**Compiler for Go Langauge**

Presented by : - Abhilash ( Roll no: 1 , D-1)

- Shivam Varshney ( Roll no: 17 , D-1)

- Nihal Pereira ( Roll no: 19 , D-1)



1. **Objective** : The objective of our project is to make a compiler in Flex/Bison for the programming language Go ( by Google) . The functionalities that we are including in the compiler are :

- Generation of tokens

- Error Detection

- Producing the Symbol Table

- Showing the line number where error occurs

1. **Grammar :**

program-> PackageClause S | PackageClause Import S

PackageClause-> PACKAGE PackageName

PackageName-> ID

Import-> ImportDecl Import | ;

ImportDecl-> IMPORT ImportSpec

ImportSpec-> DOT ImportPath | PackageName ImportPath | ImportPath ;

ImportPath-> DOUBLE\_QUOTE pp DOUBLE\_QUOTE

pp-> ID pp | CO pp | NUMBER pp | MOD pp | ;

S-> Statement\_list S | ;

VarDecl-> VAR VarSpec ;

VarSpec-> identifier\_list Type | identifier\_list Type EQUAL ExprList | identifier\_list EQUAL ExprList

Type-> TYPE | TypeLit | TypeName | OPEN\_BRAC Type CLOSE\_BRAC

TypeLit-> ArrayType | FunctionType | StructType

TypeName-> ID | QualifiedIdent

QualifiedIdent-> ID DOT ID

ArrayType-> SQUA\_OPEN expn SQUA\_CLOSE Type

StructType-> STRUCT OPEN\_FLOW FieldDecl CLOSE\_FLOW

FieldDecl-> identifier\_list Type FieldDecl | ID FieldDecl | MUL TypeName FieldDecl | ;

FunctionType-> FUNC Signature

Signature-> Parameters | Parameters Result

Result-> Parameters | Type

TypeDecl-> TYP VarSpec

Parameters-> OPEN\_BRAC Parameter\_List CO CLOSE\_BRAC | OPEN\_BRAC Parameter\_List CLOSE\_BRAC | OPEN\_BRAC CLOSE\_BRAC | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CLOSE\_BRAC | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CO identifier\_list CLOSE\_BRAC

Parameter\_List-> Parameter\_Decl | Parameter\_Decl Param

Param-> CO Parameter\_Decl Param| ;

Parameter\_Decl-> Type | identifier\_list Type

FunctionDecl-> FUNC ID Signature | FUNC ID Function

Function-> Signature FunctionBody

FunctionBody-> Block

Block-> OPEN\_FLOW Statement\_list CLOSE\_FLOW

Statement\_list-> statement Statement\_list | ;

ExprList-> expn | expn CO ExprList

identifier\_list-> ID | ID CO identifier\_list | ID CO identifier\_list EQUAL expn

statement-> assign\_stat | decision\_stat | looping\_stat | return\_stat | break\_stat | continue\_stat | Declaration | Block | SimpleStmt

Declaration-> VarDecl | FunctionDecl| TypeDecl

SimpleStmt-> inDec\_stat | expression\_stat | assign\_stat | shortvarDecl

expression\_stat-> expn

shortvarDecl-> identifier\_list COLEQU ExprList

assign\_stat-> ExprList EQUAL ExprList | ID PLUS EQUAL expn | ID MUL EQUAL expn

expn-> simple\_expn eprime | primaryexpr

index-> SQUA\_OPEN expn SQUA\_CLOSE

primaryexpr-> Operand | primaryexpr Arguments | primaryexpr Selector | primaryexpr index

Operand-> Literal | OperandName | OPEN\_BRAC expn CLOSE\_BRAC

Literal-> BasicLit | CompositeLit | FunctionLit

FunctionLit-> FUNC Signature FunctionBody

BasicLit-> NUMBER

CompositeLit-> LiteralType LiteralValue

LiteralType-> StructType | ArrayType | SQUA\_OPEN NUMBER SQUA\_CLOSE Type

LiteralValue-> OPEN\_FLOW CLOSE\_FLOW | OPEN\_FLOW ElementList CLOSE\_FLOW | OPEN\_FLOW ElementList CO CLOSE\_FLOW

ElementList-> KeyedElement | KeyedElement CO ElementList

KeyedElement-> Element | Key COLON Element

Key-> ID | expn | LiteralValue

Element-> expn | LiteralValue

OperandName-> ID | QualifiedIdent

Arguments-> OPEN\_BRAC CLOSE\_BRAC | OPEN\_BRAC ExprList CLOSE\_BRAC | OPEN\_BRAC Type CLOSE\_BRAC| OPEN\_BRAC Type CO ExprList CLOSE\_BRAC| OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CLOSE\_BRAC | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CO identifier\_list CLOSE\_BRAC

