Justin Domingue – 260588454

Daniel Pham – 260526252

Serguei Nevarko – 260583807

Final Report

COMP 520

Presented to Prof. Laurie Hendren

McGill University

Friday, April 15, 2016

Since the dawn of time, Man has pursued happiness by crafting compilers. In this project, we were asked to write a compiler for a subset of the Go language called GoLite. The present document is divided into two main sections: front end and back end. Front end includes the scanner, the parser, the pretty printer, the syntax tree and the weeder while back end elaborates on the symbol table and symbol pass, the type checker and the code generator.

LANGUAGE AND TOOLS

We decided to use flex/bison because that's what we used in the assignment and while SableCC seems like a promising avenue, we were already comfortable with flex/bison. We also decided to generate C++ code as it seemed like a close analogue to Go.

TEAM ORGANIZATION

Here is a quick overview of the work done by each member (do note that peer programming was extensively used throughout the project):

Dan:

- Lexer
- Parser
- Weeder
- Type checker
- Symbol table printing
- Code generation

Justin

- Lexer
- Parser
- Tree
- Weeder
- Symbol pass
- Code generation

Serguei

- Investigated automating tree generation
- Pretty printer + AST tree printer (for debugging purposes)
- Symbol table using red-black tree

- Intensive programs written in Go

Front End

Scanner

As discussed in the Language and Tools, we used flex to generate a lexical analysis of the input Golite code. Most rules are basic regex similar to what we have seen in class and done in the assignment. However, some intricacies of Go(lite) needed to be parsed at a finer level. To do so, we used a regex to capture the first few characters ("/*" for comments, " ' " and " ` " for comments, interpreted strings and raw strings). We then analyzed the input character by character, checking for allowed escapes and saving to *yytext* as needed. This implementation was suggested to us by the ANSI C grammar.

Parser

The parser was completely handwritten from scratch in Bison following the Go(lite) specification document. It was rather similar to what we did in the assignments. It should be noted that while we used to allow optional semi-colon, this feature was turned off to comply with the Golite specification. Also, parenthesized types are allowed, to be inline with the Go specs (although the reference compiler does not allow them). https://golang.org/ref/spec#Types

To make the grammar simpler, we allowed over-generation to mitigate reduce/shift-reduce conflicts.

1. In function calls, the id is an expression to allow nested parentheses.
2. Function declaration with returns do not need to end with a terminating statement
3. In function declarations, the number of arguments in a return statement doesn't have to match the number of return arguments defined in the function signature
4. Continue and break statement can be anywhere in the code
5. Expression statements can be anything.
6. Post conditions for for loops can contain variable declarations statements
7. In short variable declarations, a list of identifiers cannot be distinguished from a list of expression, leading to shift reduce conflicts. Therefore, we use an exp_list on the left side instead of id_list.
8. Left-hand side of assignment, increment and decrement statements doesn't have to be an lvalue
9. A switch statement can have multiple default cases
10. In a variable declaration and assignments, the number of identifiers doesn't have to match the number of expressions
11. Division by 0
12. '_' can be used as value (a = _)

Pretty Printer

The pretty printer follows has a standard implementation. Runes are escaped. It should be noted that we support indentation.

Tree

Serguei built a tool that read the grammar file and automatically generated the code for the tree, the pretty printer and the updated grammar file (where the code for the tree generation was inserted). It

didn't work so well because the generated tree was following the grammar too closely. We ended up with a concrete syntax tree which was much more difficult to use than an abstract syntax tree. There was an idea of eliminating all the nodes that didn't carry information to make it more like an AST but we ended up just writing it up by hand just like in the assignments.

Had we been able to write the tool to convert our CFG to an AST automatically (without specifying any node merging rules) we would have had the ability to rewrite the grammar with the greatest of ease without needing to worry about rewriting the code of the tree.

A particularity of our abstract syntax tree is that we have a node that is that our EXP (expression) node has a kind for every operation (binary, unary, or other) but there is a single structure for binary expressions and a single structure for unary expressions. It made the AST shorter and more readable.

Weeding

Every weeding prospects identified in the Parser section were dealt with. However, the following over-generations have a few particularities:

- 1. (Function calls) will be weeded out by the type checker
- 11. (Division by 0) used to be weeded out – but after talking with Vincent, we simply allow the possibility of having runtime errors
- 12. (Blank ids) can only be used on the left of assignments and short variable declarations. They also aren't allowed as function ids since Golite does not support function literals. The same reasoning is applied to struct declarations.

