How to run

replace * with the test file you want to run

run: ezsharp.exe test/Test*.cp

If it dosent work run: go build main.go then run: main.exe test/Test*.cp

Lexical Analyzer (Compiler Phase 1) - Marc Mikhail

This is phase of 1 of 4 phases of building a compiler for the language ez sharp designed by Prof. Eugene Zima. Phase 1 of the compiler is the lexical analysis phase which is basically taking source code and parsing each bit and categorizing whether its a identifier, keyword or else.

I decided to build the compiler in Go (golang) because I think the simplicity of the language will help in delivering a concise project that will aid in my understanding of compilers.

File Structure

```
ezsharp/
    Table/
        table.txt
    keywords/
        keywords.txt
    lexer/
        lexer.go
    output/
        errors.txt
        token.txt
    test/
        Test1.cp
        Test2.cp
        ...
    token/
        token.go
    go.mod
    main.exe
    main.go
    ezsharp.exe
```

Documentation

In main.go I read the source file using io input, then I process it using a double buffer and then I initiate a new lexer and tokenize my input and print the tokens

and errors to their respective files.

./main.go

```
l := lexer.New(string(data))
for tok, line := l.NextToken(); true; tok, line = l.NextToken() {
    if tok.Type == token.ILLEGAL {
        e1 := fmt.Sprintf("{Type:%s Literal:%s line:%d}\n", tok.Type, tok.Literal, line + 1)
        errorf.WriteString(e1)
    } else {
        t1 := fmt.Sprintf("%+v\n", tok)
        tokenf.WriteString(t1)
    }
    if tok.Type == token.EOF {
        break
    }
}
```

In lexer.go I made a Lexer struct with the required vars to read through my input, a function to initialize my Lexer object and a NextToken() function to return tokens to my main program. The function is a switch statment built according to my transition table and since I made my transition table 128 characters long according to the ascii chart I just have to read the int value of the char I am reading and look it up in the table. Below is a snippet of the three things mentioned.

lexer.go

```
type Lexer struct {
    input        string
    position     int  // current position in input (points to current char)
    readPosition int  // current reading position in input (after current char)
    ch           byte // current char under examination
    line         int
}

func New(input string) *Lexer {
    l := &Lexer{input: input}
    l.readChar()
    return l
}

func (l *Lexer) NextToken() (token.Token, int) {
    var tok token.Token
    for ((state >= 0) && (state <= 22)) {
        switch (state) {
        case 0:
            state = TT[state][int(l.ch)]
```

```
            if l.ch == '%'{
                tok = newToken(token.PERCENT)
                goto next
            } else if l.ch == '*' {
                tok = newToken(token.ASTERISK)
                goto next
        case 1:
            l.readChar()
            l.skipWhitespace()
            state = TT[state][int(l.ch)]
        }
        case 2:
            tok = newToken(token.LE)
            goto next
    }
    next:
}
```

Finally I have a token.go file to sort out everything related to the actual tokens themself. Consts representing the tokens. A token struct with two strings to represent the type and literal and a lookupIdent function to determine whether a string is an identifier or a keyword. For LoopupIdent() it takes in the file keywords.txt which is a file consisting of all the keywords in the language ezsharp that have been sorted alphabetically to improve efficency,

token.go

```
const (
    ILLEGAL = "ILLEGAL"
    EOF     = "."

    IDENT = "IDENT"
    INT   = "INT"
    DOUBLE= "DOUBLE"

    PERCENT  = "%"
    ASSIGN   = "="


    LE = "<="
    LT = "<"
)

type TokenType string

type Token struct {
    Type    TokenType
```

```
    Literal string
}

func LookupIdent(ident string) TokenType {
    file, err := os.Open("./keywords/keywords.txt")
    _ = err
    defer file.Close()

    var keywords []string
    scanner := bufio.NewScanner(file)

    for scanner.Scan() {
        keyword := scanner.Text()
        keywords = append(keywords, keyword)
    }

    for _, keyword := range keywords {
        if ident == keyword {
            return KEYWORD
        }
    }
    return IDENT
}
```

Syntax Analysys (Compiler Phase 2) - Marc Mikhail

This is phase of 2 of 4 phases of building a compiler for the language ez sharp designed by Prof. Eugene Zima. Phase 2 of the compiler is the syntax analysis phase which takes the lexed output form phase 1 parsing each token and categorizing whether its a identifier, keyword or else. But most importantly it checks the syntax of the code to see if it is correct, and follows the grammer rules of the language.

File Structure

```
ezsharp/
    Table/
        table.txt
    keywords/
        keywords.txt
    lexer/
        lexer.go
    output/
        errors.txt
```

```
        lexerOutput.txt
        parserOutput.txt
    parser/
        parser.go
    test/
        Test1.cp
        Test2.cp
        ...
    token/
        token.go
    README.ME
    README.pdf
    go.mod
    main.exe
    main.go
    ezsharp.exe
```

Documentation

In main.go I pass the output of the lexer to the parser and then I print the output of the parser to a file called parserOutput.txt. The parser is a top-Down LL1 parser that uses the lexer to get the tokens and then checks the syntax of the code. The parser is built using the the grammar rules of the language, which are made sure to be removed of left recursion and left factoring then first and follow sets are created to make a LL1 table that is used to step through the code. The parser is top-Down because we start from the grammer and compare it with the source code, instead of starting with the source code and going through the grammer.

