GOLD Final Report

Language Guru: Timothy E. Chung (tec2123)
System Architect: Aidan Rivera (ar3441)
Manager: Zeke Reyna (eer2138)
Tester: Dennis Guzman (drg2156)

October 16th, 2017

Table of Contents

1 Introduction

NOTE: Typestructs were not developed by the turn-in date of the project, and would be a future implementation*****

GOLD is a language very similar in syntax to C. We aim to create a language in which the description of command-line games may be made easier. In order to fulfill our goal we have placed special importance a form of structure which we have appropriately named a “type-struct”. This associates a name to the type of struct which can hold named fields to describe more abstract concepts. Functions, too, have an alteration in which they can be associated with a particular type of type-struct, and can access its members through the keyword “self”. We hope that these two additions into a C-like language, bringing an association of actions with some form of entities, can aid in the creation of command line games.
We supply default functions with which to interact with strings and the command line interface, and in the future we will include a package which would supply a default or suggested interface of functions.

GOLD aims to be a simple language that takes advantage of the concept of struct, pointers, and easy pointer manipulation by removing the complex rules of pointer manipulation one sees in C or C++. It allows for creation of type struct and at the same time, struct function/method that one can use for a specific struct. This allows us to take advantage of object oriented concept using structs. GOLD aims to keep the standard library as thin as possible like C so it is easy to learn and start with.

2 Tutorial

Run make in the unzipped folder. This should give you a gold.native executable. Once you have the gold.native executable, run compile.sh with your custom .gold program as a parameter. This should give you an executable that will run your program.

3 Reference Manual

3.1 Lexical Conventions/Tokens

3.1.1 Comments

Comments in GOLD occur between the token “//” and the end of the line.

3.1.2 Identifiers

An identifier in GOLD consists of an alphabetic character or underscore followed by a sequence of alphanumeric characters and/or underscores “_”.

3.1.3 Keywords

These identifiers are reserved and can only be used as keywords:

`if`	`void`	`else`	`func`	`return`	`for`
`int`	`string`	`float`	`true`	`false`	`type`
`bool`	`break`	`continue`	`null`	`print`	`input`
`struct*`	`self*`	`elif*`			

*these keywords are not fully functional

3.1.4 Literals

3.1.4.1 Integer Literals

An integer consists of a sequence of digits and will be represented as that string’s decimal value.

3.1.4.2 Floating Point Literals

A float will be represented by a sequence of one or more digits, a decimal point, and a sequence of one or more digits. At least the integer part must be present in addition to the decimal point, but can take a value of 0.

3.1.4.3 Character Literals

A character literal will consist of a single character, or a backslash followed by one of {`“, r, n, t,`} or a sequence of digits whose decimal value falls between 0 and 127, inclusively, forming a character’s ASCII representation. “ `\”` ” , “ `\r` ”, “ `\n` ”, and “ `\t` ” have their conventional meaning.

3.1.4.4 Boolean Literals

A boolean literal can be either of the keywords `true` and `false`.

3.1.4.5 String Literals

String literals consist of a sequence of characters surrounded by “”. Any of the escapes in 2.4.3 can be included in the string.
The creation of new strings from other variables is covered in the library functions section (8.2).

3.2 Types

CFG:

typ -> | Bool | Int | Float | String | Array(typ, Int option) | Void | Pointer(typ)

3.2.1 Primitive Types

`bool`: Llvm.i1_type
A boolean value, can take either a true, false, or null value

`int`: Llvm.i32_type
An integer numerical value

`float`: Llvm.float_type
A floating point numerical value, consisting of a decimal point and at least one of either an integer or fractional part

`string`: Llvm.pointer_type(Llvm.i8_type)
A sequence of characters as defined in (2.4.5).

3.2.2 Derived

`Arrays`: Llvm.array_type(typ, Int) or Llvm.pointer_type(typ)
Represented as [], a 0-indexed list of predefined-typed variables of fixed size

`Pointers`: Llvm.pointer_type(typ)
Stores address location of any other type in memory

******FUTURE*****3.2.3 Type-Structs*******

A GOLD specific concept, these consist of a set of capitalized, named strings, which map to values of predefined type. All are required at definition, but can be instantiated to null.

Ex.
```
type User struct {
      Name string;
      Age int;
}
User U = {Name=”Stephen”; Age=null;};
```

The fields and type-name of the type-struct must be capitalized at definition.
****

3.3 Operators

CFG:

expr -> | IntLit | StringLit | FloatLit | True | False | ID
| ID = expr | ID = ArrayLit | ID(actuals) | (expr)
| ID[expr] | lvalue = expr | and the operations specified in this section
lvalue -> | ID[expr] | ~expr | @expr

3.3.1 Unary Operators

`-expr (num)`

Evaluates to the negation of either a floating point or integer value; -0 -> 0

`!expr (boolean, boolean expression)`

Evaluates to the negation of a boolean value,

```
!true == false // returns true
!false == true // returns true
```

`@expr (ID)`

Evaluates to the address of the identifier in memory if the expression is an identifier, otherwise return an error, the expression can also be a function

`~expr (ID)`

Evaluates to an identifier to the object or literal that identifier was pointing to. If the identifier doesn't reference a pointer, then a dereference error will be thrown.

3.3.2 Numeric Binary Operators

`expr + expr (num)`

Adds either float to float or int to int

`expr - expr (num)`

Subtracts the right numerical value from the left numerical value

`expr / expr (num)`

Divides the right value from the left, returning its integer value if integers are used

`expr * expr (num)`

Multiplies the numerical values

`expr % expr (num)`

Takes the remainder of division of the left value by the right

These group from left to right and do not have side-effects on the variables involved unless they evaluate to the right hand expression of an assignment. These operations can not be crossed between ints and floats, as GOLD is a strongly typed language.

3.3.3 Relational Operators

`expr < expr` (num)	Less than
`expr > expr` (num)	Greater than
`expr >= expr` (num)	Greater than or equal to
`expr <= expr` (num)	Less than or equal to

These operators return true if the relation is true and the types are the same, and false otherwise. If types are mismatched an error is thrown.

3.3.4 Equality Operators

`expr == expr (num)`

Returns true if the numbers are equal to each other and the same type, false if they are not equal, and an error for a type mismatch.

`expr != expr (num)`

Returns true if the numbers are not equal, false if they are, and an error for a type mismatch.

3.3.5 Logical Operators

`expr && expr (boolean)`

Returns true if both the expressions are true, and false otherwise.

`expr || expr (boolean)`

Returns true if either expr is true, and false otherwise.

3.3.6 Assignment Operator

```
ID = expr
ID = array_lit
lvalue = expr
```

Replaces the left-hand value with the expression if their types match, otherwise it throws up an error. If also declaring a variable whilst assigning a value to it, the type of the variable must be prepended to it.

Ex. `<variable-type> <variable-name> = expr`

3.3.7 Subset Operator

`<Array>[<index value>]`

This operator requires an identifier in the <Array> position, and an int i that satisfies 0 < i < length_of_array for the <index value>. Expression would evaluate to the 0-indexed value of the list.

3.3.8 Precedence of Operators

Order of Precedence is from to to bottom,

`<Operator Type>`	`<Operator>`	<Associativity>
`Subset Operator`	`[ ]`	left-to-right
`Unary Operators`	`*, &`	right-to-left
	`!, -(negation)`	
`Numeric Binary Operators`	`+, -(subtraction)`	left-to-right
	`*, /, %`	
`Relational Operators`	`<, >, <=, >=`	left-to-right
	`==, !=`	
`Logical Operators`	`&&, ||`	left-to-right
`Assignment Operator`	`=`	right-to-left

3.4 Declarations

All declared variables have space allocated to them on the stack; these variables are null until defined. Defining a variable more than once with the same name will throw an error.

3.4.1 Type specifiers

```
int, string, float, bool, void, array[<size>]
      ****FUTURE***type <name> struct***
```

"`void`" should be specified as the return type of a function if no value is to be returned from it.
If a struct is declared, a type name must be specified as a sequence of alphanumeric digits or "_" that begins with a capitalized alphabetic character. If assigning a variable as a struct, it is proper convention to capitalize the beginning of the variable's name, but is not required.

These type specifiers define types of variable, parameter types and return types. "void" is exclusively a return type.

3.4.2 Object Declarators

Each object declaration statement takes the form of

```
      <type> <variable-name>;
```

Where the variable-name is the identifier of the object. This holds true of all types but “`void`”.

3.4.3 Function Declarators

```
func <function-name>(arg-name arg-type, … ) <return-type> {... }
***FUTURE**func [type-struct type] <function-name>(arg-name arg-type, … )
<return-type> {... }*****
```

A function declaration consists of the func keyword followed by an optional type-struct association, the function name, parentheses optionally filled by a sequence of argument names and types, separated by commas. The return type must be specified after, with statements in the function surrounded by braces. ****FUTURE***If the function has a type-struct association, the function is to be used by calling the defined variable’s identifier of the type-struct, appended with a “.”, and then appended with the function name, parentheses, and any required arguments; see (7.2 for example).******

3.4.4 Array Declarator

Ex. `<type-name>[<size>] <identifier>;`
Or
Ex. `[<type-name>[<size>] ] <identifier> = {<value>, ….};`

The first example shows the declaration of an array of type “type-name”; the size (in number of values) must be specified. The second illustrates the declaration and definition of an array. If the array identifier is already instantiated as the right size and type, the <type-name>[<size>] will throw an error, as a variable can not be declared more than once.

3.5 Statements

CFG:

```
      stmt -> expr; | vdecl_stmt | return; | return expr; |
      | if (expr) stmt
      | if (expr) stmt else stmt
      | for (expr_opt; expr; expr_opt ) stmt
      | {stmt_list}

      vdecl_stmt -> typ ID; | typ ID = expr; | typ ID = array_lit;
```

expr_opt -> Noexpr | expr

3.5.1 Assignment Statement

An assignment statement can take the form of
Ex. [<identifier-type>] <identifier> = expr
Where if the <identifier> is already declared, the <identifier-type> can be omitted, but the expression must be of the correct type. If the <identifier-type> is specified, the expression's type can overwrite the previous type of the identifier.