Back End

Symbol Table

The symbol table is slightly different than what was shown in class. The concept of layers for scoping is exactly the same and it has the same functions, but instead of using hash tables we are using red-black trees. The idea was to try something new. Unlike hash tables that need a good hash function to be efficient, a red-black tree is always fast because the runtime of all searches and inserts are at most O(log(n)) where n is the number of elements in the tree.

Symbol Pass

Symbols are typed as: variables, type aliases, functions or inferred where inferred has its type determined at type checking.

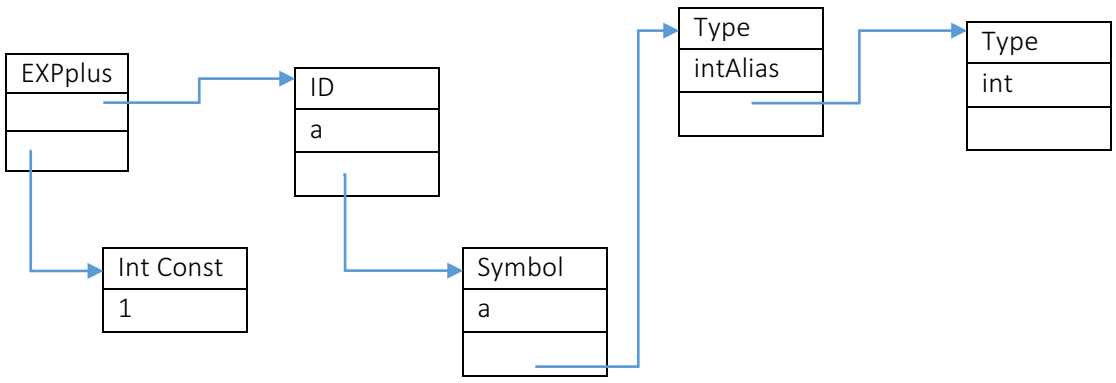
Type Check

These follow the GoLite specifications.

The type checker works by creating links between the relevant nodes. If the type is of base type, it links the type that of a predefined node. If it is an alias, it links the type to the type declaration. (The declaration must have been declared before it was called). One particularity is the fact the the type of an ID is stored in the symbol it refers.
It is here where casts to non basic types are fixed in our tree as we know the types associated with the symbols.

Part of the tree after running the type checker on:

```
type intAlias int
var a intAlias = 2
_ = 1 + a
```



The `Type intAlias` node is the same node that was created at the typedef.
The `Symbol` node is the same node that was created during the symbol pass.

Computation Intensive Programs

primeNumbers.go:

This program generates the i'th prime number where i is a number that is hardcoded in a variable. We will change this number to be able to choose the computation time that we want.

The algorithm is quite simple. For every number we check if it can be divided by the previous prime numbers that we found. We only check with prime numbers that are less or equal to the upper bound of sqrt(number). Since we don't have the sqrt() function we just keep track of it using a variable and its square value.

queenPuzzle.go:

This is the famous queens problem where we have a chessboard of size **N**x**N** and we have to place **N** queens on it such that no two queens attack each other (horizontally, vertically or diagonally).

In our program this problem is solved using simple recursion. To have more control on the computation time we just set the variable **N** to the size that we want.

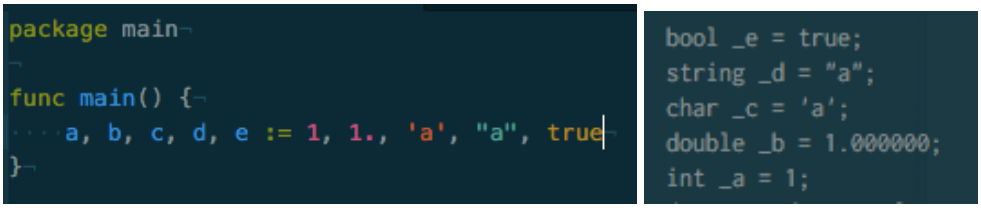
Code Generation

Target language: C++

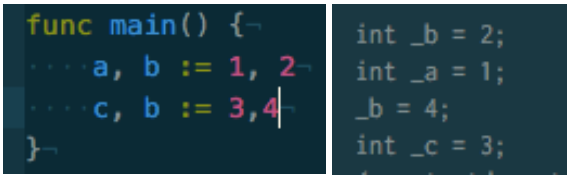
We wanted a fast, highly optimized target language. By choosing C++, the code generation is quite simple. In fact, Go-lite and C++ share many constructs and for the basic type systems are compatible meaning that we don't have to handle these ourselves. The generator operates in the same way the pretty printer does, in that it will traverse the tree from top to bottom printing appropriately.