Selector-> DOT ID

eprime-> relop simple\_expn | ;

simple\_expn-> term seprime

seprime-> addop term seprime | ;

term-> factor tprime

tprime-> mulop factor tprime | ;

factor-> ID | NUMBER

decision\_stat-> IF expn Block dprime

dprime-> ELSE Block | ELSE decision\_stat | ;

looping\_stat-> FOR ForClause Block | FOR Condition Block | FOR Block

ForClause-> stm l SS exprTT SS stml | ;

Condition-> expn | ;

stml-> SimpleStmt | ;

inDec\_stat-> expn DPLUS | expn DMINUS

exprTT-> expn | ;

return\_stat-> RETURN | RETURN ExprList

break\_stat-> BREAK

continue\_stat-> CONTINUE

relop-> GREATER\_THAN | GREATER\_THAN\_EQUAL | LESSER\_THAN | LESSER\_THAN\_EQUAL | EQUALS | NOT\_EQUALS

addop-> PLUS | MINUS

mulop-> DIV | MUL | MOD

1. **Language used for implementation** : Flex, Bison
2. **Type of Parser** : Bottom up Parser
3. **Methedology :** Parsing is the process of matching grammar symbols to elements in the input data, according to the rules of the grammar. The parser obtains a sequence of tokens from the lexical analyzer, and recognizes it’s structure.

Bottom –up parser generates the parse tree bottom-up. Given the string to be parsed and the set of productions, it traces a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol.

Symbol table stores the identifiers and labels when they are first encountered during parsing action.

1. User Documentation Readme.txt: C compiler for the go language. The main parser of the file is found in the a.y bison file. This has the symbol table code imported as a header. The tab.h of this file is imported in the lex file for token generation. The symbol table creates and pints the table and its entries. The symbol table code is compiled and headers are taken out. Then the bison code is compiled with bison to generate a.tab.c and a.tab.h. The header file is used in the flex file a.l. Then the lex file after flex compilation giving lex.yy.c is compiled together with the header file of the bison, a.tab.c.