3.5.2 Function-Call Statement

*** NOT IMPLEMENTED, FOR THE FUTURE ***
If a function is defined as having a type-struct association, a function is called in the format
Ex. <type-struct identifier>.<function-name>([correctly typed arguments, ….]);
Or
Ex. [identifier type] <identifier> = ...(see above)
and the type-struct variable and variable members will be accessible by the keyword "self" in the function.
****

If a function does not have a type-struct association, then a function-call statement takes the form of
Ex. <function-name>([correctly typed argument, correctly typed argument, ...]);
Or
Ex. [identifier type(optional)] <identifier> = <function-name>([args...]);

The latter function-call in each case will assign the result of the function to the identifier, or declare an identifier of the correct type if the types match and the type is specified. If the identifier is already being used, the type specification will overwrite the variable; if no type is given and the return type of the function does not match that of the existing variable, an Error will be raised.

3.5.3 Sequence Statement

Statements occur in sequence or in singular fashion, separated by semicolons. Alternately, a semicolon can be used by itself in absence of a statement
Ex. stmt ; stmt; stmt ….; or ;

3.5.4 Control-Flow Statement

- if, elif, else

The control flow statements evaluate expressions to determine which block of statement(s) to execute. There must be at least one statement, or a semicolon in absence of statements.
Ex. if (expr) {stmt}

Ex. if (expr) {stmt} else {stmt}
Ex. if (expr) {stmt} elif (expr) {stmt} ….
Ex. if (expr) {stmt} elif (expr) {stmt} …. else {stmt}

3.5.5 Loop Statement

- for

for (initial expr; conditional expr; iterative expr) { stmt }

GOLD has no while loop, only a "for" loop. A for loop begins with its initial expression, which can be blank, can set an existing variable to some value, or can instantiate a new variable, in a C-like style. A for loop without any conditional expression will run forever like a while loop that always evaluates to true. At the end of each statement block, the conditional expr is evaluated, which if false will cause the end of the loop. Otherwise, the iterative expression is evaluated and the statement block begins again.

3.6 Library Functions

3.6.1 print

Ex. print("Hello World\n");

Prints the string value to stdout, without adding a newline character.

3.6.2 println

Ex. println("Hello World");

Prints the string value to stdout, with a newline character.

3.6.3 sprint

Ex. string s = sprint("%s am %i feet tall", "I", 6);

Returns a string from a formatted string and variable input

Signifier	Variable Type
%s	Replaces with a string
%i	Replaces with an integer
%f	Replaces with a float

%p	Replaces with an address

The maximum length returnable by sprint in our language is 4096 bytes. Sprint is linked to the C library's "sprintf" function, but instead of worrying about passing a char array as the first argument, we limited the length possible and allowed for a slight bit higher abstraction.

3.6.4 input

Ex. string s = input(); //takes user input

The GOLD library "input" function is linked to the C library's "gets" function which, although unsafe, suited our purposes. The function returns the string, without ending newline character. This function waits until the enter key is pressed to return the string representation of user input.

3.6.5 srand

Ex. srand();

The GOLD library "srand" function is linked to the C library's implementation, but instead of passing an unsigned integer to srand on which to seed, the function links to C's "time" function and passes the current time as a 32 bit llvm type, effectively removing any pointers and declarations from the user.

3.6.6 rand

Ex. int i = rand();

The GOLD "rand" function returns a 32 bit integer from 0 to the maximum allowable value. It can be seeded for more true randomness by calling "srand" once, or can be left without seeding for consistent pseudo-random output.

3.6.7 atoi

Ex. int i = atoi("45jkl");

The GOLD "atoi" function is linked to the C library and takes its handling of edge-cases. In the example above, 45 would be the output value. If no numeric value is found at the front of the string, 0 is output. The conversion effectively stops at the first non-numeric value.

3.7 Scope

Local Variables - Local variables are declared inside a function or block and can only be accessed within that function or block.

Note: {stmt; stmt} starts a nested scope, and variables declared in this scope will not be available outside of it.

Actual Parameters - These are the parameters declared in the function or method signature and can be accessed anywhere within the function/method.

Global Variables - These are the variables declared outside any function and can be accessed anywhere within the file.

4 Project Plan - Ezekiel Reyna

4.1 Team Member Roles and Responsibilities

Team Member	**Position**
Ezekiel Reyna	Manager
Aidan Rivera	System Architect
Timothy Chung	Language Guru
Dennis Guzman	Tester

For the most part, these positions weren't the defining feature of what we worked on. Many of us performed multiple roles on top of what we were initially assigned, and when crunch time came (the last couple weeks) roles more or less became meaningless as we focused on implementing features.

4.2 Planning, Developing, and Testing

Initially a lot of work was put into developing the language, figuring out our strengths and what we wanted to leverage to develop a good product. However, as the semester dragged on productivity dropped and the initial work on the specification didn't matter as the work necessary to develop the compiler wasn't being put in.

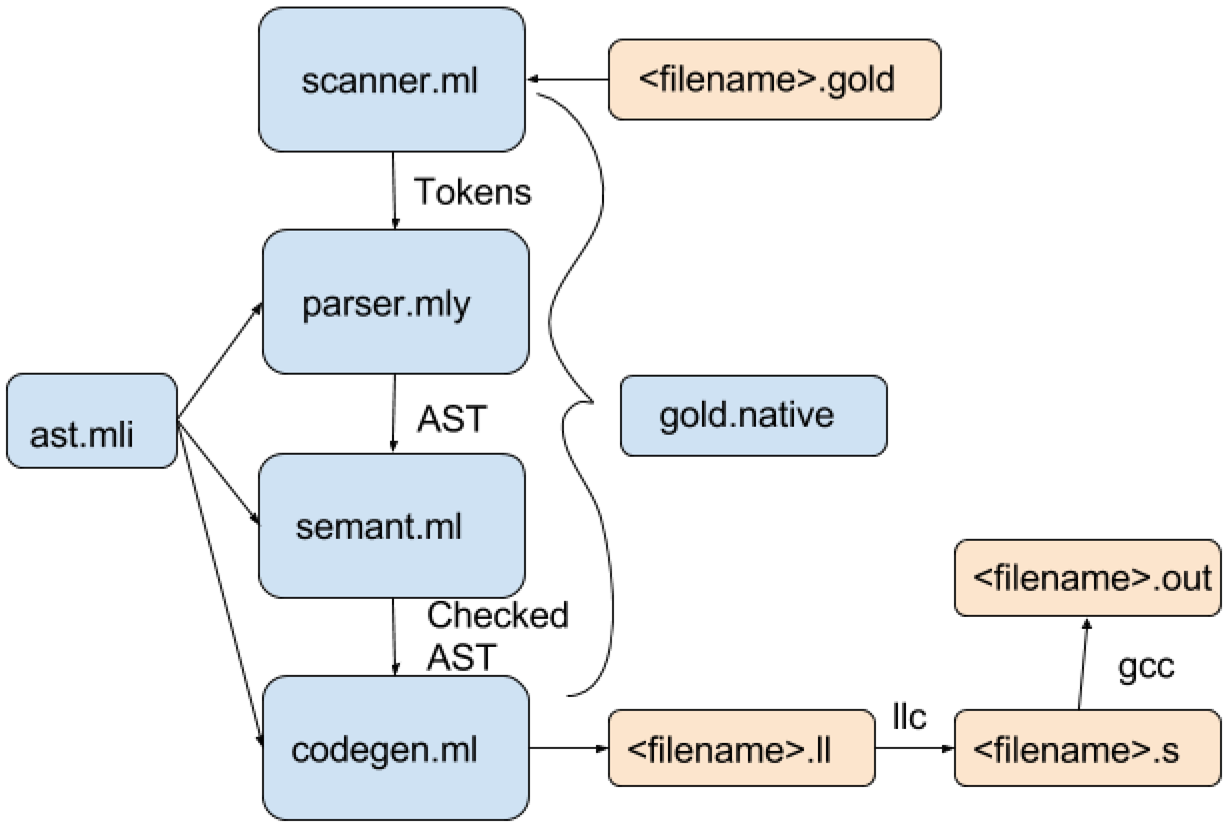
However, to keep track of work a simple todo list was setup and the team used it to keep track of which features needed implementing and where the features were lacking.

We held weekly meetings where we hammered out parts of the project specifications, and met up sporadically for pair programming. The pair programming, which always wasn't physical sessions, was incredibly important for understanding how the MicroC compiler worked, and how we could leverage this understanding to implement features in our compiler.

5 Language Evolution - Timothy Chung

6 Translator Architecture - Aidan Rivera

6.1 Block Diagram



6.2 Compiler Components

All components handle their purpose fairly rigidly, exceptions to this statement would be Errors thrown in codegen, or ast.ml supplying types and functions to many of the files.

Scanner: The scanner reads the input file character by character in order to parse the token stream into a sequence of tokens, which include those for identifiers, keywords, operators, literals of types and comments. Comments are handled at this step, as the scanner simply skips over characters in the range specified by comment conventions.

