GoLite

- Compiler Design -

Project Report

Group 05b

260683698 Duan Li

McGill University
Computer Science

Contents

Chapter 1

Introduction

This report first introduces Go and GoLite, gives an overview of the GoLite compiler, then briefly discusses the first 5 components (Scanner, Parser, Weeder, Symbol Table and Type Checker) of the compiler, finally talks about the final component code generator in detail.

1.1 GoLite and Go

Go (often referred to as Golang) is a programming language created at Google in 2009 by Robert Griesemer, Rob Pike, and Ken Thompson. Go is a statically typed language in the tradition of C, with memory safety, garbage collection, structural typing, and CSP-style concurrent programming features added.

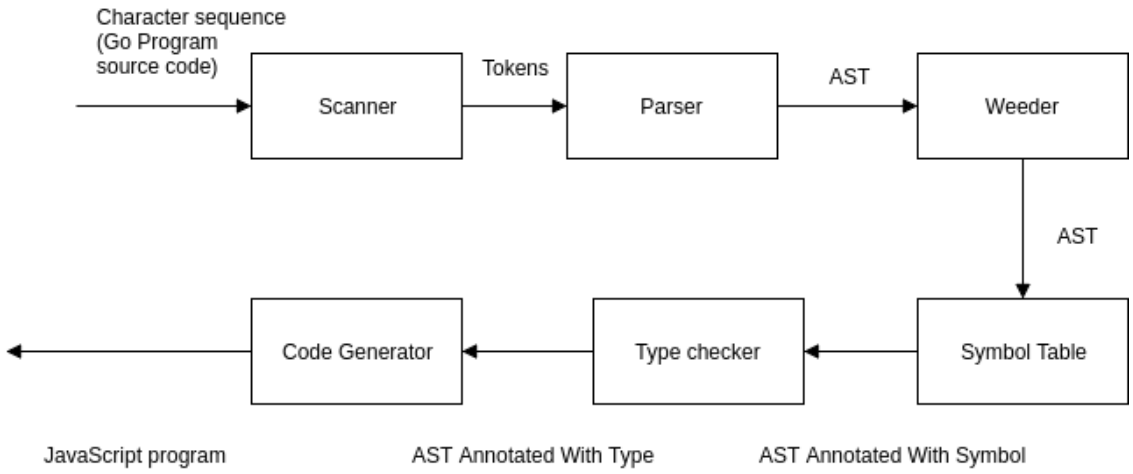
GoLite is a subset of Go, that contains interesting features including variable, function, type declaration, control structures like if-else statement, switch statement and a bunch of other basic Go features. The detail language specification of GoLite can be found in Appendix.

1.2 GoLite Project

The purpose of this project is to build a compiler for GoLite, to gain better understanding of modern compiler development via hands-on experience.

For most compilers, the compilation process is a sequence of various phases. Each phase takes input from its previous stage and feeds its output to the next phase of the compiler.

The GoLite compiler has 6 phases: Scanning, Parsing, Weeding, Symbol Table Generation, Type Checking and Code Generation.



1. Scanner scans the source code as a stream of characters and converts it into meaningful tokens.
2. Parser takes the string of tokens and builds a parse tree according to the grammar. the GoLite compiler uses AST (abstract syntax tree) for parse tree, and it will be used as intermediate representation of the program in the future phases.
3. Weeder rejects some bad parse tree.
4. Build a symbol table to collect and analyze symbol declarations, and relate symbol uses with their respective declarations.
5. After symbol table phase, each symbol in AST is annotated with its declaration, therefore its type is inferable. Type checker determines the types of all expressions which are made up by symbols, checks if the values and variables are used correctly.
6. Code Generator generates code in JavaScript.

Many modern compilers have an additional phase to optimize the code before final code emission, for example, a peephole optimizer. GoLite compiler does not contain this feature.

Chapter 2

Language and Tool Choices

2.1 implementation language/tool

The GoLite compiler in implemented in C.

C provides powerful libraries flex and Bison that greatly simplify compiler development.