Commands

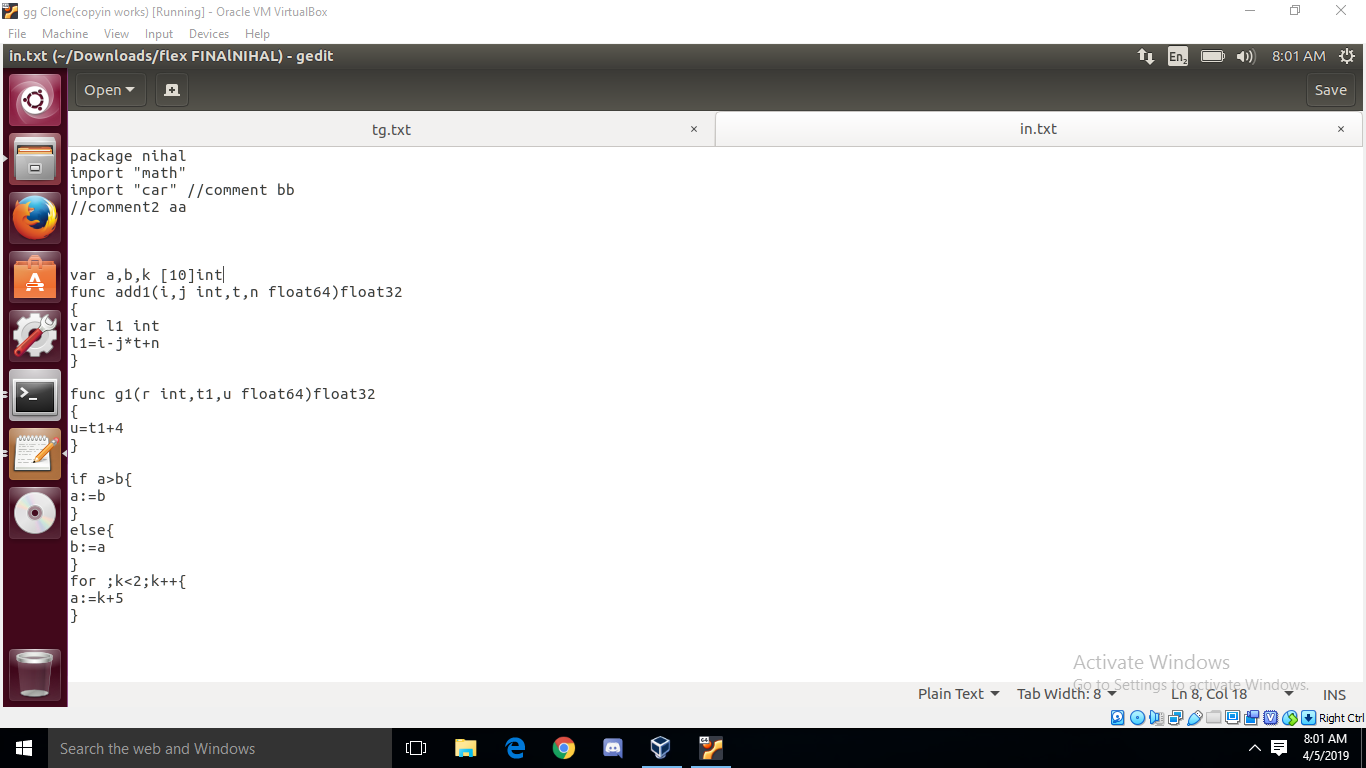
* bison -d a.y
* flex a.l

gcc lex.yy.c a.tab.c -o a

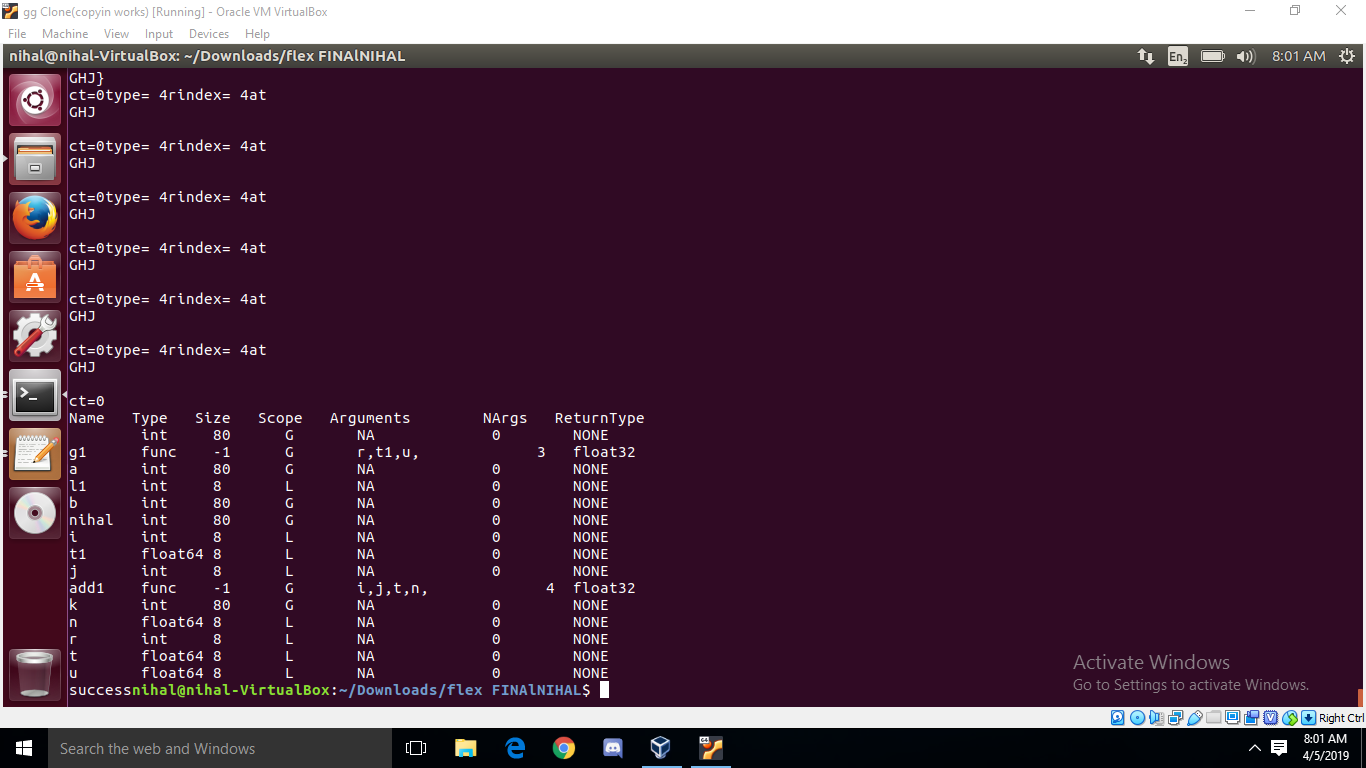
1. Code

* Symbol table
* #include <stdio.h>
* #include <string.h>
* #include <stdlib.h>
* #include <ctype.h>
* #define TableLength 30
* int row=1,col;
* char ca,cb,buf[10];int i=0;
* FILE \*fp,\*fout;
* char \*keywords[]={"break","default","func","interface","select","case","defer","go","map","struct","chan","else","goto","package","switch","const","fallthrough","if","range","type","continue","for","import","return","var"};
* char \*datatypes[]={"bool","byte","complex64","complex128","float32","float64","int","int8","int16","int32","int64","rune","uint","uint8","uint16","uint32","uint64"};
* int sizes[]={1,1,8,16,4,8,8,1,2,4,8,4,8,1,2,4,8};
* int recDTindex = 0;
* int globalScope = 1;
* typedef struct token{
* char name[10];
* char type[10];
* int size;
* char scope;
* int nargs;
* char args[20];
* char returnType[10];
* }token;
* token te[5];
* int ct=0;
* int i;
* int sizeArr=1;
* struct ListElement{
* token tok;
* struct ListElement \*next;
* };
* struct ListElement \*TABLE[TableLength];
* void Initialize(){
* for( i=0;i<TableLength;i++){TABLE[i] = NULL;}
* }
* int HASH(char \*str){
* int x=0;
* for( i=0;str[i]!='\0';i++) x += str[i];
* x %= TableLength;
* return x;
* }
* int SEARCH(char \*str){
* int val = HASH(str);
* struct ListElement \* ele= TABLE[val];
* while(ele!=NULL){
* if(strcmp(ele->tok.name,str)==0) break;
* ele = ele->next;
* }
* if(ele == NULL) return 0;
* return 1;
* }
* void INSERT(token tk){
* if(SEARCH(tk.name)==1){return;}
* int val = HASH(tk.name);
* struct ListElement\* cur = (struct ListElement\*)malloc(sizeof(struct ListElement));
* cur->tok = tk;
* cur->next = NULL;
* if(TABLE[val]==NULL){
* TABLE[val] = cur;
* }
* else{
* struct ListElement \* ele= TABLE[val];
* while(ele->next!=NULL){
* ele = ele->next;
* }
* ele->next = cur;}
* }
* int getVal(char s[]){
* int res = 0;
* for( i=0;s[i]!='\0';i++){
* res \*= 10;
* res += s[i]-'0';
* }
* return res;
* }
* int getType(char buf[],int len){
* if(strcmp(buf,"package")==0)
* {
* while(ca != '\n') ca = getc(fp);
* return 0;
* }
* for( i=0;i<25;i++) {if(strcmp(buf,keywords[i])==0) return 0;}
* for( i=0;i<17;i++) {if(strcmp(buf,datatypes[i])==0){
* printf("GGG %s",buf);
* recDTindex = i;
* return 1;
* }
* }
* buf[len] = '\0';
* if(SEARCH(buf)) return 4;
* int i;
* for(i=0;i<len;i++) if(!isdigit(buf[i])) break;
* if(i==len) return 2;
* return 3;
* }
* void getArguments(token \*t){
* token temp[10];
* int ti = 0,no\_of\_args=0;
* i=0;
* while(ca!=')'){
* ca = getc(fp);
* if(isalnum(ca) || ca == '\_'){buf[i++] = ca;col++;}
* else{
* buf[i] = '\0';
* printf("buffer =%s ",buf);
* int type = getType(buf,i);