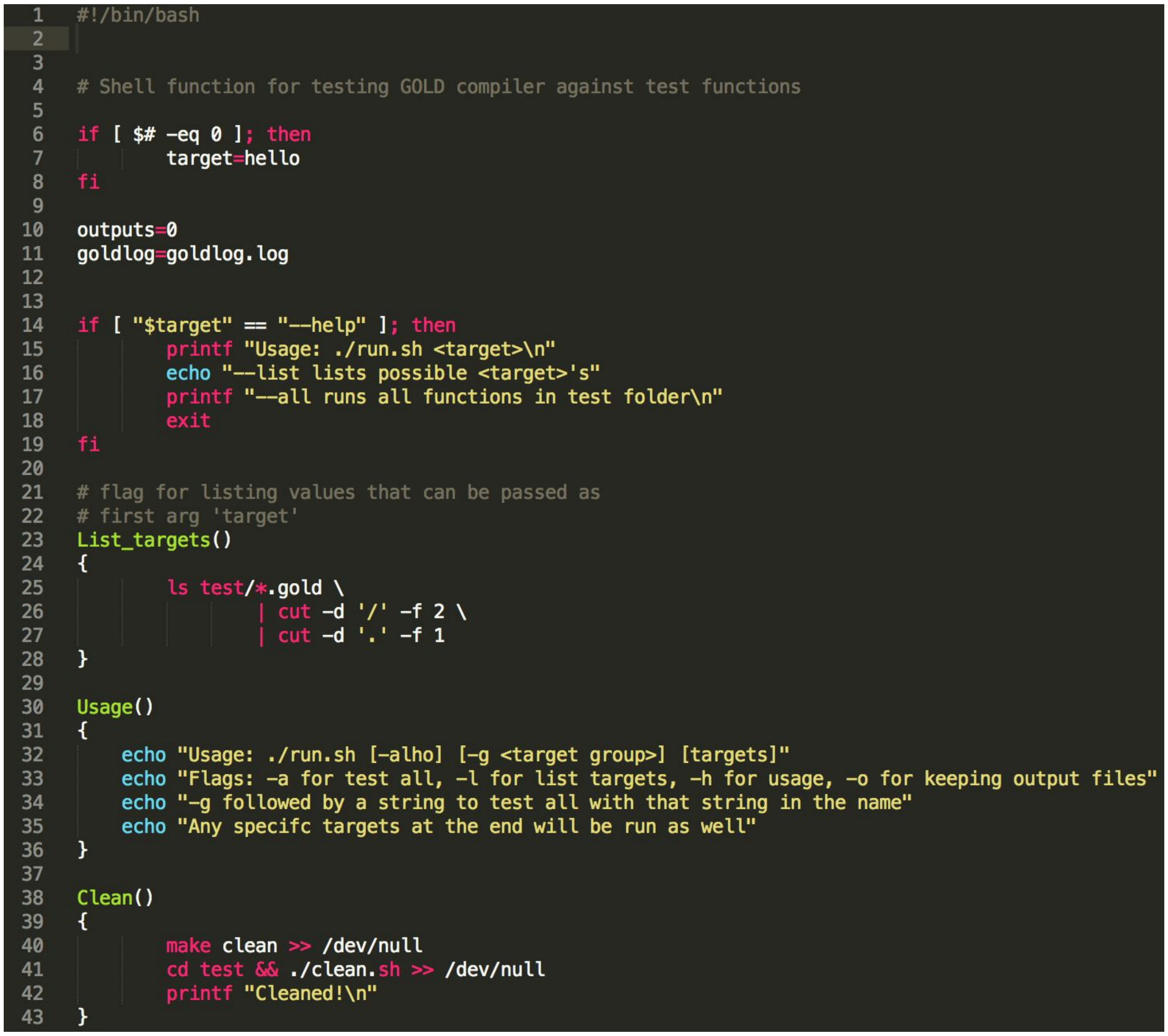
Parser: The parser takes the sequence of tokens and uses the parser's CFG to create a unique Abstract Syntax Tree. In OCaml, these leafs are associated with types outlined in "ast.mli" and the process returns a "program" type.

Semant: Takes the "program" returned by parser.mly and combs through the constructed Abstract Syntax Tree checking edge cases, ensuring that the parsed file is of a proper format to be sent to codegen. This step handles any possible error messages in regard to improper user actions.

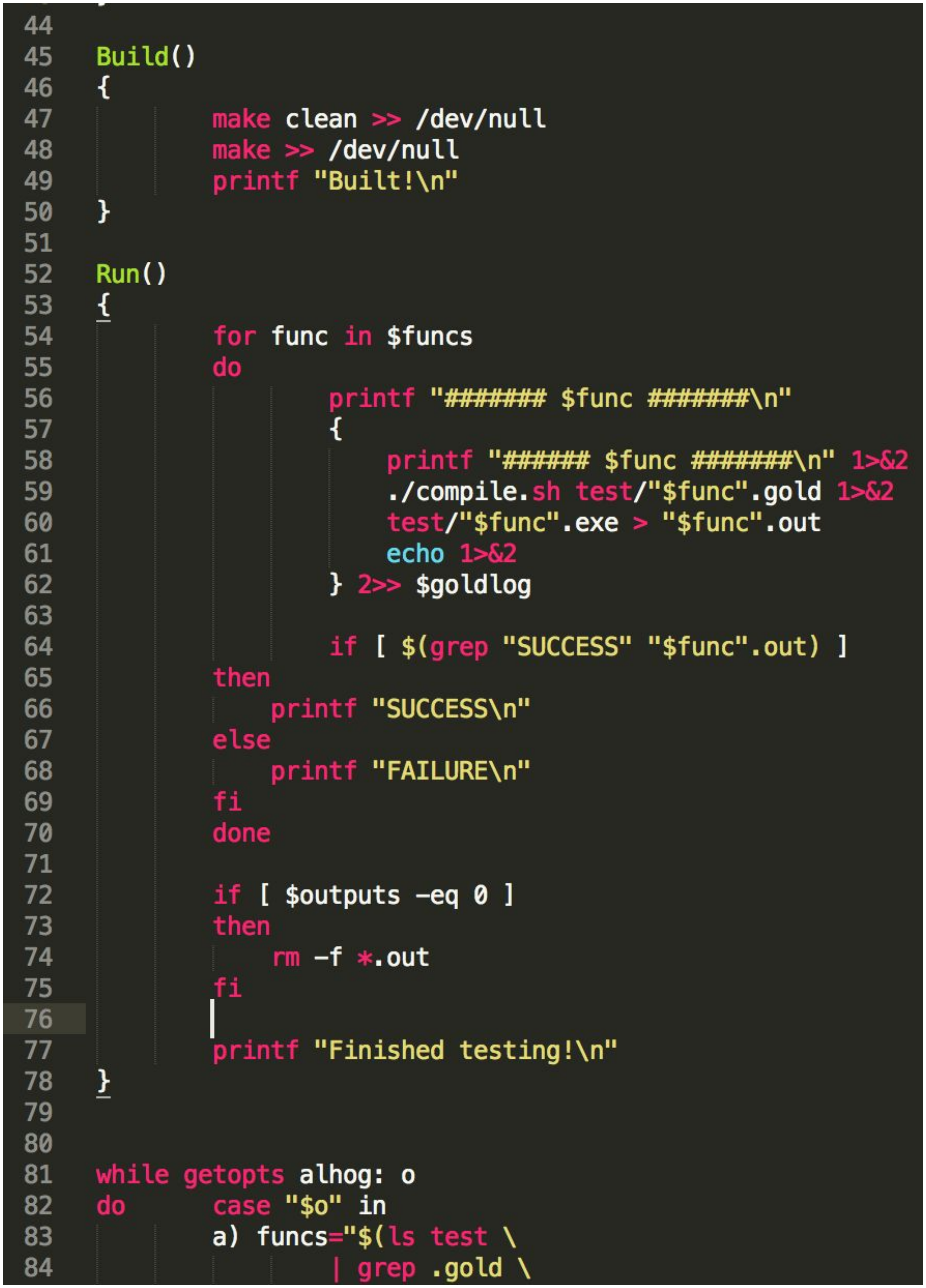
Codegen: Takes the semantically sound "program" and iterates through global declarations, then functions, to create an llvm intermediate representation of what our language interprets the input file as.

The compiled gold.native executable should handle this process for the user. Additional scripts are included to take the output .ll file, run it against llc, and then run it against gcc. From here one should have a proper gold executable of the input file.