Flex is a tool for generating scanners. It allow us to define regular expression for each type of tokens, associated actions to be performed when matches on those expressions are found, and generates a scanner correspondingly.

Flex generated scanner has many advantages over a handwritten one. First of all, it's simpler to code with flex, as you can focus on specifying tokens with regular expressions and actions. There is no need to deal with file IO. The flex version of a scanner is invariably much shorter than equivalent handwritten one, which makes it a lot easier to debug. Moreover, for complex scanners with many patterns, like the one for GoLite, a flex scanner may be faster, since handwritten code will usually do many comparisons per character, while flex always does one with the help of pattern matching technique.

Bison, being a parser generator, allows us to specify the GoLite's grammar directly in a form that very closely identifies with the Backus-Naur form that is used to describe language grammars generally. This is opposed to a hand-crafted parser which often has much of the parser's logic intertwined with the grammar and action rules. Again, it is easier to debug errors in grammar logic as a result.

We also considered Java as an option before, since Java is a higher level language, coding in Java would be much simpler than C as it provides many convenient features

like built-in helper functions, and automatic garbage collection. With Java, we don't need to deal with pointer and memory allocation.

However, we cannot find as much resource about Java scanner, parser generating tool online as that of flex and Bison. Consider the fact that flex/Bison has a more helpful community, we choose C as the language for the project.

For future compiler project, we are interested in trying out a high level functional language. As with C there are too many code, functional language will greatly reduces the amount of code.

2.2 target language

The target language of the GoLite compiler is JavaScript. (Actually Node.js as JavaScript only support "println" but not "print".)

As a high-level, dynamic typed and prototype-based programming language, JS is very easy to code. Following are some advantages of JS.

Simple User Defined Type Implementation Unlike Go and many other languages, JS only has a very limited number of types - only 6 types. Specifically:

- Undefined
- Null
- Boolean
- String
- Number
- Object

It also does not support user defined type. All non-null value that is not of basic type (Boolean/String/Number) is of type Object. This makes type declaration super easy as we simply don't need to declare any type in JS, we could basically ignore all type declaration statements.

When we need to initialize some variable of a user defined type, we create a object according to the type definition. We will talk about this in detail in code generator section.

Simple Slice Implementation Slice type in GoLite is like array, but is dynamic sized. Its underlying implementation utilizes pointer. Fortunately, JS supports dynamic sized array. So only with minimum modification we can support Slice with JS built-in array. We will talk about this in detail in code generator section.

Good Performance Server side JavaScript Node.js is really fast.

Following is the benchmark execution time comparison between Node.js, Python, and C++.

Benchmark	Node.js	Python	C++
nqueens	19.8s	34s	1.7s
mergesort	19.3s	1m8s	6.9s
insertionsort	2.3s	54.6s	7.4s
primes	15.8s	1m5.9s	5.5s
selectionsort	28.0s	1m8.5s	11.1s
tsp	17.5s	1m35.5s	10.1s
hanoi	7.8s	57.5s	3.4s

Both being dynamic language, Node.js is 2 5 times faster than Python.

Chapter 3

Scanner, Parser

3.1 Scanner Overview

Scanning is the first phase of a compiler. It's also called lexical analysis.

The GoLite compiler takes an arbitrary GoLite source file as input, the scanner reads the input stream, identifies meaningful sequences of characters and output tokens.

For the GoLite compiler, we take advantage of flex to generate scanner. In flex file, we just write a bunch of regular expressions to define tokens.

3.2 Scanner Major Design Decisions

Base Type Name Unreserved Unlike what we originally thought, the names of base types in Go is not reserved keyword. (You can define a variable with name "int".) Also are words "true" and "false".

Therefore, we decide to scan all of these as tIDENTIFIER.

Handling reserved keywords Reserved, unsupported keywords such as fallthrough and defer were originally caught and error-handled at the scanner level. This was because of the ease with which Flex could match these keywords, immediately throw an error and exit. But after checking the specification/reference compiler, we found out that they need to be handled at the parser level.