* if(type == 3||ca=='['){
* printf("inside type=3");
* no\_of\_args++;
* strcpy(te[ct].name,buf);
* strcpy(te[ct].returnType,"NONE");
* te[ct].scope = 'L';
* if(ca == '['){
* char arrsize[10]; int ai = 0;
* ca = getc(fp);
* while(ca!=']'){
* arrsize[ai++] = ca;
* ca = getc(fp);
* }
* arrsize[ai] = '\0';
* sizeArr = getVal(arrsize);
* printf("size is%d",sizeArr);
* }
* strcpy(te[ct].args,"NA");
* te[ct].nargs = 0;
* ct++;
* i=0;
* }
* if(type == 1){
* printf("inside type=1");
* printf("ct=%d ",ct);
* int z;
* for(z=0;z<ct;z++){
* strcat(t->args,strcat(te[z].name,","));
* strcpy(te[z].type,datatypes[recDTindex]);
* te[z].size = sizes[recDTindex]\*sizeArr;
* INSERT(te[z]);
* i=0;
* }
* printf("done insertion");
* ct=0;
* sizeArr=1;
* }
* memset(buf,0,sizeof(buf));
* }
* printf("this is %c",ca);
* }
* memset(buf,0,sizeof(buf));
* printf("afff");
* ca = getc(fp);
* i=0;
* while(ca!='\n' && ca!='{'){
* if(isalnum(ca) || ca == '\_'){buf[i++] = ca;col++; printf("buf %s\n",buf);}
* ca=getc(fp);
* }
* int type = getType(buf,i);
* strcpy(t->returnType,datatypes[recDTindex]);
* t->nargs=no\_of\_args;
* INSERT(\*t);
* }
* int getNextToken(){
* ca = fgetc(fp);
* if (fp == NULL){printf("Cannot open file \n");exit(0);}
* while(ca!=EOF){
* if (ca=='/'){
* cb = getc(fp);
* if (cb == '/'){
* while(ca != '\n') ca = getc(fp);
* row--;
* }
* else if (cb == '\*'){
* do{
* while(ca != '\*')
* ca = getc(fp);
* ca = getc(fp);
* }while (ca != '/');
* row--;
* }
* else{
* buf[i++]=ca;
* buf[i++]=cb;
* col += 2;
* }
* }
* else if(ca == '#' && col == 1){
* while(ca != '\n') ca = getc(fp);
* row--;
* }
* else if(ca=='\"'){
* buf[i++]=ca;
* ca = getc(fp);
* while(ca!='\"'){
* buf[i++]=ca;
* ca = getc(fp);
* }
* buf[i++] = '\"';
* buf[i++] = '\0';
* fprintf(fout,"<%s , %d , %d , string literal >\n",buf,row,col);
* col++;
* i=0;
* return 1;
* }
* else if(isalnum(ca) || ca == '\_'){buf[i++] = ca;col++;}
* else{
* buf[i] = '\0';
* if(ca=='}')
* {
* globalScope = 1;
* }
* int type = getType(buf,i);
* printf("type= %d",type);
* printf("rindex= %d",recDTindex);
* printf("at %s\n",buf);
* printf("GHJ%c\n",ca);
* if(type == 3||ca=='['){
* if(ca == '('){
* printf("ct=%d",ct);
* token te1;
* strcpy(te1.name,buf);
* strcpy(te1.type,"func");
* te1.size = -1;
* te1.scope = 'G';
* strcpy(te1.args,"");
* te1.nargs = 0;
* globalScope = 0;
* printf("callign getargs");
* getArguments(&te1);
* }else{
* strcpy(te[ct].name,buf);
* strcpy(te[ct].returnType,"NONE");
* if(globalScope) te[ct].scope = 'G'; else te[ct].scope = 'L';
* if(ca == '['){
* printf("inside [");
* char arrsize[10]; int ai = 0;
* ca = getc(fp);
* while(ca!=']'){
* arrsize[ai++] = ca;
* ca = getc(fp);
* }
* arrsize[ai] = '\0';
* sizeArr = getVal(arrsize);
* printf("size is%d",sizeArr);
* }
* strcpy(te[ct].args,"NA");