We compare each token in the source code with the grammer rules, causing us to go through a series of rules and ending up at a non-terminal which is compared with the source code. If the source code matches the non-terminal we move on to the next token by popping the top of the stack which removes the top rule, if not we throw an error. If we reach the end of the source code and the stack is empty we have a successful parse, if not we throw an error.

./parser/parser.go
```go
if top == curr {
    stack = stack[:len(stack)-1]
}
```

The parsingTable is a 2D array, where the rows are the terminals and the columns are the non-terminals, which are added to an array which is then looped through to find the row and col.

Below is a snippet of the code that does this.

./parser/parser.go

```
terminals = append(terminals, "<program>", "<fdecls>")
nonTerminals = append(nonTerminals, "%", "=")
...
row := getIndex(top, terminals)
col := getIndex(curr, nonTerminals)
value := parsingTable[row][col]
...
var parsingTable = [][]string{}
```

If the source code input dosen't match the top of the stack it means we are at a grammer rule, so we get the next grammer rule based on the source code input we are on and the grammer rule we are on. To get the row we pass the top of the stack which is an array that is updated with the current grammer rule. We then pass the terminals array and the nonTerminals array to the getIndex function which returns the index of the terminal and nonTerminal in the array. We then pass the row and col to the parsingTable to get the value which is the next rule to be used.

./parser/parser.go

```
row := getIndex(top, terminals)
col := getIndex(curr, nonTerminals)
value := parsingTable[row][col]

if value == "-1" {
    flag = 1
        break
}

if value != "@" {
strArr := strings.Split(value, " ")

            strArr = reverse(strArr)

            stack = stack[:len(stack)-1]

            for _, element := range strArr {
                stack = append(stack, element)
            }
} else {
    stack = stack[:len(stack)-1]
}
```

If the value is -1 it means we have an error, if the value is @ it means we have an epsilon rule and we pop the top of the stack. If the value is not @ we split the value into an array and reverse it, then we pop the top of the stack and add the array to the stack.

The last function in our parse will help us in the next phase it sets the token of the current lexeme of source code to the grammer rule associated with it. The token is things such as if its a expression, operator, a scope starter/ender, a function call and etc.

```
./parser/parser.go
func setToken(input []Parse, index int, top string) (string) {
    if top == "<expr>"  || top == "<factor>" {
        input[index].Token = "EXPR"
    }
    ...
}
```

In this snippet we check if the grammer rule we are on is of type “” or “” and set the token to “EXPR”.

Finally we return the output of the parser to our main.go file to output parserOutput.txt and pass the results to our next phase.

Semantic Analysys (Compiler Phase 3) - Marc Mikhail

This is phase of 3 of 4 phases of building a compiler for the language ez sharp designed by Prof. Eugene Zima. Phase 3 of the compiler is the semantic analysis phase which takes the parsed output form phase 2 and checks the semantics of the code to see if it is correct, making sure types match while doing operations, and passing function params, and, makes sure variables are declared and references in the correct scope. This phase is important because it checks the meaning of the code and makes sure it is correct. We also create a symbol table which will help us create our intermediate code in phase 4.

File Structure

```
ezsharp/
    Table/
        table.txt
    keywords/
        keywords.txt
    lexer/
        lexer.go
    output/
        errors.txt
        lexerOutput.txt
        parserOutput.txt
        symbolOutput.txt
    parser/
```

```
        parser.go
    semantic/
        semantic.go
    test/
        Test1.cp
        Test2.cp
        ...
    token/
        token.go
    README.ME
    README.pdf
    go.mod
    main.exe
    main.go
    ezsharp.exe
```

Documentation

In phase 2 we set the token of each Lexeme according to the grammer rule it used, one of the things we set was the token for "SCOPE_START" and "SCOPE_END" which are used to determine the start and end of a scope.

If we see a "SCOPE_START" token we increment the current scope which creates a new row in our symbol table 2D array, if we see a "SCOPE_END" token we decrement the current scope which removes the last row in our symbol table 2D array.

./semantic/semantic.go

```go
var symbolTable [][curr]

switch Token {
    case "SCOPE_START":
        NewScope(&curr)
    case "SCOPE_END":
        ReturnScope(&symbolTable, &curr)
}

func NewScope(curr *int) {
    *curr++
}

func ReturnScope(symbolTable *[][]parser.Parse, curr *int) {
    (*symbolTable)[*curr] = nil
    *curr--
}
```

Next we have create two functions that are used to lookup and declare variables and functions, so they can be assigned, referenced and called when needed in the correct scope. An example use of the functions would be to lookup a variable in the symbol table to check whether it is declared or not, and to assign it if it is or declare if it isn't. If it is being referenced and not declared we would throw an error.

./semantic/semantic.go

```
func Lookup(symbolTable [][]parser.Parse, token parser.Parse) (int, parser.Parse) {
    scopeNum := -1
    var returnNode parser.Parse

    for i := len(symbolTable) - 1; i >= 0; i-- {
        for j := len(symbolTable[i]) - 1; j >= 0; j-- {

            value := symbolTable[i][j]
            if value.Literal == token.Literal {
                scopeNum = i
                break
           }
        }
    }
    return scopeNum
}

func Declare(symbolTable *[][]parser.Parse, curr parser.Parse, currScope int,
             tok string) {
    curr.Token = tok
    (*symbolTable)[currScope] = append((*symbolTable)[currScope], curr)
}
```

Lookup loops through our symbol table backwords and breaks at the first instance of the variable we are looking for, if it is not found we return the scope its in otherwise -1. Declare appends the variable to the symbol table in the current scope.

Finally we return the output of the semantic analysis to our main.go file to output symbolOutput.txt and pass the results to phase 4.