To model our compiler's behaviour more closely on the reference compiler, the scanner now return tokens to the parser for reserved keywords, and the parser is responsible for throwing errors. Because the parser has no matching production rules, it always throws an error when these unsupported keywords were encountered.

Block comments Writing regex to recognize multi-line comments was difficult. The original strategy was simply to look for an opening /* and a closing */, there are many potential edge cases, including escape sequences, newlines and (escaped) slashes. Writing regex to handle all these edge cases brings the extra danger of 'eating' too much input.

To avoid dealing with edge-cases and other annoyances of using regular expressions, we followed a suggestion of the Flex manual to keep track of the comment state (i.e. whether or not the scanner is currently inside a block comment or not) via start conditions. If the scanner sees a '/*' while not in the INITIAL state, we set the global state to *C_COMMENT*. If the program enters the *C_COMMENT* state and sees a '*/', the program state goes back to INITIAL.

Optional Semicolons Optional semicolons were tricky because they need to be added on a newline but only if the last token inserted was a literal value, a type, a break/continue/fallthrough/return statement, "++"/"–" operation, or a '}', ')' or ']'

The scanner keeps track of the most recent token. To do this, we followed the tinylang example and defined a macro called 'RETURN' which records the last token in a global variable and then returns the token. This way the scanner knows what the last token was and whether to insert a semicolon.

3.3 Parser Overview

Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols/tokens, conforming to the rules of a formal grammar. It is the second phase of the compiler.

3.4 Parser Major Design Decisions

AST node data structures

The AST tree classes were not generalized. As a result we had a lot of classes that could have otherwise been merged together. This required us to write a lot more code. There is definitely a trade-off between having "dumb" structs which are effectively 'one-time use' and higher level structs which take multiple parameters and can be used in different parts of the grammar.

While the former option results in a much smaller, but more complex code base, the latter option more closely aligns with JOOS' by involving more specific structures. We chose the latter because for readability and ease of understanding, which is important when multiple people need to read and understand the code base.

Chapter 4

Weeder

4.1 Overview

Weeding is the third phase of the compiler. The weeder will take the AST output by the parser, and remove some bad parse tree.

Some grammar are not context-free. It's hard, if not impossible for parser to reject all syntactic errors. For the benefit of cleaner and simper grammar at parsing phase, the weeder is added to the compiler.

4.2 Major Design Decisions

Weeder Responsibilities

- Make sure blank identifier is used appropriately.
- Make sure every branch of the source program ends up in a terminating statement.

Chapter 5

Symbol Table, Type Checker

5.1 Overview

Symbol Table is used to perform two important functions:

1. collects and analyzes symbol declarations
2. relate symbol uses with their respective declarations

At this phase, undeclaration and redeclaration of variable are reported.

After symbol table associates each symbol with its declaration, type checker performs the semantic analysis. It infers and assigns types to expressions, outputs an annotated AST.

5.2 Major Design Decisions

Scoping Rules For the most part, our scope rules are simple. In minilang we only needed a single global scope, but GoLite has a few levels. Functions, and switch, if/else, and for statements introduce a new scope. Within the symbol table, this consists of creating a new symbol table, linking it to the current scope (i.e. setting the current scope as parent), and using this new symbol table for the rest of the block.

All scope-related rules are upheld by the symbol phase. Mainly, we need to determine the question each time an identifier (variable, type, or function) is found: has this been previously defined, and if so, at what scope? Once that information is retrieved from the symbol table, we can add additional type information to the AST which will be needed for type checking. Any variables/types/functions being defined for the first time or re-defined (at a new scope) must have their information logged in the symbol table.

Shadowing of variables and (especially) types is one of the more difficult features we had to implement. Keeping track of whether a given identifier is referring to the current scope or a higher one is challenging, and we weren't able to fully implement it in time.

Others When implementing the symbol table, we realized that there were some unresolved issues from milestone 1 that weren't caught from previous testing. Fortunately, once we figured out the source of our segfaults (not setting type information in our AST for base-type expressions), we were able to move along implementing the symbol phase.