* te[ct].nargs = 0;
* ct++;
* }
* }
* if(type == 1){
* int z;
* for(z=0;z<ct;z++){
* strcpy(te[z].type,datatypes[recDTindex]);
* te[z].size = sizes[recDTindex]\*sizeArr;
* INSERT(te[z]);
* }
* ct=0;
* sizeArr=1;
* }
* printf("ct=%d",ct);
* if(ca=='\n'){row++;col = 1;}
* i=0;
* return 1;
* }
* if(ca == '\n'){row++;col = 1;}
* ca = fgetc(fp);
* }
* if(ca==EOF) return 0;
* return 1;
* }
* void printSymbolTable(){
* printf("\nName Type Size Scope Arguments NArgs ReturnType\n");
* for( i=0;i<TableLength;i++){
* struct ListElement \*e = TABLE[i];
* while(e!=NULL){
* token t = e->tok;
* printf("%s\t%s\t%d\t%c\t%s\t\%d\t%s\n",t.name,t.type,t.size,t.scope,t.args,t.nargs,t.returnType);
* e = e->next;
* }
* }
* }
* flex code
* %option yylineno
* %{
* #include <stdio.h>
* #include <stdlib.h>
* #include "a.tab.h"
* %}
* %%
* \n {}
* [0-9]+ {printf("%s\n", yytext); return NUMBER;}
* \[ {printf("[\n"); return SQUA\_OPEN;}
* \] {printf("]\n"); return SQUA\_CLOSE;}
* "++" {printf("++\n"); return DPLUS;}
* "--" {printf("--\n"); return DMINUS;}
* "-" {printf("-\n"); return MINUS;}
* "+" {printf("+\n"); return PLUS;}
* "\*" {printf("\*\n"); return MUL;}
* "/" {printf("/\n"); return DIV;}
* "%" {printf(" MOD\n"); return MOD;}
* ";" {printf(";\n"); return SS;}
* ":=" {printf(":=\n"); return COLEQU;}
* "," {printf(",\n"); return CO;}
* "=" {printf("=\n"); return EQUAL;}
* ">" {printf(">\n"); return GREATER\_THAN;}
* ">=" {printf(">=\n"); return GREATER\_THAN\_EQUAL;}
* "<" {printf("<\n"); return LESSER\_THAN;}
* "<=" {printf("<=\n"); return LESSER\_THAN\_EQUAL;}
* "==" {printf("==\n"); return EQUALS;}
* "!=" {printf("!=\n"); return NOT\_EQUALS;}
* "(" {printf("(\n"); return OPEN\_BRAC;}
* ")" {printf(")\n"); return CLOSE\_BRAC;}
* "{" {printf("{\n"); return OPEN\_FLOW;}
* "}" {printf("}\n"); return CLOSE\_FLOW;}
* "." {printf(".\n"); return DOT;}
* ":" {printf(":\n"); return COLON;}
* \" {printf("DC\n"); return DOUBLE\_QUOTE;}
* "var" {printf("var\n"); return VAR;}
* "type" {printf("type\n"); return TYP;}
* "import" {printf("import\n"); return IMPORT;}
* "package" {printf("package\n"); return PACKAGE;}
* "func" {printf("func\n"); return FUNC;}
* "struct" {printf("struct\n"); return STRUCT;}
* "break" {printf("break\n"); return BREAK;}
* "continue" {printf("continue\n"); return CONTINUE;}
* "return" {printf("return\n"); return RETURN;}
* "int" {printf("%s\n", yytext); return TYPE;}
* "bool" {printf("%s\n", yytext); return TYPE;}
* "int64" {printf("%s\n", yytext); return TYPE;}
* "int32" {printf("%s\n", yytext); return TYPE;}
* "float32" {printf("%s\n", yytext); return TYPE;}
* "float64" {printf("%s\n", yytext); return TYPE;}
* "main" {printf("%s\n", yytext); return MAIN;}
* "while" {printf("%s\n", yytext); return WHILE;}
* "for" {printf("%s\n", yytext); return FOR;}