Some code generation:

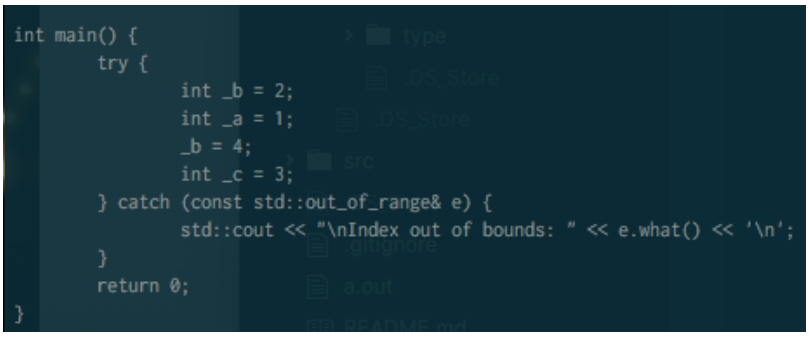
- We use the base types in C++.



- Short variable re-declarations are handled correctly. This is possible as we tag re-declarations during the symbol pass.

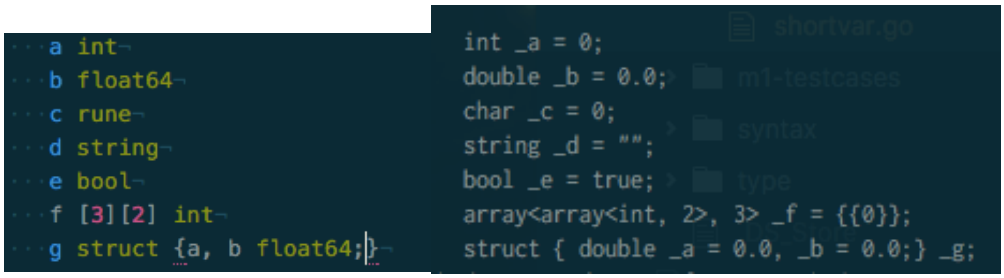


- Array out of bounds are handled instead of having undefined behavior. This is done by have a try catch block in the main function.



```
int main() {
        try {
                int _b = 2;
                int _a = 1;
                _b = 4;
                int _c = 3;
        } catch (const std::out_of_range& e) {
                std::cout << "\nIndex out of bounds: " << e.what() << '\n';
        }
        return 0;
}
```

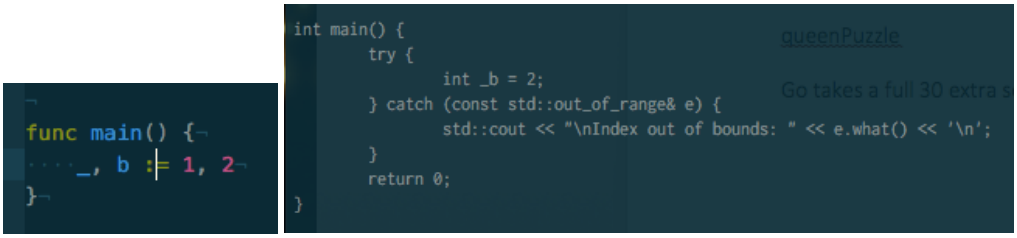
- Like Go, variables are given a default value.



```
a int
b float64
c rune
d string
e bool
f [3][2] int
g struct {a, b float64;}
```

```
int _a = 0;
double _b = 0.0;
char _c = 0;
string _d = "";
bool _e = true;
array<array<int, 2>, 3> _f = {{0}};
struct { double _a = 0.0, _b = 0.0;} _g;
```