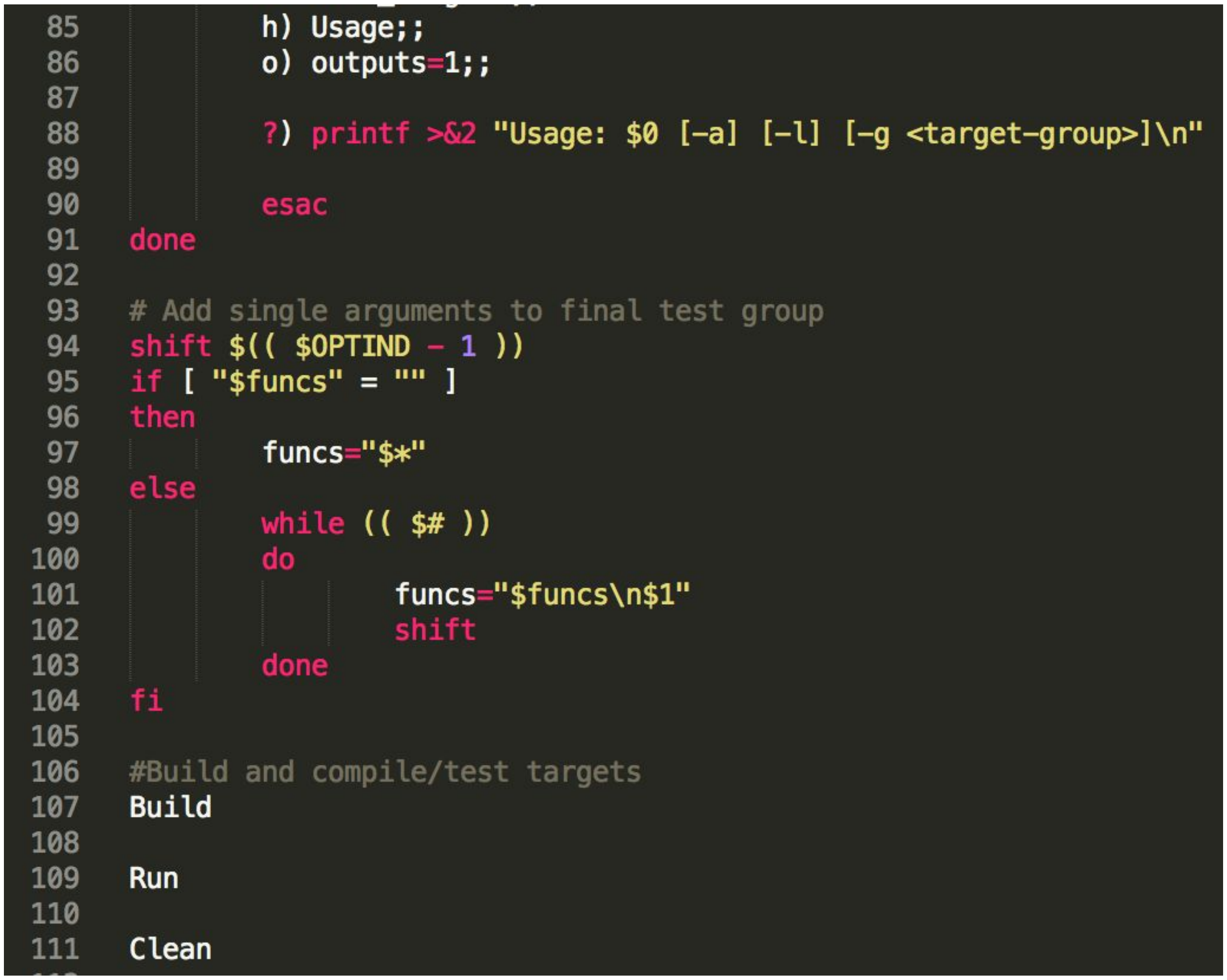
7 Test Plan and Scripts - Dennis Guzman



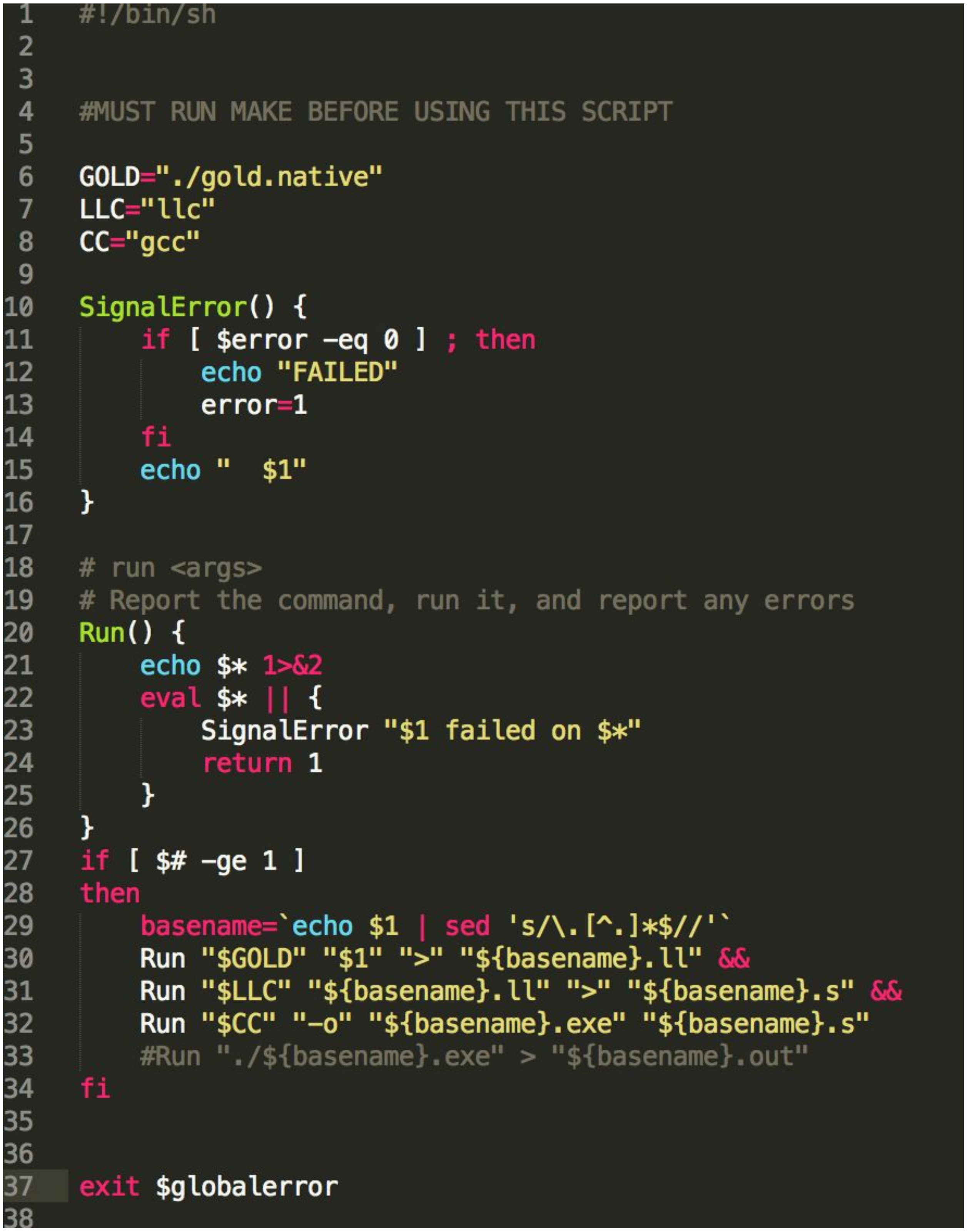
```bash
#!/bin/bash


# Shell function for testing GOLD compiler against test functions

if [ $# -eq 0 ]; then
        target=hello
fi

outputs=0
goldlog=goldlog.log


if [ "$target" == "--help" ]; then
        printf "Usage: ./run.sh <target>\n"
        echo "--list lists possible <target>'s"
        printf "--all runs all functions in test folder\n"
        exit
fi

# flag for listing values that can be passed as
# first arg 'target'
List_targets()
{
        ls test/*.gold \
                | cut -d '/' -f 2 \
                | cut -d '.' -f 1
}

Usage()
{
    echo "Usage: ./run.sh [-alho] [-g <target group>] [targets]"
    echo "Flags: -a for test all, -l for list targets, -h for usage, -o for keeping output files"
    echo "-g followed by a string to test all with that string in the name"
    echo "Any specifc targets at the end will be run as well"
}

Clean()
{
        make clean >> /dev/null
        cd test && ./clean.sh >> /dev/null
        printf "Cleaned!\n"
}
```



```
Build()
{
        make clean >> /dev/null
        make >> /dev/null
        printf "Built!\n"
}

Run()
{
        for func in $funcs
        do
                printf "####### $func #######\n"
                {
                    printf "###### $func #######\n" 1>&2
                    ./compile.sh test/"$func".gold 1>&2
                    test/"$func".exe > "$func".out
                    echo 1>&2
                } 2>> $goldlog

                if [ $(grep "SUCCESS" "$func".out) ]
        then
            printf "SUCCESS\n"
        else
            printf "FAILURE\n"
        fi
        done

        if [ $outputs -eq 0 ]
        then
            rm -f *.out
        fi

        printf "Finished testing!\n"
}


while getopts alhog: o
do      case "$o" in
        a) funcs="$(ls test \
                | grep .gold \
```



```
		h) Usage;;
		o) outputs=1;;

		?) printf >&2 "Usage: $0 [-a] [-l] [-g <target-group>]\n"

		esac
done

# Add single arguments to final test group
shift $(( $OPTIND - 1 ))
if [ "$funcs" = "" ]
then
		funcs="$*"
else
		while (( $# ))
		do
				funcs="$funcs\n$1"
				shift
		done
fi

#Build and compile/test targets
Build

Run

Clean
```



```sh
#!/bin/sh


#MUST RUN MAKE BEFORE USING THIS SCRIPT

GOLD="./gold.native"
LLC="llc"
CC="gcc"

SignalError() {
    if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
    fi
    echo "  $1"
}

# run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
        SignalError "$1 failed on $*"
        return 1
    }
}
if [ $# -ge 1 ]
then
    basename=`echo $1 | sed 's/\.[^.]*$//'`
    Run "$GOLD" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s"
    #Run "./${basename}.exe" > "${basename}.out"
fi


exit $globalerror
```

8 Conclusions

8.1 Member Reflections

8.1.1 Zeke

Heeding the warnings on CULPA we immediately set to brain storming possible language ideas, as well as functionalities that could be relevant, useful or interesting. Whereas we had a very strong start, unfortunately during the middle part of the semester productivity almost completely dropped. This led to a very very stressful end of the semester, and the biggest takeaway I got from this was that communication is incredibly important. Although I was part of the low productive team members in the middle of the semester, I still ended putting forth a monstrous effort in the end to not only catch up but contribute, which came at a huge cost to other classes and my own sleep schedule.

I think above all it'd be worthwhile to look for reliable and hard working team members, since a project of this scope and scale requires a lot of consistent work. Also, as CULPA says, don't wait for the class to get to the MicroC compiler - instead start learning how to develop a compiler for your language as fast as possible.

Furthermore, one other thing that led to a lack of productivity in the interim between the hello world demo and the actual demo was the lack of a concrete milestone in the class. This wasn't a class fault - rather we didn't create own for ourselves, and that led to us lagging behind as we prioritized other classes. So make sure you consistently come up with concrete milestones.

8.1.2 Aidan

When sitting in Edwards first lecture, I saw his warning underscoring the importance of proper group selection. I understood the nature of many of us on the team, but figured all would come through in the end to make significant progress. I worked on and off through the semester, making a large amount of the first big steps, so that when crunch time came around, we would not have to worry about everything, only the last bits to the finish line. Crunch time came, and the warnings of Edwards became prophecy.

Admittedly, the work involved with this project has a high learning curve. It takes time and effort to really dig into the material and grasp an understanding of what is going on at every level. The architecture laid out in the sample MicroC does very well at keeping you confined into each section's existential purpose: if you try to debug in Codegen for example, things can get messy extremely quickly; if you try to handle every concept as a subtype of a subtype, you are going to have a bad time; if you are doing things right and with enough care and thought, you will still hit a wall wondering what went wrong, take two steps back, and realize an easier way that elegantly accomplishes what you want.

If you do not put time into this material, none of that can happen. I'm glad that by the end of this project I was not the only one accomplishing real progress.

Realize that people look at the git commits. To commit a single line addition of a comment, change a variable name, add a single ill thought-out test script, or change a small detail that may or may not end up breaking things, your teammates will eventually see this and think poorly of that action, taken in the name of sheer quantity of commits.

This being said, with the amount of effort I feel that I put into this class, I feel that my skills as a developer have increased. Seeing this process in a "hands-on" manner has given me a deeper appreciation for what goes on underneath the hood, has sparked curiosity into new branches of this field, and has been pretty fun when things slowly started to come together in each part.

8.2 Advice to Future Groups

If you for a second -- even a second -- think you may be in one of "those groups" when Edwards first lectures on the perils of poor group dynamic, then you are in one.

Also, go hands-on with the MicroC code and then learn Llvm IR a bit hands on. If you understand these two specific things, everything will start clicking earlier.