* "if" {printf("if\n"); return IF;}
* "else" {printf("else\n"); return ELSE;}
* [a-zA-Z][a-zA-Z0-9\_]\* {printf("%s\n", yytext); return ID;}
* . {}
* %%
* int yywrap(){
* return 1;
* }
* int yyerror(char \*msg){
* printf("%d: %s at '%s' Invalid Expression\n",yylineno,msg,yytext);
* return 1;
* }
* bison
* %{
* #include <stdio.h>
* #include <stdlib.h>
* #include "q1.h"
* int yylex();
* int yyerror();
* extern FILE \*yyin;
* %}
* %token NL PACKAGE IMPORT TYP ID COLON NUMBER DOT COLEQU DOUBLE\_QUOTE VAR STRUCT FUNC RETURN BREAK CONTINUE DPLUS DMINUS IF GREATER\_THAN GREATER\_THAN\_EQUAL LESSER\_THAN LESSER\_THAN\_EQUAL EQUALS ELSE NOT\_EQUALS SS CO OPEN\_FLOW CLOSE\_FLOW OPEN\_BRAC CLOSE\_BRAC MAIN TYPE SQUA\_OPEN SQUA\_CLOSE PLUS MINUS MUL DIV MOD WHILE FOR EQUAL
* %right EQUAL
* %precedence NOT\_EQUALS
* %left PLUS MINUS
* %left MUL DIV
* %%
* program: PackageClause S
* | PackageClause Import S
* ;
* PackageClause: PACKAGE PackageName
* ;
* PackageName: ID
* ;
* Import: ImportDecl Import
* |
* ;
* ImportDecl: IMPORT ImportSpec
* ;
* ImportSpec: DOT ImportPath
* | PackageName ImportPath
* | ImportPath
* ;
* ImportPath: DOUBLE\_QUOTE pp DOUBLE\_QUOTE
* ;
* pp: ID pp
* | CO pp
* | NUMBER pp
* | MOD pp
* |
* ;
* S: Statement\_list S
* |
* ;
* VarDecl: VAR VarSpec
* ;
* VarSpec: identifier\_list Type
* | identifier\_list Type EQUAL ExprList
* | identifier\_list EQUAL ExprList
* ;
* Type: TYPE
* | TypeLit
* | TypeName
* | OPEN\_BRAC Type CLOSE\_BRAC
* ;
* TypeLit: ArrayType
* | FunctionType
* | StructType
* ;
* TypeName: ID
* | QualifiedIdent
* ;
* QualifiedIdent: ID DOT ID
* ;
* ArrayType: SQUA\_OPEN expn SQUA\_CLOSE Type
* ;
* StructType: STRUCT OPEN\_FLOW FieldDecl CLOSE\_FLOW
* ;
* FieldDecl: identifier\_list Type FieldDecl
* | ID FieldDecl
* | MUL TypeName FieldDecl
* |
* ;
* FunctionType: FUNC Signature
* ;
* Signature: Parameters
* | Parameters Result
* ;
* Result: Parameters
* | Type
* ;
* TypeDecl: TYP VarSpec
* ;
* Parameters: OPEN\_BRAC Parameter\_List CO CLOSE\_BRAC
* | OPEN\_BRAC Parameter\_List CLOSE\_BRAC
* | OPEN\_BRAC CLOSE\_BRAC
* | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CLOSE\_BRAC
* | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CO identifier\_list CLOSE\_BRAC
* ;
* Parameter\_List: Parameter\_Decl
* | Parameter\_Decl Param
* ;
* Param: CO Parameter\_Decl Param
* |
* ;
* Parameter\_Decl: Type
* | identifier\_list Type
* ;
* FunctionDecl: FUNC ID Signature
* | FUNC ID Function
* ;
* Function: Signature FunctionBody
* ;
* FunctionBody: Block
* ;
* Block: OPEN\_FLOW Statement\_list CLOSE\_FLOW
* ;
* Statement\_list: statement Statement\_list
* |
* ;
* ExprList: expn
* | expn CO ExprList
* ;
* identifier\_list: ID
* | ID CO identifier\_list
* | ID CO identifier\_list EQUAL expn
* ;