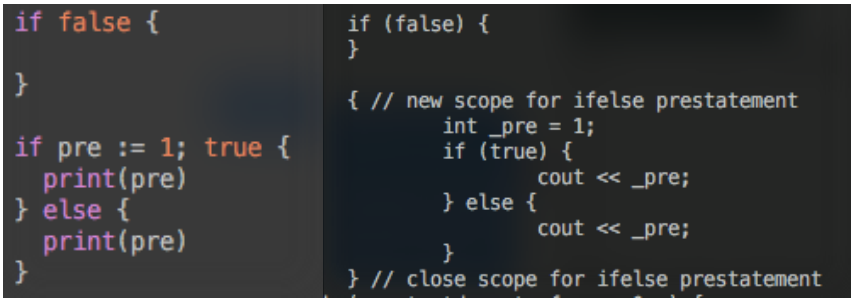
- Blanks are dropped from generation



```
func main() {
    _, b := 1, 2
}
```

```
int main() {
        try {
                int _b = 2;
        } catch (const std::out_of_range& e) {
                std::cout << "\nIndex out of bounds: " << e.what() << '\n';
        }
        return 0;
}
```

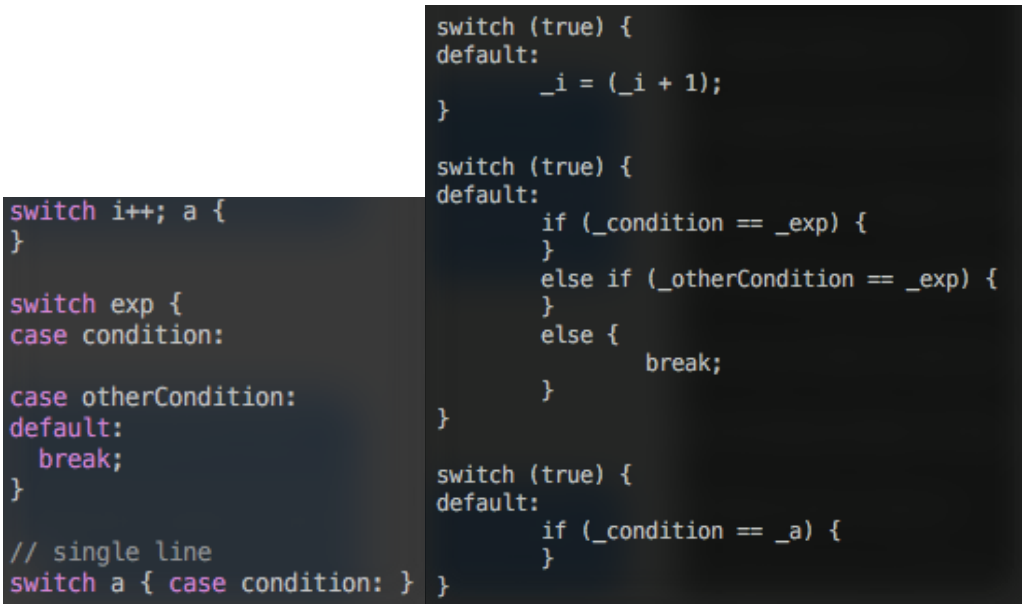
- If(-else) statements



```
if false {

}

if pre := 1; true {
  print(pre)
} else {
  print(pre)
}
```

```
if (false) {
}

{ // new scope for ifelse prestatement
        int _pre = 1;
        if (true) {
                cout << _pre;
        } else {
                cout << _pre;
        }
} // close scope for ifelse prestatement
```

- Switch statements. Go(Lite) cases allow for conditions (e.g. x>0). We decided to generate C++ if statements inside a switch with a single case – default – so that break statements would still work without any modification



```
switch i++; a {
}

switch exp {
case condition:

case otherCondition:
default:
  break;
}

// single line
switch a { case condition: }
```

```
switch (true) {
default:
        _i = (_i + 1);
}

switch (true) {
default:
        if (_condition == _exp) {
        }
        else if (_otherCondition == _exp) {
        }
        else {
                break;
        }
}

switch (true) {
default:
        if (_condition == _a) {
        }
}
```

Some of the constructs that will generate code but not run correctly:

- Array or slice with inline declaration of structs. This could possibly be solved by having a struct list with which we compare every struct declarations and replace them with an appropriate type alias.