Incremental testing was another issue we ran into when building the symbol table. Printing is interleaved with the actual symbol generation process. This allowed us to slowly test our symbol table in relation to the reference compiler, confirming that both print out the same information. However, this reliance lead to problems when integrating with the typechecker. Although our symbol table was clearly correct (at least for our test cases), what isn't visible is the type information added to the AST in the process. Because forgetting to generate types correctly generally caused segfault errors, it took a lot of time to iron out our issues with GDB.

Implementation of the symbol table and type checker were largely done separately. This was necessary due to time/resource constraints, but meant that the integration step between the two was quite nasty. The type checker can't be easily tested until it has a reliable symbol table and generated type information within the AST. So not only did we have to heavily refactor the type checker during integration, but we also had to work through many nasty bugs internal to the type checker.

One additional trip-up in the implementation of the symbol table was realizing the need to return a SymbolTable struct from a for clause. This is because not only does a for statement create a new scope, but it adds to that new scope any variable definitions made in the init statement.

Our GoLite implementation is modelled off JOOS. Like JOOS, we do not need to pass symbol table around as each variable already has a pointer to its declaration. Typechecking is based on this. However, unlike JOOS, Golite requires declaration of (e.g. variables, types and functions) before usage, so we only need one pass each for both the symbol phase and typechecking phase. There is no need to split each phase into two passes like JOOS which has an interface pass which discovers all the types first, and an implementation pass.

Further, the process declaring a type in GoLite is different from JOOS. JOOS (like Java) uses classes, and it is impossible to declare a class or a class alias inside another class. In GoLite, we need to add additional support for type aliases and resolving types. These both require that we link every usage of a type to its specification. So in the symbol phase, for every usage of an identifier, we find its symbol in symbol table, get the its specification from symbol, and add the specification reference to the identifier node itself.

In symbol phase, if we find a declaration statement x (i.e. a declaration or a short

declaration), we create a new symbol s, set ‘s->specification = x‘ and put it in the symbol table.

Each type check function calls the type check function of its components. E.g a program is comprised of declarations and statements, so the first thing that happens is: typeCheckProgram-> typeCheckDecl and typeCheckStmt etc. When we get the identifier exp level, we implement its type by referring to its specification and then build the types from bottom up.

Chapter 6

Code Generator

6.1 Overview

Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into another form (e.g., machine code, assembly code, or some other language).

Our code generator takes the type annotated AST, traverse down the tree and generates server side JavaScript, Node.js code line by line. In this chapter we will first go over the design decisions, then the problems encountered during the implementation.

At this phase, we can assume that the source program is a valid GoLite program. Lexical errors are rejected at scanning phase, syntactic errors are rejected at parsing and weeding phase, type related semantic errors are rejected at type checking phase. There could still be errors, e.g. invalid array/slice index, and we leave this to be runtime error.

Due to time constraint, evaluation order constraint is not implemented. However, the GoLite compiler passed all the standard test cases and achieve great performance in benchmarks.

6.2 Design Decisions and Problems In Testing

6.2.1 Type System

JS's type system is really simple: only 6 types, specifically:

- Undefined
- Null

- Boolean
- String
- Number
- Object

JS does not support user to define its own type.

GoLite's type system is more complex. It has base types as well as user defined type. For user defined type, it supports recursive type for slice.

We will talk about how we map GoLite type to JS type in three parts: base type mapping sequence type (array, slice) and user defined type mapping.

Base Type

We represent value of base types GoLite using JS base types.

GoLite	JavaScript
int	Number
rune	Number
float64	Number
string	String
bool	Boolean

In JavaScript there is only one number type: the double-precision 64-bit binary format IEEE 754 value (numbers between -(253 -1) and 253 -1). There is no specific type for integers. Therefore we have to map all numeric GoLite types (int, rune, float64) to JS Number type.