9 Future Work

Type Structs, Elif statements, Passing arrays to functions

10 Code Listing

10.1 Compiler Code

```
(* Ocamllex scanner for GOLD *)
(* Author: Aidan Rivera *)
(* Contributor: Ezekiel Reyna *)

{ open Parser }
```

```
rule token = parse
        [' ' '\t' '\r' '\n']  { token lexbuf }
      | "//"            { comment lexbuf }
      | '"'       { str (Buffer.create 16) lexbuf }
      | ';'       { SEMI }
      | '{'       { LBRACE }
      | '}'       { RBRACE }
      | '('       { LPAREN }
      | ')'       { RPAREN }
      | '['       { LBRACKET }
      | ']'       { RBRACKET }
      | ','       { COMMA }
      | '+'       { PLUS }
      | '-'       { MINUS }
      | '*'       { TIMES }
      | '/'       { DIVIDE }
      | '%'       { MOD }
      | '='       { ASSIGN }
      | "=="      { EQ }
      | "!="      { NEQ }
      | "<"       { LT }
      | "<="      { LEQ }
      | ">"       { GT }
      | ">="      { GEQ }
      | "&&"      { AND }
      | "||"      { OR }
      | "!"       { NOT }
      | "func"    { FUNCTION }
      | "type"    { TYPE }
      | "struct"  { STRUCT }
      | "if"      { IF }
      | "else"    { ELSE }
      | "for"     { FOR }
      | "int"     { INT }
      | "float"   { FLOAT }
      | "string"  { STRING }
      | "bool"    { BOOL }
      | "void"    { VOID }
      | "true"    { TRUE }
      | "false"   { FALSE }
      | "return"  { RETURN }
      | "~"       { DEREF }           (* changed, DEREF is equivalent to * 
in C *)
      | "@"       { REF }             (* changed, REF is equivalent to & in 
C *)
      (* need to have more pointer types... or not? *)
      | ['0'-'9']+ as word { INTLIT(int_of_string word) }
      | ['0'-'9']+ ['.'] ['0'-'9']+ as word { FLOATLIT(float_of_string 
word) }
      | ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '_' '0'-'9']* as word { 
ID(word) }
      | eof { EOF }
      | _ as char { raise (Failure("Illegal character " ^ Char.escaped 
char)) }


and comment = parse
        '\n'      { token lexbuf }
      | _   { comment lexbuf }

and str strbuf = parse
        '"'             { STRINGLIT( Buffer.contents strbuf) }
      (* escape characters *)
      | '\\' '"'  { Buffer.add_char strbuf '"'; str strbuf lexbuf }
      | '\\' '\\' { Buffer.add_char strbuf '\\'; str strbuf lexbuf } 
(*Should this be two \\s?*)
      | '\\' '/'  { Buffer.add_char strbuf '/'; str strbuf lexbuf }
      | '\\' 'n'  { Buffer.add_char strbuf '\n'; str strbuf lexbuf }
      | '\\' 't'  { Buffer.add_char strbuf '\t'; str strbuf lexbuf }
      | '\\' 'r'  { Buffer.add_char strbuf '\r'; str strbuf lexbuf } (* can 
add other escape chars later *)
      | [^ '\\' '"']+   { Buffer.add_string strbuf (Lexing.lexeme lexbuf); 
str strbuf lexbuf }
      | eof             { raise (Failure("Unterminated String")) }
      | _         { raise (Failure("Problem with string")) }
```

```
/*
parser.mly
Author: Aidan Rivera
Contributor: Ezekiel Reyna
*/
```

```
%{
      open Ast
%}


%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR FUNCTION
%token TYPE STRUCT INT STRING BOOL VOID FLOAT
%token DEREF REF
%token <int> INTLIT
%token <float> FLOATLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right DEREF REF
%right NOT NEG

%start program
%type <Ast.program> program

%%
program:
      decls EOF { $1 }

decls:
        /* Base */      { { globals=[]; functions=[]; tstructs=[] } }
      | decls vdecl     { { globals=($2 :: $1.globals);
functions=$1.functions; tstructs=$1.tstructs } }
      | decls fdecl     { { globals=$1.globals; functions=($2 ::
$1.functions); tstructs=$1.tstructs } }
```

```
      | decls tdecl     { { globals=$1.globals; functions=$1.functions;
tstructs=($2 :: $1.tstructs) } }

fdecl:
      FUNCTION ID LPAREN formals_opt RPAREN typ LBRACE stmt_list RBRACE {
            { typ = $6;
              fname = $2;
              var_args = false;
              formals = $4;
              body = List.rev $8
            }
      }

tdecl: /* maybe change to typ inst. of TYPE */
      TYPE ID STRUCT LBRACE members RBRACE {
            { tname = $2;
              members = $5
            }
      }

members:
        { [] }
      | mem_list { List.rev $1 }

mem_list:
        /*Base*/  { [] }
      | mem_list ID typ SEMI { ($2, $3) :: $1 }

formals_opt:
        { [] }
      | formal_list { List.rev $1 }

formal_list:
        typ ID    { [($1, $2)] }
      | formal_list COMMA typ ID { ($3, $4) :: $1 }

typ:
      | FLOAT           { Float }
      | typ LBRACKET INTLIT RBRACKET      { Array($1, Some $3) }
      | typ LBRACKET RBRACKET      { Array($1, None) }
      | INT       { Int }
      | STRING    { String }
```

```
       | BOOL            { Bool }
       | VOID            { Void }
       | typ DEREF       { Pointer($1) }

vdecl:
      typ ID SEMI { ($1, $2) }

vdecl_stmt:
       | typ ID SEMI { VDecl($1, $2) }
       | typ ID ASSIGN expr SEMI { VDeclAss($1, $2, $4) }
       | typ ID ASSIGN array_lit SEMI { VDeclAss($1, $2, $4) }

stmt_list:
        /*Base*/  { [] }
       | stmt_list stmt { $2 :: $1 }

stmt:
        expr SEMI              { Expr $1 }
       | vdecl_stmt                  { $1 }
       | RETURN SEMI                 { Return Noexpr }
       | RETURN expr SEMI            { Return $2 }
       | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
       | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
       | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
            { For($3, $5, $7, $9) }
       | LBRACE stmt_list RBRACE     { Block(List.rev $2) }

expr_opt:
        /* nothing */ { Noexpr }
       | expr          { $1 }

expr:
        INTLIT                 { IntLit($1) }
       | FLOATLIT              { FloatLit($1) }
       | STRINGLIT                   { StringLit($1) }
       | TRUE                        { BoolLit(true) }
       | FALSE                       { BoolLit(false) }
       | ID                    { Variable($1) }
       | expr PLUS expr        { Binop($1, Add,  $3) }
       | expr MINUS      expr        { Binop($1, Sub,  $3) }
       | expr TIMES      expr        { Binop($1, Mult, $3) }
       | expr DIVIDE     expr        { Binop($1, Div,  $3) }
       | expr MOD  expr        { Binop($1, Mod,  $3) }
       | expr EQ   expr        { Binop($1, Equal,      $3) }
       | expr NEQ  expr        { Binop($1, Neq,  $3) }
       | expr LT   expr        { Binop($1, Less, $3) }
       | expr LEQ  expr        { Binop($1, Leq,  $3) }
       | expr GT   expr        { Binop($1, Greater,    $3) }
       | expr GEQ  expr        { Binop($1, Geq,  $3) }
       | expr AND  expr        { Binop($1, And,  $3) }
       | expr OR   expr        { Binop($1, Or,         $3) }
       | MINUS expr %prec NEG        { Unop(Neg, $2) }
       | NOT expr              { Unop(Not, $2) }
       | ID ASSIGN expr              { Assign($1, $3) }
       | ID ASSIGN       array_lit   { Assign($1, $3) }
       | ID LPAREN actuals_opt RPAREN      { Call($1, $3) }
       | LPAREN expr RPAREN          { $2 }
       | ID LBRACKET expr RBRACKET   { Access($1, $3) }
       | lvalue ASSIGN expr          { RefChange($1, $3) }
       | DEREF ID              { Deref($2, None) }
       | REF ID                { Ref($2)   }

lvalue:
       | ID LBRACKET expr RBRACKET   { Deref($1, Some $3) }
       | DEREF ID              { Deref($2, None)  }
       | REF ID                { Ref($2)   }


array_lit:
       | LBRACE actuals_opt RBRACE   { ArrayLit($2) }

actuals_opt:
            { [] }
       | actuals_list    { List.rev $1 }

actuals_list:
        expr      { [$1] }
       | actuals_list COMMA expr { $3 :: $1 }
```

```
(* ast.ml *)
(* Author: Aidan Rivera *)
(* Contributor: Ezekiel Reyna *)
```

```
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater
|
            Geq | And | Or

type uop = Neg | Not

type typ =
      | Int | String | Bool | Void | Float
      | Array of typ * int option
      | Pointer of typ

type bind = typ * string

type member = string * typ

type expr =
      | IntLit of int          | Noexpr
      | StringLit of string    | Call of string * expr list
      | BoolLit of bool | Assign of string * expr
      | Variable of string     | Binop of expr * op * expr
      | Unop of uop * expr     | FloatLit of float
      | ArrayLit of expr list | Access of string * expr
      | RefChange of expr * expr
      | Deref of string * expr option
      | Ref of string

type stmt =
      | VDeclAss of typ * string * expr
      | VDecl of typ * string
      | Block of stmt list
      | Expr of expr
      | Return of expr
      | If of expr * stmt * stmt
      | For of expr * expr * expr * stmt
      | While of expr * stmt

type func_decl = {
      typ :       typ;
      fname :           string;
      var_args :  bool;
      formals :   bind list;
      body :            stmt list;
}

type tstruct = {
      tname :           string;
      members :   member list;

}

type program = {
      globals :   bind list;
      functions : func_decl list;
      tstructs :  tstruct list;(* Change later *)
}

let string_of_op = function
      | Add ->          "+"
      | Sub ->          "-"
      | Mult ->         "*"
      | Div ->          "/"
      | Mod ->          "%"
      | Equal ->        "=="
      | Neq ->          "!="
      | Less ->         "<"
      | Leq ->          "<="
      | Greater ->        ">"
      | Geq ->          ">="
      | And ->          "&&"
      | Or ->           "||"

let string_of_uop = function
      | Neg -> "-"
      | Not -> "!"

let rec string_of_typ = function
      | Int ->    "int"
      | String ->       "string"
      | Bool ->   "bool"
      | Void ->   "void"
      | Float -> "float"
      | Pointer t ->    string_of_typ t ^ " ~"
      | Array (t, n) -> match n with
```

```
            | Some i -> string_of_typ t ^ "[" ^ string_of_int i ^  "]"
            | None -> string_of_typ t ^ "[]"

let rec string_of_expr = function
      | IntLit i ->           string_of_int i
      | FloatLit f ->         string_of_float f
      | StringLit s ->  "\"" ^ s ^ "\""
      | BoolLit(true) ->      "true"
      | BoolLit(false) ->     "false"
      | Variable s ->         s
      | Noexpr ->       "void"
      | Binop (e1, o, e2) ->  string_of_expr e1 ^ " " ^ string_of_op o ^ "
" ^ string_of_expr e2
      | Unop(o, e) ->   string_of_uop o ^ string_of_expr e
      | Assign (s, e) ->      s ^ " = " ^ string_of_expr e
      | Call (s, el) -> s ^ "(" ^ (List.fold_left (fun b a -> b ^ " " ^
string_of_expr a ^ " ") "" el) ^ ")"
      | ArrayLit el ->  "{" ^ (List.fold_left (fun b a -> b ^ " " ^
string_of_expr a ^ ", ") "" el) ^ "}"
      | Access (s, e) ->      s ^ "[" ^ string_of_expr e ^ "]"
      | RefChange (e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
      | Ref s ->        "@" ^ s
      | Deref (s, e) -> match e with
            | Some e' ->      s ^ "[" ^ string_of_expr e' ^ "]"
            | None ->   "~" ^ s

let rec string_of_stmt = function
      | VDecl(t, id) ->       string_of_typ t ^ " " ^ id ^ ";\n"
      | VDeclAss(t, id, e) -> string_of_typ t ^ " " ^ id ^ " = " ^
string_of_expr e ^ ";\n"
      | Block(stmts) ->       "{\n" ^ String.concat "" (List.map
string_of_stmt stmts) ^ "}\n"
      | Expr(expr) ->   string_of_expr expr ^ ";\n"
      | Return(expr) ->       "return " ^ string_of_expr expr ^ ";\n"
      | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
      | If(e, s1, s2) ->      "if (" ^ string_of_expr e ^ ")\n" ^
            string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
      | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1  ^ " ; " ^
string_of_expr e2 ^ " ; " ^
            string_of_expr e3  ^ ") " ^ string_of_stmt s
      | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt
```

```
s

let string_of_fdecl fdecl =
      string_of_typ fdecl.typ ^ " " ^ fdecl.fname ^ "(" ^
      String.concat ", " (List.map snd fdecl.formals) ^ ")\n{\n" ^
      String.concat "" (List.map string_of_stmt fdecl.body) ^ "}\n"

let string_of_program program =
      String.concat "" (List.map (fun (t,id) -> string_of_stmt (VDecl
(t,id))) program.globals) ^ "\n" ^
      String.concat "\n" (List.map string_of_fdecl program.functions) (*
Add tstructs later *)
```

```
(* semant.ml *)
(* Author: Aidan Rivera *)
(* Contributor: Ezekiel Reyna *)

open Ast

module StringMap = Map.Make(String)

(* Returns void if true, else throws an exception *)
let check program =

      (* HELPER FUNCTIONS *)
      let report_duplicate exceptf list =
            let rec helper = function
                    n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf
n1 ))
                  | _ :: t -> helper t
                  | [] -> () in
            helper (List.sort compare list) in
      let check_duplicate_definition exceptf n vars =
            if StringMap.mem n vars then raise (Failure (exceptf n)) else
() in
      let check_not_void exceptf = function
              (Void, n) -> raise (Failure (exceptf n))
            | _ -> () in
      let check_member_not_void exceptf = function
              (n, Void) -> raise (Failure (exceptf n))
            | _ -> () in
      let check_assign lvaluet rvaluet err = (*lvaluet in*)
            match lvaluet with
            | Array(lt, None) ->
                  (match rvaluet with
                  | Array(rt, _) -> if lt == rt then lvaluet else raise err
                  | _ -> raise err)
            | _ -> if lvaluet = rvaluet then lvaluet else raise err in


      (* CHECKING GLOBALS *)
      List.iter (check_not_void (fun n -> "illegal void global " ^ n))
program.globals;
      report_duplicate (fun n -> "duplicate global " ^ n) (List.map (fun fd
-> fd.fname) program.functions);
      List.iter (fun t -> List.iter (check_member_not_void (fun n ->
"illegal void tstruct member " ^ n)) t.members) program.tstructs;


      (* CHECKING FUNCTIONS *)
      let built_in_functions = [ "print" ; "println" ; "sprint" ; "input" ;
"atoi" ] in
      List.iter (fun name ->
            if List.mem name (List.map (fun fd -> fd.fname)