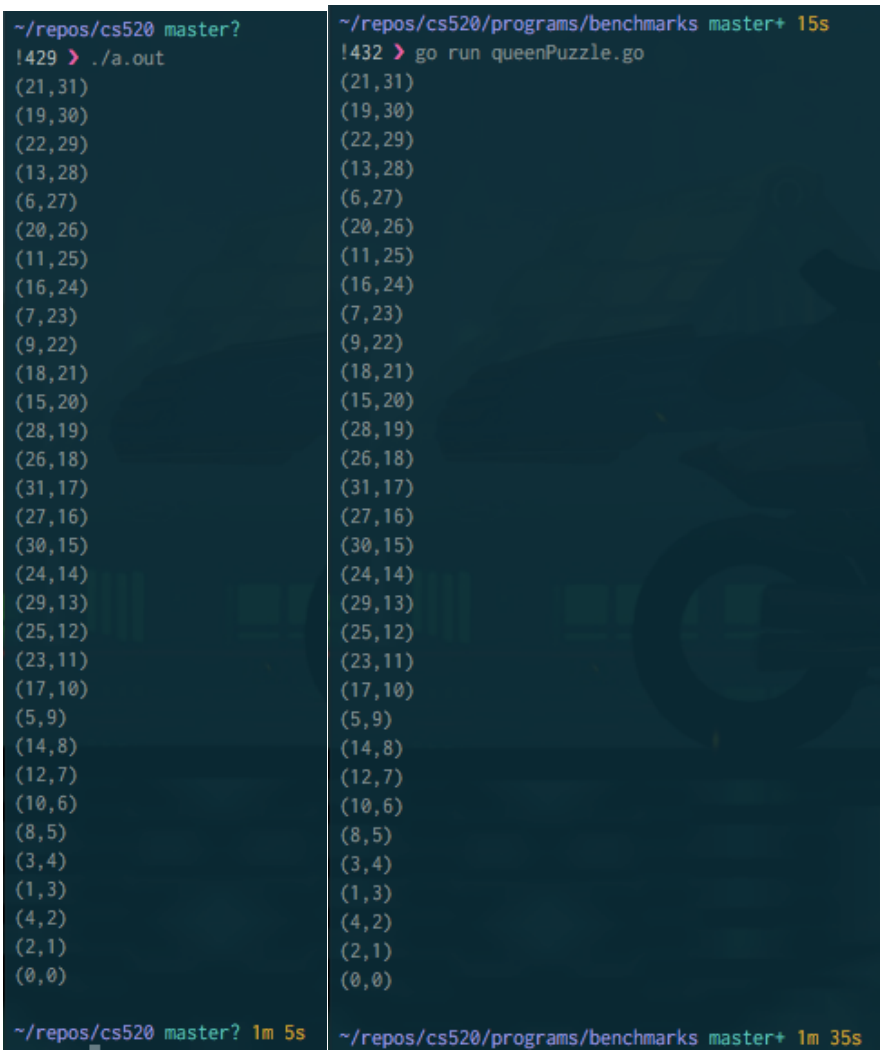
```
f [3][2] struct {a int}
```

- Division by zero is not handled properly as C++ has no predetermined behavior in this case. To mitigate this, we weed out division by constant zero. But this still allows for cases where an expression or function call can still evaluate to zero and cause errors.

Benchmarks

queenPuzzle:

Go takes a full 30 extra seconds to compute the same result.



primeNumbers

Here the difference is even greater at about 50 seconds.

Analysis on benchmarks:

The second benchmark is heavily centered on the use of slices. We constantly add more items to the list (in this case 5 million numbers added) and we believe that the main reason why the code written in C is a lot faster is because slices in Go are really slow (or at least the way we use them for GoLite).

The queen's problem only had 32 elements that were inserted at the start of the program and it never grew during the execution of the program. Although the difference isn't as big as for the prime numbers benchmark it still exceeded our expectations. However, queen's puzzle used *structs* and one thing we realized when the code was written is that if you extract it from the slice and change its values, the next time it will be extracted for the slice it will be the old copy and the modifications made previously are lost. The way we fixed the problem was to put the modified object back into the slice and overwrite the old value. This suggests that in Go *structs* are copied every time we extracted it from the slice. But in C we just get a pointer and no time is wasted.

Conclusion

To conclude this report, we have designed and implemented a compiler for a subset of the Go language, learning to rationally defend difficult design decisions, devise meaningful and reusable patterns and code better in general. In this report, we have presented the front end and the back end of our final compiler. Considering that working with C is known to be more tedious than working with other languages which abstract away low-level structures, it would be interesting to consider writing a compiler for GoLite using SableCC and to compare the advantages and disadvantages of both approaches.