This makes things a little bit trickier when we do math operation division between integer types. In GoLite, for binary operation between two int types, the result is also of type int. For example, 6/5 should evaluate to 1. However, as JS use float64 for all numbers, 6/5 will evaluate to 1.2.

Therefore, we provides an utility function "intDiv", which takes in the lhs and rhs as input, and performs the division operation. It will do the division, and round down the result if it's positive, or round up the result if it's negative. (In GoLite, -6/5 = -1, if we round down -1.2, it will be -2 but not -1.)

The utility function "intDiv" as well as all the other utility functions, will be print to the JS code at the very beginning, so the the following program could call the utility functions when it needs.

The following is an example.

```
// GoLite source code snippet
a = 7 / 6;
a = (b+c) / k;
```

```
// generated JS program
// ... utility function intDiv
function __intDiv(a, b){
    let result = a/b;
    return result > 0? Math.floor(result) : Math.ceil(result);
}
// ...
a = __intDiv(7, 6)
a = __intDiv((b+c), k)
```

Sequence Type

For sequence type Array and Slice, we use JS array, but with some modification.

Array In GoLite, an array is a numbered sequence of elements of a specific length. In JS, array is dynamic sized, and can contain element of different types.

We use JS built-in array to represent GoLite array. Even though JS array is more generic then GoLite array as it does not check if the elements are of the same type, we have already checked element in type checking phase, so it's okay.

There are just two problems we have to deal with:

1. JS array is dynamic typed, it does not throw invalid index error. (It will just return "Undefined".) Therefore we provide two utilities functions "arrayIndex" (get element at index x) and "setArrayValue" (set element at index x). We call the corresponding utility function when we get/set array element value. The utility function will be responsible for checking if the index if valid.

   ```
   // utility function setArrayValue
   function __setArrayValue(a, index, value, size){
   ```

```
    if (index >= size){
       console.log("Error: out-of-bounds")
       process.exit(1)
   }
   a[index] = value
}

// utility function arrayIndex
function __arrayIndex(a, size, i){
   if (i >= size) {
       console.log("Error: out-of-bounds")
       process.exit(1)
   }
   return a[i]
}
```

Following is an example.

```
// GoLite source code snippet
a[1] = 1;
b = a[2];
```

```
// generated JS program
// input size (array size) is required. JS has no idea about the size of
    the array, but our codeGenerator knows the type of EXP, including the
    size of the array. It will provide the size of the array as parameter
    when it generates the statement.
// Assume size of a is 3.
__setArrayValue(a, 1, 1, 3)
b = __arrayIndex(a, 3, 2)
```

2. JS initilizes array with value of undefined. Therefore we provide utility function "newArray" to initialize array with appropriate initial value.

```
// utility function newArray
function __newArray(len, instance){
   return Array.from(new Array(len), x => instance)
}
```

Example:

```
// GoLite source code snippet
```

```
var a [5]int
var b [4]string
```

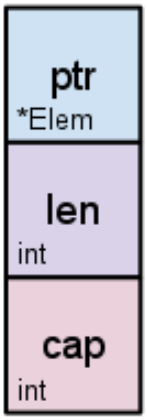
```
// generated JS program
// We have to provide instance (initial value of the element type) for the
    function.

// a's element type is int, the initial value of int is 0 (of JS type
    Number)
let a = __newArray(5, 0)

// b's element type is string, the initial value of string is empty string
    "" (of JS type String)
let b = __newArray(4, "")
```

For how to generate the initial value of a certain type, we will discuss in detail later in section "Variable Initialization".

Slice The GoLite slice type is an abstraction built on top of GoLite's array type, it does not have specified length. Internally it's a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).