* statement: assign\_stat
* | decision\_stat
* | looping\_stat
* | return\_stat
* | break\_stat
* | continue\_stat
* | Declaration
* | Block
* | SimpleStmt
* ;
* Declaration: VarDecl
* | FunctionDecl
* | TypeDecl
* ;
* SimpleStmt: inDec\_stat
* | expression\_stat
* | assign\_stat
* | shortvarDecl
* ;
* expression\_stat: expn
* ;
* shortvarDecl: identifier\_list COLEQU ExprList
* ;
* assign\_stat: ExprList EQUAL ExprList
* | ID PLUS EQUAL expn
* | ID MUL EQUAL expn
* ;
* expn: simple\_expn eprime
* | primaryexpr
* ;
* index: SQUA\_OPEN expn SQUA\_CLOSE
* ;
* primaryexpr: Operand
* | primaryexpr Arguments
* | primaryexpr Selector
* | primaryexpr index
* ;
* Operand: Literal
* | OperandName
* | OPEN\_BRAC expn CLOSE\_BRAC
* ;
* Literal: BasicLit
* | CompositeLit
* | FunctionLit
* ;
* FunctionLit: FUNC Signature FunctionBody
* ;
* BasicLit: NUMBER
* ;
* CompositeLit: LiteralType LiteralValue
* ;
* LiteralType: StructType
* | ArrayType
* | SQUA\_OPEN NUMBER SQUA\_CLOSE Type
* ;
* LiteralValue: OPEN\_FLOW CLOSE\_FLOW
* | OPEN\_FLOW ElementList CLOSE\_FLOW
* | OPEN\_FLOW ElementList CO CLOSE\_FLOW
* ;
* ElementList: KeyedElement
* | KeyedElement CO ElementList
* ;
* KeyedElement: Element
* | Key COLON Element
* ;
* Key: ID
* | expn
* | LiteralValue
* ;
* Element: expn
* | LiteralValue
* ;
* OperandName: ID
* | QualifiedIdent
* ;
* Arguments: OPEN\_BRAC CLOSE\_BRAC
* | OPEN\_BRAC ExprList CLOSE\_BRAC
* | OPEN\_BRAC Type CLOSE\_BRAC
* | OPEN\_BRAC Type CO ExprList CLOSE\_BRAC
* | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CLOSE\_BRAC
* | OPEN\_BRAC DOUBLE\_QUOTE pp DOUBLE\_QUOTE CO identifier\_list CLOSE\_BRAC
* ;
* Selector: DOT ID
* ;
* eprime: relop simple\_expn
* |
* ;
* simple\_expn: term seprime
* ;
* seprime: addop term seprime
* |
* ;
* term: factor tprime
* ;
* tprime: mulop factor tprime
* |
* ;
* factor: ID
* | NUMBER
* ;
* decision\_stat: IF expn Block dprime
* ;
* dprime: ELSE Block
* | ELSE decision\_stat
* |
* ;
* looping\_stat: FOR ForClause Block
* | FOR Condition Block
* | FOR Block
* ;
* ForClause: stml SS exprTT SS stml
* |
* ;
* Condition: expn
* |
* ;
* stml: SimpleStmt
* |
* ;
* inDec\_stat: expn DPLUS
* | expn DMINUS
* ;
* exprTT: expn
* |
* ;
* return\_stat: RETURN
* | RETURN ExprList
* ;
* break\_stat: BREAK
* ;
* continue\_stat: CONTINUE
* ;
* relop: GREATER\_THAN
* | GREATER\_THAN\_EQUAL
* | LESSER\_THAN
* | LESSER\_THAN\_EQUAL
* | EQUALS
* | NOT\_EQUALS
* ;
* addop: PLUS
* | MINUS
* ;
* mulop: DIV
* | MUL
* | MOD
* ;
* %%
* void main(){
* yyin = fopen("in.txt", "r");
* do{
* if(yyparse()){
* printf("\n Failure");
* exit(0);
* }
* }while(!feof(yyin));
* fp=fopen("in.txt","r");
* fout = fopen("out.c","w");
* col = 1;
* Initialize();
* while(getNextToken());
* printSymbolTable();
* printf("success");
* }