program.functions)
            then raise (Failure ("function " ^ name ^ " may not be
defined")) else ();
            report_duplicate (fun n -> "duplicate function " ^ n) (List.map
(fun fd -> fd.fname) program.functions)
      ) built_in_functions;

      (* FUNCTION DECL FOR NAMED FUNCTIONS *)
      let built_in_decls = List.fold_left (fun m (n, t) -> StringMap.add n
t m) StringMap.empty
            [("print", { typ=Void; fname="print"; var_args=false; formals =
[(String, "x")]; body=[] });
            ("println",{ typ=Void; fname="println"; var_args=false; formals
= [(String, "x")]; body=[] });
            ("sprint", { typ=String; fname="sprint"; var_args=true; formals
= [(String, "x")]; body=[] });
            ("input",  { typ=String; fname="input"; var_args=false; formals
= []; body=[] });
            ("atoi", { typ=Int; fname="atoi"; var_args=false; formals =
[(String, "x")]; body=[] });
            ("rand", { typ=Int; fname="rand"; var_args=false; formals = [];
body=[] });
            ("srand",{ typ=Void; fname="rand"; var_args=false; formals =
[]; body=[] })]
      in

      let function_decls =
            List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
built_in_decls program.functions in
      let function_decl s = try StringMap.find s function_decls
            with Not_found -> raise (Failure ("unrecognized function " ^
s)) in


      (* ENSURE "MAIN" IS DEFINED *)
      let _ = function_decl "main" in

      let check_function func =
            List.iter (check_not_void (fun n ->
                  "illegal void formal " ^ n ^ " in " ^ func.fname))
func.formals;
            report_duplicate (fun n ->
                  "duplicate formal " ^ n ^ " in " ^ func.fname) (List.map
snd func.formals);

            (* VARIABLE SYMBOL TABLE *)
            let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t
m)
                  StringMap.empty (program.globals @ func.formals) in
            let type_of_identifier vars s =
                  try StringMap.find s vars
                  with Not_found -> raise (Failure ("undeclared identifier
" ^ s)) in


            (* RETURN TYPE OF EXPRESSION OR THROW EXCEPTION *)
            let rec expr locals = function
                  | IntLit _ ->          Int
                  | FloatLit _ ->        Float
                  | StringLit _ ->  String
                  | BoolLit _ ->         Bool
                  | Noexpr ->            Void
                  | Variable s ->        type_of_identifier locals s
                  | Binop(e1, op, e2) as e -> let t1 = expr locals e1 and
t2 = expr locals e2 in
                        (match op with
                        | Add | Sub | Mult | Div | Mod when t1 = Int && t2
= Int -> Int
                        | Add | Sub | Mult | Div | Mod when t1 = Float &&
t2 = Float -> Float
                        | Equal | Neq when t1 = t2 -> Bool
                        | Less | Leq | Greater | Geq when t1 = Int && t2 =
Int -> Bool
                        | Less | Leq | Greater | Geq when t1 = Float && t2
= Float -> Bool
                        | And | Or when t1 = Bool && t2 = Bool -> Bool
                        | _ -> raise (Failure ("illegal binary operator " ^
                              string_of_typ t1 ^ " " ^ string_of_op op ^ "
" ^
                              string_of_typ t2 ^ " in " ^ string_of_expr
e)))
                  | Unop(op, e) as ex -> let t = expr locals e in
                        (match op with
                        | Neg when t = Int -> Int
                        | Neg when t = Float -> Float
                        | Not when t = Bool -> Bool
                        | _ -> raise (Failure ("illegal unary operator " ^
string_of_uop op ^
                              string_of_typ t ^ " in " ^ string_of_expr
ex)))
                  | Assign(var, e) as ex -> let lt = type_of_identifier
locals var and rt = expr locals e in
                        check_assign lt rt (Failure ("illegal assignment "
^ string_of_typ lt ^
                                    " = " ^ string_of_typ rt ^ " in " ^
string_of_expr ex))
                  | Call(fname, actuals) as call -> let fd = function_decl
fname in
                        if fd.var_args = false then
                              (if List.length actuals != List.length
fd.formals then
                                    raise (Failure ("expecting " ^
string_of_int (List.length fd.formals)
```

```
                                          ^ " arguments in " ^
string_of_expr call))
                              else
                                    List.iter2 (fun (ft, _) e -> let et =
expr locals e in
                                          ignore (check_assign ft et
                                                (Failure ("illegal actual
argument found " ^
                                                string_of_typ et ^ "
expected " ^
                                                string_of_typ ft ^ " in " ^
string_of_expr e))))
                                          fd.formals actuals;
                                    fd.typ)
                        else
      (*                      List.iter2 (fun (ft, _) e -> let et = expr
locals e in
                                    ignore (check_assign ft et
                                          (Failure ("illegal actual argument
found " ^
                                          string_of_typ et ^ " expected " ^
                                          string_of_typ ft ^ " in " ^
string_of_expr e))))
      *)
                              fd.typ (*Change to check equality in all
formals, not actuals *)
                  | ArrayLit el ->
                        let size = List.length el in
(* add more *)
                        Array((expr locals (List.hd el)), Some size)

                  | Access (id, _) ->
                        (match type_of_identifier locals id with
                        | Array(t, _) -> t
                        | t -> t)
                  | Ref s -> Pointer(type_of_identifier locals s)
                  | Deref (s,e) ->
                        (match e with
                        | Some _ -> raise (Failure ("Access should be hit,
not deref in " ^ string_of_expr (Deref(s, e))))
                        | None ->
                              (match type_of_identifier locals s with
                              | Pointer t -> t
                              | _ -> raise(Failure ("Dereferencing non
pointer value in " ^ string_of_expr (Deref(s,e))))))
                  | RefChange (e1, e2) ->
                        let expr_string = string_of_expr (RefChange(e1,e2))
in
                        (match e1 with
                        | Ref s -> Pointer(type_of_identifier locals s)
                        | Deref (s,e) ->
                              (match e with
                              | Some _ ->
                                    (match type_of_identifier locals s with
                                    | Array(t, _) -> t
                                    | _ -> raise (Failure ("RefChange array
error in " ^
                                          expr_string)))
                              | None ->
                                    (match type_of_identifier locals s with
                                    | Pointer t -> t
                                    | _ -> raise (Failure ("Pointer deref
error in " ^
                                          expr_string))))
                        | _ -> raise (Failure ("Changing unavailable ref "
^ expr_string)))
                  in

            let check_bool_expr locals e = if expr locals e != Bool
                  then raise (Failure ("expected Boolean expression in " ^
string_of_expr e))
                  else () in


            (* VERIFY STATEMENT OR THROW EXCEPTION *)
            let rec stmt locals = function
                  | Block sl ->
                        let rec check_block block_locals = function
                        | [Return _ as s] -> stmt block_locals s
                        | Return _ :: _ -> raise (Failure "nothing may
follow a return")
                        | Block sl :: ss -> stmt block_locals (Block sl);
check_block block_locals ss
                        | s :: ss ->
```

```
                              (match s with
                              | VDecl (t, id) ->
                                    check_duplicate_definition (fun n ->
"variable already defined " ^ n ^ " in " ^ func.fname) id block_locals;
                                    let block_locals = StringMap.add id t
block_locals in
                                    check_block block_locals ss
                              | VDeclAss (t,id,e) ->
                                    check_duplicate_definition (fun n ->
"variable already defined " ^ n ^ " in " ^ func.fname) id block_locals;
                                    let block_locals = StringMap.add id t
block_locals in
                                    ignore (expr block_locals e);
check_block block_locals ss
                              | _ -> stmt block_locals s; check_block
block_locals ss)
                        | [] -> () in
                        check_block locals sl
                  | VDecl (_,_) -> () (* Placeholder, as this shouldn't be
hit *)
                  | VDeclAss(_,_,_) -> ()
                  | Expr e -> ignore (expr locals e)
                  | Return e ->
                        let t = expr locals e in
                        if t = func.typ then ()
                        else raise (Failure ("return gives " ^
string_of_typ t ^ " expected " ^
                              string_of_typ func.typ ^ " in " ^
string_of_expr e))
                  | If(p, b1, b2) -> check_bool_expr locals p; stmt locals
b1; stmt locals b2
                  | For(e1, e2, e3, st) -> ignore (expr locals e1);
check_bool_expr locals e2;
                        ignore (expr locals e3); stmt locals st
                  | While(_, _) -> () in (* For pattern matching warning *)

            stmt symbols (Block func.body) (* Body of check function *) in
      List.iter check_function program.functions (* Body of check *)
```

```
(* codegen.ml *)
(* Author: Aidan Rivera *)
(* Contributor: Ezekiel Reyna *)

module L = Llvm
module A = Ast

module StringMap = Map.Make(String)

exception Foo of string

let translate program =
      let context = L.global_context() in
      let the_module = L.create_module context "Gold"

      and i32_t   = L.i32_type      context           (* int *)
      and flt_t   = L.float_type context        (* 32 bit float *)
      and i8_t    = L.i8_type context           (* for print *)
      and i1_t    = L.i1_type context           (* bool *)
      and str_t   = L.pointer_type (L.i8_type context)
      and void_t  = L.void_type     context     (* void *)
      and array_t = L.array_type
      and pointer_t     = L.pointer_type in

      let rec ltype_of_typ = function (* Llvm type for Ast type *)
              A.Int ->  i32_t
             | A.Float -> flt_t
             | A.String ->     str_t
             | A.Bool -> i1_t
             | A.Void ->       void_t
             | A.Pointer t -> pointer_t (ltype_of_typ t)
             | A.Array(t,n) -> match n with
                   | Some i -> array_t (ltype_of_typ t) i
                   | None -> pointer_t (ltype_of_typ t)      in

      (* Initialize global variables *)
      let global_vars =
            let global_var m (t, n) =
                  let init = L.const_int (ltype_of_typ t) 0
                  in StringMap.add n (L.define_global n init the_module) m
in
            List.fold_left global_var StringMap.empty program.A.globals in
      (* Declare external function print *)
      let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t
|] in
      let print_func = (* For print and println *)
            L.declare_function "printf" printf_t the_module in(*If this
isnt prinf hello.gold doesnt compile*)
      let sprintf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t
; L.pointer_type i8_t |] in
      let sprint_func =
            L.declare_function "sprintf" sprintf_t the_module in
      let input_t = L.function_type (pointer_t i8_t) [| L.pointer_type i8_t
|] in
      let input_func =                          (* Uses gets, I know it's
dangerous *)
            L.declare_function "gets" input_t the_module in
      let atoi_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |]
in
      let atoi_func =
            L.declare_function "atoi" atoi_t the_module in
      let rand_t = L.var_arg_function_type i32_t [| |] in
      let rand_func =
            L.declare_function "rand" rand_t the_module in
      let time_t = L.var_arg_function_type i32_t [| L.pointer_type i32_t |]
in
      let time_func =
            L.declare_function "time" time_t the_module in
      let srand_t = L.var_arg_function_type i32_t [| i32_t |] in
      let srand_func =
            L.declare_function "srand" srand_t the_module in

      (* Safe stdin support *)
(*    let file_t = L.named_struct_type context "file" in
      let file_ptr_t = pointer_t file_t in
      let fdopen_t = L.var_arg_function_type i32_t [| i32_t ; pointer_t
i8_t |] in
      let fdopen_func = L.declare_function "fdopen" fdopen_t the_module in
      let mode_ptr = L.define_global "r" (L.const_array i8_t [| L.const_int
i8_t 0 ; L.const_int i8_t 0 |]) the_module in
*)


      (* Define each function *)
      let function_decls =
            let function_decl m fdecl =
                  let name = fdecl.A.fname
                  and formal_types = Array.of_list
                        (List.map (fun (t, _) -> ltype_of_typ t)
fdecl.A.formals)
                  in let ftype =
                        L.function_type (ltype_of_typ fdecl.A.typ)
formal_types in
                  StringMap.add name (L.define_function name ftype
the_module, fdecl) m in
            List.fold_left function_decl StringMap.empty
program.A.functions in

      (* Fill in function body *)
      let build_function_body fdecl =