As JS array is dynamic in nature, we use it to build the slice type.

```
// Slice JS implementation
  {
    isSlice: true,
    val:[],
```

```
      size:0
    }
```

Similarly, we have utility functions newSlice, sliceIndex, setSliceValue for slice.

```
// utility functions for slice
function __newSlice(){
   return {
          isSlice: true,
          val:[],
           size:0
       }
}

function __sliceIndex(s, i){
   if (i >= s.size){
       console.log("Error: out-of-bounds")
       process.exit(1)
   }
   return s.val[i]
}

function __setSliceValue(a, index, value){
   if (index >= a.size){
       console.log("Error: out-of-bound")
       process.exit(1)
   }
   a.val[index] = value
}
```

One tricky thing about slice is that modification of a slice in sub function call may not be visible.

```
// Go program
package main

import (
   "fmt"
)
func f(a []int){
 a = append(a, 1)
}
```

```
func f2(a []int){
 a[0] = 2
}
func main() {
   var a []int
   f(a) // append in sub function call is not visible
   fmt.Println(a) // []

   a = append(a, 1)
   f2(a)   // modify element in sub function call is visible
   fmt.Println(a) // [2]

}
```

To mimic this, we utilize the size attribute of slice. When we pass a slice object to a function as parameter, we clone the slice object. The cloned slice object has attribute val points to the same array.

When we append an element to the slice, we add the element to the end of val array (sliceObj.val[sliceObj.size]), then increment the size by one. If this happens in a sub function, even though the element is appended to the array, it's not visible to the caller since the original slice object's size remains the same.

```
// JS utility function append
function __append(array, element){
    let _a = __clone(array)
    _a.val[_a.size] = element
     _a.size++
    return _a;
}
```

But when we modify an element value in a sub function call, it's visible to the caller since the cloned slice object's val and original slice object's val refer to the same array.

User Defined Type

Basically the code generator will ignore all type declaration statements, and use objects to represent all user defined types.

```
// GoLite source code snippet
type my struct {
```

```
   a int
   b string
}

var i my
```

```
// generated JS program snippet

// do not generate anything for type declaration

let i = {
   a: 0,
   b: ""
}
```

How to initialize a variable of user defined type is discussed in section "Variable Initialization".

6.2.2 Variable Initialization

Variable needs to be initialized with initial value when it's declared with a type but not assigned any value.

```
// a needs to be initialized with initial value of type t1
var a t1
```

codeTypeInstance(TYPE* t) is responsible for generating the initial instance of type t.

Base Type It's easy to initialize instance for base types.

Type	Initial Value
int	0
rune	0
float64	0
string	""
bool	false

Sequence Type For sequence types, as said before, utility functions newArray and newSlice are used. For array type, We not only create an JS array, but also create instances of the element type and fill the array with the instances.

User Defined Type

1. Create a JS empty object o.
2. For each field f of the struct type do:
   (a) Generate instance fi for f.type
   (b) Add key-value pair (f.fieldName, fi) to o

6.2.3 Assignment

Each assign statement node have a list of Assign nodes, the code generate will generate a assign statement in JS for each Assign node.

```
// GoLite source code snippet
x, y, z = 1, 2, 3
```

The most intuitive way is :

```
// generated JS code snippet (first design)
x = 1
y = 2
z = 3
```

Then we find a problem in the following case:

```
// GoLite source code snippet
x = 1
x, y = 2, x
// x: 2, y: 1
```

We realize we cannot simply update the variable value during the middle of assign STMT, as some later ASSIGN may use the variable value. Therefore we save the evaluated result to temp variable first.

```
// generated JS code snippet
temp0 = 1
temp1 = 2
temp2 = 3
x = temp0
y = temp1
z = temp2
```

Blank Identifier When the lhs is a blank identifier, the code generator will only generate the rhs part, since in GoLite blank identifier does no trigger any binding, but the rhs should be executed.

```
// GoLite source code snippet
_ = func1()
_ = 12
```

The most intuitive way is :

```
// generated JS code snippet
func1()
12
```

In JS, any expression can be a statement.

Other STMTs For other statements that involve variable assignment: inc, dec, OpAssign, and shortVarDecl (for defined variables it's assignment instead of declaration), the code generator generates code in similar fashion.

6.2.4 Passing Variables Between Functions

In GoLite all parameters are passed by value. JavaScript uses pass by value strategy for primitives but uses a call by sharing, which is similar to pass by reference for objects.

