GOLITE TO C++ compiler

Xiru Zhu 260498154
Tianzi Yang 260599365
Hongji Cheng 260610116

April 15, 2015

1. Introduction

We initially considered X86 assembly language as potential implementation languages as compiling Go to X86 is a very challenging task. However, we give it up due to some critical problems in code generation part and it might take a long time to find the solution. Instead we choose C++ because it is much more simple and less time consuming.

We decided to use Bison and Flex on scanner/parser because most of the team members are good at with writing C code. Also, C is several times faster than other languages like Java and python. This is also the reason we choose C++ as our output language. Moreover, C++ code made us easier if we want to compile it to lower level language.

2 Lexing

2.1 tokens

Most of our tokens generated by the Flex scanner followed the rule from (http://www.sable.mcgill.ca/~hendren/520/2016/assignments/syntax.pdf)

2.2 semicolon insertion

Go provides the basic rules for semicolon tokens insertion. To implement that, first we create a global variable called prev_token which store the last token passed. Then, whenever we get a "\n" token, we will return a ";" token if prev_token satisfies any rules in Go. Also, we will increase the line_num counter to store the number of new lines.

2.3 get syntax errors for switch

In GoLite, switch statement is created by four parts, ExprSwitchStmt = "switch" [SimpleStmt ";"] [Expression] "{" { ExprCaseClause } "}" (from https://golang.org/ref/spec#ExprSwitchStmt). Here both SimpleStmt and Expression are optional. However, if only SimpleStmt is given, then the expression part should be true by default.

Example:

```
switch x := f(); {  // missing switch expression means "true"
        case x < 0: return -x
        default: return x
}
```
(from https://golang.org/ref/spec#ExprSwitchStmt)

It is fine if token "{" is on the same line with "switch" [SimpleStmt ";"] [Expression], however, if we write the "{" on a new line, this could cause a serious syntax error.

Example:

```
switch food
{
		case "tomato":
			// Do something
		case "pepper":
			// Do something
		case "cucumber":
			// Do something
		default:
			// Do something
}
```
(from team 5 invalid case switch_rob_brace_no_init_stmt.go)

Our semicolon insertion will return a ";" token on the first line after food(id token). That ";" adding will make Bison recognizing it as a SimpleStmt. However food is an expression and there should be no ";" after it.

Also, if the if the first line is

```
	switch food := "dill pickle"; food
```
Due to the same reason, this is also wrong.

It is pretty hard to solve the first case(switch food), because at Lexing part, we can not distinguish a token is SimpleStmt or expression. Initially, we add some extra counter to solve the above two problems which is not that success. At the end, we find type checking can handle the first mistake and Bison will get the second mistake.

2.4 overflow error

If original code assign a huge number to int or to the size of array, we will not allow that. So, whenever we meet a int_literal or float_literal, we will use strtof() function to check if it is overflow or not. If it is, then we return an error and stop the entire compiler.

2.5 (.) case

if there exits a case which doesn't belong to any token we set, Flex will report an default error. To prevent that default error, we put (.) at the end of all the tokens and return our own error token.

2.6 for statement

```
for i:=0; i < 5; i++
{
        print("a")
}
```

This is not allowed in GoLite, but below two are correct

```
for i:=0; i!=0;
{
}
```

```
for i:=0; i < 5; i++{print("a")}
```

So, whenever we get a "for" keyword token, we will check the last token but one at that line. In other words, if the token before "\n" is neither "{" nor ";" then we report a mistake.

3. Parser

3.1 Grammar

The basic grammar followed (http://www.sable.mcgill.ca/~hendren/520/2016/assignments/syntax.pdf). And it has three main parts Declaration, Expression and Statements.

3.2 some changes

To prevent some shift reduce error and reduce reduce error. We changed some grammar and those changes will be fixed in the next phase.(weeding)

1) append_expr ::= 'append' '(' ID , expr ')' . Here, we use primary expression instead of ID.
2) Type casts. We use type_cast ::= basic_type_cast '(' expr ')', here it cannot accept base types. The aliases type will be recognized as function call which will be fixed in weeding.

4. Type Checking

4.1 Symbol Table

The symbol table is a tree based system with each node representing a single scope. The table itself uses a simple hash table of 2048 entries where each entry is a linked list to allow for collision. Performance is acceptable and rather well spread.

Certain nodes on the AST tree with get annotated with a scoped symbol table to help with code generation. I would like to note that it is each scope which gets its own table and that memory use can get rather high with large number of scopes.

4.2 Type

A type is composed of a single number, a pointer to its declared AST node and potentially recursively more types in case of Slice, Arrays, Struct or type alias. Each AST node that is an expr will have an annoted type pointer to aid code generation. Type comparison is done the following manner, for all basic type, directly. For type alias, by comparing directly an alias number, which requires all alias type to be declared at the same location or else will not be equal.