            let (the_function, _) =
                  StringMap.find fdecl.A.fname function_decls in
            let builder =
                  L.builder_at_end context (L.entry_block the_function) in

(*          if fdecl.A.fname = "main" then
                  ignore(L.build_call fdopen_func [| L.const_int i32_t 0;
                        L.const_in_bounds_gep mode_ptr [| mode_ptr ;
L.const_int i32_t 0; L.const_int i32_t 0|] |] "stdin" builder);
*)

            let local_vars =
                  let add_formal m (t, n) p = L.set_value_name n p;
                        let local = L.build_alloca (ltype_of_typ t) n
builder in
                        ignore (L.build_store p local builder);
                        StringMap.add n local m in
                  List.fold_left2 add_formal StringMap.empty
fdecl.A.formals
                        (Array.to_list (L.params the_function)) in

            let lookup n vars = try StringMap.find n vars
                  with Not_found -> StringMap.find n global_vars in

            let rec expr builder local_vars = function
                    A.IntLit i -> L.const_int i32_t i
```

```
                   | A.FloatLit f -> L.const_float flt_t f
                   | A.StringLit s -> L.build_global_stringptr s "str"
builder
                   | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
                   | A.Noexpr -> L.const_int i32_t 0
                   | A.Variable s -> L.build_load (lookup s local_vars) s
builder
                   | A.Binop (e1, op, e2) ->
                       let e1' = expr builder local_vars e1
                       and e2' = expr builder local_vars e2 in
                       let int_bopr opr =
                             (match opr with
                               A.Add           -> L.build_add
                             | A.Sub           -> L.build_sub
                             | A.Mult    -> L.build_mul
                             | A.Div           -> L.build_sdiv
                             | A.Mod           -> L.build_srem
                             | A.And           -> L.build_and
                             | A.Or            -> L.build_or
                             | A.Equal   -> L.build_icmp L.Icmp.Eq
                             | A.Neq           -> L.build_icmp L.Icmp.Ne
                             | A.Less    -> L.build_icmp L.Icmp.Slt
                             | A.Leq           -> L.build_icmp L.Icmp.Sle
                             | A.Greater -> L.build_icmp L.Icmp.Sgt
                             | A.Geq           -> L.build_icmp L.Icmp.Sge
                             ) e1' e2' "tmp" builder in
                       let flt_bopr opr =
                             (match opr with
                               A.Add           -> L.build_fadd
                             | A.Sub           -> L.build_fsub
                             | A.Mult    -> L.build_fmul
                             | A.Div           -> L.build_fdiv
                             | A.Mod           -> L.build_frem
                             | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
                             | A.Neq           -> L.build_fcmp L.Fcmp.One
                             | A.Less    -> L.build_fcmp L.Fcmp.Olt
                             | A.Leq           -> L.build_fcmp L.Fcmp.Ole
                             | A.Greater -> L.build_fcmp L.Fcmp.Ogt
                             | A.Geq           -> L.build_fcmp L.Fcmp.Oge
                             | _         -> raise (Foo "Invalid Float
Operator")
                             ) e1' e2' "tmp" builder in
```

```
        (*
                    let othr_bopr opr =
                        (match opr with
                          A.Add           -> L.build_fadd
                        | A.Sub           -> L.build_fsub
                        | A.Mult    -> L.build_fmul
                        | A.Div           -> L.build_fdiv
                        | A.Mod           -> L.build_frem
                        | A.And           -> L.build_and
                        | A.Or            -> L.build_or
                        | A.Equal   -> L.build_icmp L.Icmp.Eq
                        | A.Neq           -> L.build_icmp L.Icmp.Ne
                        | A.Less    -> L.build_icmp L.Icmp.Slt
                        | A.Leq           -> L.build_icmp L.Icmp.Sle
                        | A.Greater -> L.build_icmp L.Icmp.Sgt
                        | A.Geq           -> L.build_icmp L.Icmp.Sge
                        ) e1' e2' "tmp" builder in
        *)
                        if (L.type_of e1' = flt_t && L.type_of e2' = flt_t)
then flt_bopr op
                        else int_bopr op
                  | A.Unop(op, e) ->
                        let e' = expr builder local_vars e in
                        (match op with
                        | A.Neg           -> L.build_neg e' "tmp" builder
                        | A.Not           -> L.build_not e' "tmp" builder)
                  | A.Assign (s, e) -> let e' = expr builder local_vars e
in
                        ignore (L.build_store e' (lookup s local_vars)
builder); e'

                  | A.Deref (s, e) ->
                        (match e with
                        | Some e1 -> let e2 = expr builder local_vars e1 in
                              let e_ref = L.build_gep (lookup s local_vars)
[| (L.const_int i32_t 0) ; e2 |] s builder in
                              L.build_load e_ref s builder
                        | None -> L.build_load (L.build_load (lookup s
local_vars) s builder) s builder)
                  | A.Ref s -> lookup s local_vars
                  | A.RefChange (e1, e2) -> let e2' = expr builder
local_vars e2 in
```

```
                        (match e1 with
                        | A.Deref(s, eo) -> (match eo with
                              | Some e' -> let e3 = expr builder local_vars
e' in
                                    let loc = L.build_gep (lookup s
local_vars) [| L.const_int i32_t 0 ; e3 |] s builder  in
                                    L.build_store e2' loc builder
                              | None ->
                                    L.build_store e2' (L.build_load (lookup
s local_vars) s builder) builder)
                        | A.Ref s -> let loc = L.build_gep (lookup s
local_vars) [| L.const_int i32_t 0 |] s builder in
                              L.build_store e2' loc builder
                        | _ -> raise (Foo "Changing the reference of
unchangeable expr"))
                  | A.Call ("print", [e]) ->
                        let format_str = L.build_global_stringptr "%s"
"fmt" builder in
                        L.build_call print_func [| format_str ; (expr
builder local_vars e) |] "printf" builder
                  | A.Call ("println", [e]) ->
                        let format_str = L.build_global_stringptr "%s\n"
"fmt" builder in
                        L.build_call print_func [| format_str ; (expr
builder local_vars e) |] "printf" builder
                  | A.Call ("sprint", e) ->
                        (match e with
                        | hd :: tl ->
                              let format_str =
                                    match hd with
                                    | A.StringLit s ->
L.build_global_stringptr s "fmt" builder
                                    | _ -> L.build_global_stringptr ""
"fmt" builder (*Shouldnt be hit, how do handle? *) in
                              let buf = L.build_alloca (L.array_type i8_t
4096) "buf" builder in
                              let buf_ref = L.build_in_bounds_gep buf [|
(L.const_int i32_t 0); (L.const_int i8_t 0) |] "buf" builder in
                              let arg_arr = Array.append [| buf_ref;
format_str |] (Array.of_list (List.map (fun arg -> expr builder local_vars
arg) tl)) in
                              ignore (L.build_call sprint_func arg_arr
"sprintf" builder); buf_ref
                        | _ ->
                              L.build_global_stringptr "" "fmt" builder)
(*Shouldnt be hit, how do handle? *)
                  | A.Call ("input", []) ->
                        let buf = L.build_alloca (L.array_type i8_t 4096)
"buf" builder in
                        let buf_ref = L.build_in_bounds_gep buf [|
(L.const_int i32_t 0) ; (L.const_int i8_t 0) |] "buf" builder in
                        let raw_in = L.build_call input_func [| buf_ref |]
"gets" builder in
                        raw_in
                  | A.Call ("atoi", [e]) -> L.build_call atoi_func [| expr
builder local_vars e |] "atoi" builder
                  | A.Call ("srand", []) ->
                        let t_holder = L.build_alloca i32_t "t_holder"
builder in
                        ignore(L.build_call time_func [| t_holder |] "time"
builder);
                        L.build_call srand_func [| (L.build_load t_holder
"tm" builder) |] "srand" builder
                  | A.Call ("rand", []) ->
                        L.build_call rand_func [| |] "rand" builder
                  | A.Call (f, act) ->
                        let (fdef, fdecl) = StringMap.find f function_decls
in
                        let actuals = List.rev (List.map (expr builder
local_vars) (List.rev act)) in
                        let result = (match fdecl.A.typ with A.Void -> ""
                              | _ -> f ^ "_result") in
                        L.build_call fdef (Array.of_list actuals) result
builder
                  | A.ArrayLit el ->
                        let elements = List.rev (List.map (expr builder
local_vars) (List.rev el)) in
                        let t = L.type_of (List.hd elements) in
                        L.const_array t (Array.of_list elements)
                  | A.Access (s, e) -> let e' = expr builder local_vars e
in
                        let e_ref = L.build_gep (lookup s local_vars) [|
(L.const_int i32_t 0) ; e' |] s builder in
                        L.build_load e_ref s builder
                  in

            let add_terminal builder f =
                  match L.block_terminator (L.insertion_block builder) with
                          Some _ -> ()
                        | None -> ignore (f builder) in

            let rec stmt builder local_vars = function
                  | A.Block sl -> List.fold_left (fun (b, lv) s -> stmt b
lv s) (builder, local_vars) sl
                  | A.VDecl (t, n) ->
                        let local_var = L.build_alloca (ltype_of_typ t) n
builder in
                        let local_vars = StringMap.add n local_var
local_vars in
                        (builder, local_vars)
                  | A.VDeclAss (t, n, e) ->
                        let local_var = L.build_alloca (ltype_of_typ t) n
builder in
                        let local_vars = StringMap.add n local_var
local_vars in
                        ignore (expr builder local_vars (A.Assign (n, e)));
(builder, local_vars)
                  | A.Expr e -> ignore (expr builder local_vars e);
(builder, local_vars)
                  | A.Return e -> ignore (match fdecl.A.typ with
                        A.Void -> L.build_ret_void builder
                              | _ -> L.build_ret (expr builder local_vars
e) builder); (builder, local_vars)
                  | A.If (predicate, then_stmt, else_stmt) ->
                        let bool_val = expr builder local_vars predicate in
                        let merge_bb = L.append_block context "merge"
the_function in

                        let then_bb = L.append_block context "then"
the_function in
                        add_terminal (fst (stmt (L.builder_at_end context
then_bb) local_vars then_stmt))
                        (L.build_br merge_bb);

                        let else_bb = L.append_block context "else"
the_function in
```

```
                        add_terminal (fst (stmt (L.builder_at_end context
else_bb) local_vars else_stmt))
                        (L.build_br merge_bb);

                        ignore (L.build_cond_br bool_val then_bb else_bb
builder);
                        (L.builder_at_end context merge_bb, local_vars)

                  | A.While (predicate, body) ->
                        let pred_bb = L.append_block context
"for_condition" the_function in
                        ignore (L.build_br pred_bb builder);

                        let body_bb = L.append_block context "for_body"
the_function in
                        add_terminal (fst (stmt (L.builder_at_end context
body_bb) local_vars body))
                        (L.build_br pred_bb);

                        let pred_builder = L.builder_at_end context pred_bb
in
                        let bool_val = expr pred_builder local_vars
predicate in

                        let merge_bb = L.append_block context "merge"
the_function in
                        ignore (L.build_cond_br bool_val body_bb merge_bb
pred_builder);
                        (L.builder_at_end context merge_bb, local_vars)

                  | A.For (e1, e2, e3, body) -> stmt builder local_vars
                        ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ;
A.Expr e3]) ] ) in

            (* Build code for each statement in the function *)
            let (builder, _) = stmt builder local_vars (A.Block
fdecl.A.body) in

            (* Add a return if last block falls off end *)
            add_terminal builder (match fdecl.A.typ with
                    A.Void -> L.build_ret_void
                  | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
            in

      List.iter build_function_body program.A.functions;
      the_module
```

```
(* gold.ml *)
(* Author: Aidan Rivera *)
(* Top-level of the Gold compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

module StringMap = Map.Make(String)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./gold.native [-a|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
usage_msg;
  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.check ast;
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

