct scope during our code generation without having to copy any of our type checker code into our code generator, as we just pass the resulting data structure from one phase to the next.

Type declarations

We easily initialize a variable with a basic type:

GoLite Code	Python Code
var i int	i = 0
var i string	i = “”
var i bool	i = false
var i float64	i = 0.0
var i rune	i = 0 (runes are handled like integers in Python)

If the type refers to another type, we just used the RT function in the symbol table to get the basic type.

Structs

There is no such thing as a struct in Python. In order to simulate them, we decided to create objects. Each time we encountered a new struct, we add a class at the top of the Python file. Each field of this class corresponds to a field in the struct. We used the same code as the one used to handle variables initialization to initialize each field.

GoLite Code	**Python Code**

type point struct {
    x, y int
  }
	class point :
	        def __init__(self):
	                self.x = 0
	                self.y = 0
	        def __eq__(self, other):
	                return ((self.x, self.y,)) == ((other.x, other.y,))

var a point	a = point()

In GoLite, structs can be compared with == and != operators. We used a Python property to easily handle this case: we override the method __eq__ internally used by Python to make comparisons. Our implementation has still some limitations, including the fact that for our implementation, structs inside a struct are not permitted.

Arrays and slices

There is no array type in Python (ie: a list with a fixed size) but it natively has lists. In order to have the behavior in python match what we can observe in GoLite, each array declared in GoLite is directly initialized with the correct number of elements as a list in Python. If the the program access' an index exceeding a list's size, Python will throw an exception. No matter the type of the array, we just have to put 0 in the Python's list (since it does not have any defined type): a nice shortcut for us.

GoLite Code	Python Code
var y [3]num	y = [0, 0, 0]

Multidimensional arrays are initialized with nested loops (basically one by dimension). This has the strange effect of making our compiler pretty slow to compile a program for very large arrays.

Slices have the same behavior as Python lists. Append also exists in Python making the code generation very straightforward.

Casts

Casts were something of a non issue in python, as we have pretty complete type checking in our compiler to prevent any impossible casts from occurring, and python natively and inherently handles any basic cast, so we pretty much had nothing to do here other than print the cast itself. After all the difficulties mentioned above, this was a very pleasant surprise.

Switch

We have replaced switch statements by a series of if statements (first switch) /elif (all other cases) / else (the default case). The generated code is more verbose than GoLite but it seems like the only way to actually implement a switch in Python without importing external libraries. We still have a problem concerning break statement inside a switch since it is not allowed inside an if statement in Python.

9. Conclusion

So, to conclude this project, we will go over what we experienced during the course of this session and throughout this project. I can briefly summarize it with a single word: Fatigue. Combined with another course, the workload was absolutely enormous and very difficult to complete at times.

Nonetheless, we feel as though we learned a great deal completing this project, not only about the different phases and inner workings of compilers, but we also have a better idea about some of the details and quirks of common programming languages we use and why they occur. We also had the chance to see how bytecode is optimized, and therefore learned that if we try to optimize our high level code, we might end up fighting the compiler's optimizer and actually make our code slower!

This was a very rich and informative, if incredibly tiring session. Thanks again and have an excellent summer!

10. Contribution

First Milestone:

As we have included in the report given with our first milestone, here is the separation of duties for the first milestone:

“We tried to be as fair as possible while separating the work for this project. Maxence built the “Backbone” of our grammar, while Laurence designed a number of valid and invalid tests for the compiler. Next, both members separated some of the more complex features to implement. As an example, Maxence handled the “for” and “while” loops, while Laurence handled the function declarations, function calls, and “Switch” statement. Finally, while Maxence handled the “AST”, Laurence worked on this report.”

Second Milestone

Next, for the second milestone, the separation of duties, as written in the report included with the second tag of our code:

“

Maxence:

- Majority of the basic type checking code.
- Implemented the scope / stack code.
- Created most of the type tests.

- Improved weeding phase (concerning return statements)

Laurence:

- Implemented the flags and information printing.
    - Implemented dumpsymtable and dumpsymtableall.
- Began partially recoding the pretty printer.
- Function declaration type checking
- Return statement type checking
- Function call type checking
- Cast statement type checking
- Created a few type tests.

“

Third Milestone

Laurence started off creating the base for the code generator and implemented the first basic functionalities and ensured we could generate a basic functional python script. Most basic operators, literals and a bunch of other stuff was implemented here

Maxence resolved most of the problems shown by the official set of tests from milestone 2 at this phase and wrote most of the port for this milestone.

Fourth Milestone

For the final milestone, most of the work we actually did together in the grad students lounge. Since most of what we had left to do was quite complex, such as figuring out how to mark the scopes from the type checking phase in a way that we could later access it reliably in the code generation phase, we generally worked together for everything in this phase.

Maxence did most of the tests.

Final Report

We produced the final report in Google Docs, allowing us to easily and fluidly work together to write the report.