To mimic this, code generator supplies an utility function "clone", and it deep clones every parameter before passing it to a function.

For reference type slice, "clone" will make sure the underlying val array (which is used to hold the elements) refers to the same one.

Performance Using a lot of "clone"s can seriously influence performance. The performance at first is really bad, but after optimizing "clone", the generated code is hundreds time faster, and 5 times faster than Python standard speed.

Here is a few optimizations we did:

1. The code generator first checks the type of the parameter expression, and does not clone it if it's of base type since JS pass primitives by value too.
2. Does not use CPU-intense operation like JSON.parse(JSON.stringify(obj)) any more to do cloning.

3. Optimized slice structure, so there is no need to clone val array in slice.

6.2.5 Variable Shadowing

There is an interesting bug we encounter during the implementation.

```
// GoLite source code snippet

var x int = 3

func main() {
   println(x)
   var x int = 5
}
```

```
// Generated JS code snippet

let x = 3

function main() {
   console.log(x) // Error: x is undefined
   let x = 5
}
```

For now I am not sure about how this happens, but my guess is that it has something to do with hoisting. JS also throws an error for statement "let x = x". It seems that if a variable is shadowed in a scope, the old declaration is not visible in the current scope.

Our solution to this problem is to add a number (shadowedNum) to each symbol to indicate how many time it has been shadowed (a symbol that has not been shadowed has shadowedNum 0, once is 1, twice is 2 etc.). We use the combination of variable name and shadowedNum to represent the variable in JS, therefore differentiate the shadowed variables and make the previous value visible.

```
// GoLite source code snippet

var x int = 3

func main() {
   println(x)
   var x int = 5
}
```

```
// Generated JS code snippet

let x0 = 3

function main() {
   console.log(x0)
   let x1 = 5
}
```

Chapter 7

Conclusion

Overall, it's a great experience. I learned a lot from the experience. Now I have an overall idea of how compiler works =)

For decisions I would change if I have the opportunity...

1. Use a high level functional language to implement the compiler, it will significantly reduce the amount of code.
2. Do not construct AST tree node in a way too syntax specific, a lot of nodes could share the same structure. For example, instead of having "Inc", "Dec", "OpAssign" and "Assignment" as four different tree node types, they can all share the same type "Assignment". This will reduce work load in the future phases as well.
3. Should have started earlier.
4. All the team member should peer code for a while, or at least set up the interface and data structures together. Every one should at least participate in some coding, some testing, and some part of report, so every one would have a more comprehensive understanding of the project.
5. Should have made more effort in communicating with teammates.

Finally, really sorry for all the inconvenience I brought by leaving the group and submitting the last two milestones late. Thanks to Alex for all the help during the project =)

Chapter 8

Contribution

8.1 Authors

- Duan Li(260683698)
- Jacob Macneal (260566105)
- Kevin Gatdula(260823424)

The first two milestones of this project are done in cooperation with Jacob Macneal and Kevin Gatdula, the last two milestones (code generation and final report) are done individually by Duan Li.

Duan Li

- Implement parser individually
- Implement half of the AST
- Implement weeder individually
- Contribute half of the test cases in milestone 1
- Fix bugs in compiler after milestone 1
- Implement type checker individually
- Contribute half of the test cases in milestone 2
- Contribute to milestone 2 report
- Fix all the bugs in compiler after milestone 2
- Implement code generator individually
- Finish final report individually

Jacob Macneal

- Implement scanner individually
- Implement half of the AST
- Implement pretty printer individually
- Fix bugs in compiler after milestone 1
- Implement symbol table individually
- Integrate symbol table and type checker
- Contribute to milestone 2 report
- Contribute to milestone 3 report

Kevin Gatdula

- Contribute half of the test cases in milestone 1
- Finish milestone 1 report individually
- Contribute half of the test cases in milestone 2
- Write most of milestone 2 report
- Contribute to milestone 3 report

Appendix A

Appendix

- source code: `https://github.com/comp520/2018_group05b`