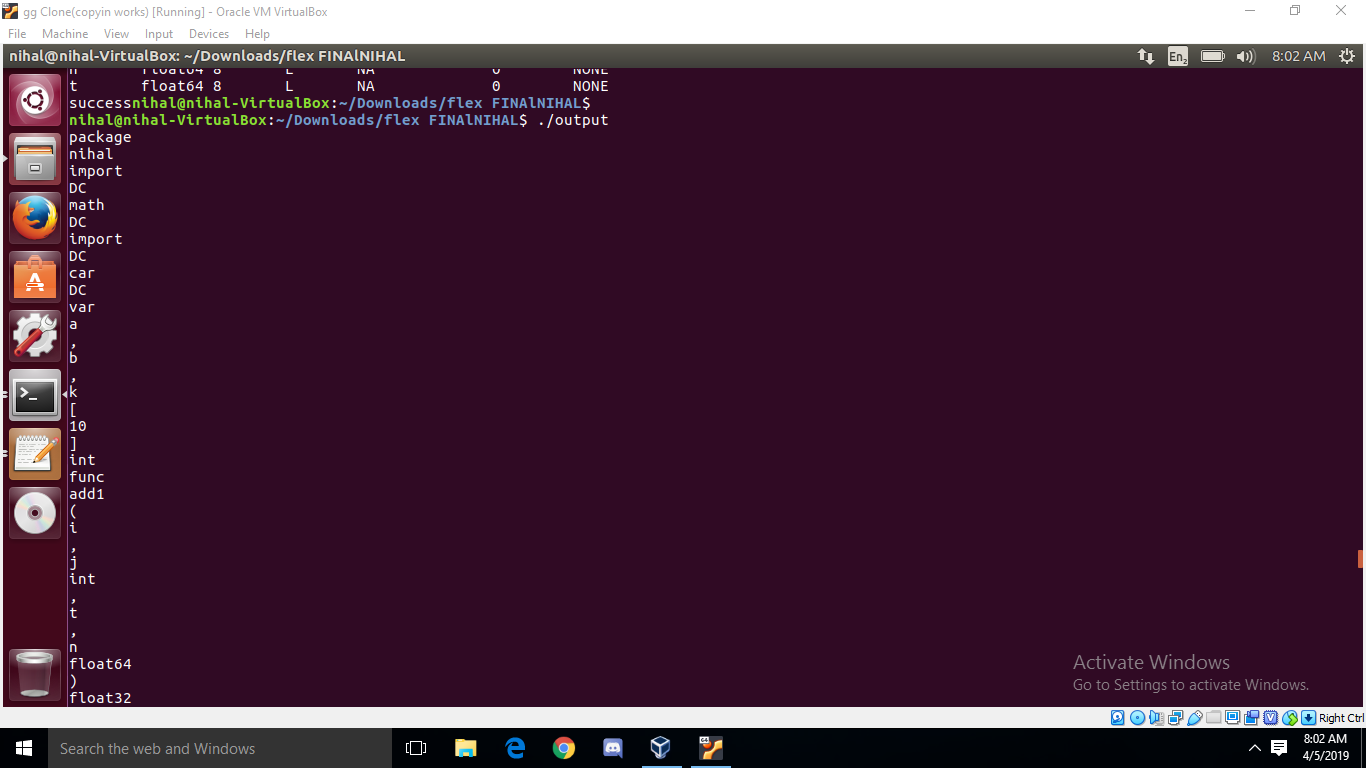
**Input and Output**

****

**Symbol Table**

****

**Final Output**

****