Example:
type a int;
type b int;
var val1 a 4
var val2 b = 3
a + b is invalid

For any arrays, array subtype is compared recursively and the length must be compared. Arrays of the same type and of unequal length must be rejected. Such rules do not apply to slice.

Alias type also contain a special flag which tells us whether it is a type or an id since we have difficulty distinguishing the two when comparing type. Values with an alias type flag off would mean it is a id. Values with an alias type on would mean it is a type and cannot be used as values.

For Structs, type comparison directly involves checking every single field id name and type and making sure the order is followed. It is a rather expensive operation.

4.3 Type Checking Traversal

The type checking traversal operations in a similar fashion to the code generation and weeding portion. To begin with, there exists a super scope so to speak which contains the following values. True, false and _, as ids predeclared which can be shadowed besides _ which will throw a weeding error. The next level is the global scope and where we begin type checking, starting from declarations, all the way down to basic literals.

In terms of special rules, any _ declarations were ignored completely but any _ for parameters required a value but could not be used. That's some of the weirdest rules I've seen. Some weird things include anonymous structs which was a bigger problem in the code generation phase. Other weird rules seen include the short declaration needing at least one new variable. As part of type checking, we have specifically IMPLEMENTED printing for arrays and slice. So for our compiler, it is not a type checking error but allowed. Only printing struct is not allowed.

Otherwise, it mostly unevenful giant pile of code to type check everything. Function declarations are also checked for type alias typecast. All if loop cause a new scope so having a series of if elseif.... will lead to a very large chain of scope table and memory use.

5. Code Generation

5.1 Type implementation

We have

string	->	std::string
rune	->	char
int	->	int
float64	->	double
bool	->	bool

The array and slice types are implemented by the template class GoLiteArray and GoLiteSlice
class GoLiteArray<arrayType, arraySize>

- operator<< invoked when printing an array
- operator[] invoked when indexing a element within the array
- operator= invoked when copying an array
- slice(begin,end) support array slicing
- slice(begin,end,capacity) support array slicing
- length return array length (const value)

class GoLiteSlice<sliceType>

- operator<< invoked when printing a slice
- operator[] invoked when indexing a element within the array
- operator= invoked when copying a slice
- slice(begin,end) support slice slicing
- slice(begin,end,capacity) support slice slicing
- length return array length (dynamic value)

A copy of a GoLiteArray is a deep copy, which means it will create a copy of all the element within the array recrsively, and build a new array out of them. A copy of a GoLiteSlice will result in two slice sharing the same memory position.

5.2 Garbage Collection

A reference count garbage collection system is build on the the GoLiteArray, and GoLiteSlice, by using the C++ standard library (std::shared_ptr<T> and std::unique_ptr<T>). When a array is created a new shared pointer is build on a newly allocated memory, and if the array is deallocated, and no one else is pointing on the memory loaction, the memory will be released by the destructor function of std::shared_ptr<T>. The element in the array will be construct on the heap, and bind with a std::unique_ptr<T> which is stored in the array. Thus, when that array is deallocated, the unique pointers will deallocate themselves and recursively free the whole list.
As shown in class, one major problem of reference count is cyclic reference. However, we claim such situation will not appear in GoLite, as we do not have pointer type, and all array are managed in a tree-like data structure.

5.3 Anonymous structs

As, Anonymous structs are forbidden for template in C++, we have to define our structs before using them. We classify anonymous structs into global anonymous structs and local structs. Global anonymous structs are those appears in global type definition and function parameter definition. Its easy to handle them. First move all the type definition to the top of the code, then generate a list of anonymous by using post order traversal on the struct definition tree.

The local anonymous structs are defined in the place where they are needed.

```
func foo() {
        var example struct {
                entry   int
        }
```

```
}
```

will be transformed into :

```
func foo() {
        type GoLite_anonymous_1 struct {
                entry   int
        }
        var example GoLite_anonymous_1
}
```

However, the transformation will be trickier when the scopes are involved, for example:

```
func foo() {
        {
                var example struct {
                        entry   int
                }
        }
        {
                var example struct {
                        entry   int
                }
        }
}
```

In this situation we have to transform the code, by defining the struct in both scopes. The functions support these features are in structManager.h

5.4 Code Generating Strategy

Most of the code generating is trivial, for example, a,b,c,_ = e1, e2, e3, e4 is transformed into

```
typeof(e1) GoLite_local_0 = e1
typeof(e2) GoLite_local_1 = e2
typeof(e3) GoLite_local_2 = e3
typeof(e4) GoLite_local_3 = e4;
a = GoLite_local_0;
b = GoLite_local_1;
c = GoLite_local_2;
```

The for loop is reduced into a initialize statement, a while condition check loop, and a increment statement, and the switch is reduced into a group of else if statements. The continue and break is implemented using a label jump.