```
# Makefile
# Author: Aidan Rivera
#
```

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : all
all : gold.native

.PHONY : gold.native
gold.native :
      ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4
gold.native

# "make clean" removes all generated files

.PHONY : clean
clean :
      ocamlbuild -clean
      cd test/; ./clean.sh
      rm -rf testall.log *.diff gold scanner.ml parser.ml parser.mli
      rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx gold.cmx

gold : $(OBJS)
      ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis
$(OBJS) -o gold

scanner.ml : scanner.mll
      ocamllex scanner.mll

parser.ml parser.mli : parser.mly
      ocamlyacc parser.mly

%.cmo : %.ml
      ocamlc -c $<

%.cmi : %.mli
      ocamlc -c $<
```

```
%.cmx : %.ml
      ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and
parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
gold.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
gold.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo


TARFILES = ast.ml codegen.ml Makefile gold.ml parser.mly \
        scanner.mll semant.ml \
      $(TESTFILES:%=tests/%)

gold-llvm.tar.gz : $(TARFILES)
      cd .. && tar czf gold-llvm/gold-llvm.tar.gz \
            $(TARFILES:%=gold-llvm/%)
```

```
#!/bin/bash
# run.sh
# Author: Ezekiel Reyna

# Shell function for testing GOLD compiler against test functions

if [ $# -eq 0 ]; then
        target=hello
fi

if [ $# -eq 1 ]; then
        target=$1
```

```
fi

if [ $# -gt 1 ]; then
        printf "Usage: ./run.sh <target>\n"
        printf "<target> doesn't have .gold suffix.\n"
        printf "use --help for more info\n"
        exit
fi

if [ "$target" == "--help" ]; then
        printf "Usage: ./run.sh <target>\n"
        echo "--list lists possible <target>'s"
        printf "--all runs all functions in test folder\n"
        exit
fi

# flag for listing values that can be passed as
# first arg 'target'
if [ "$target" == "--list" ]; then
        ls test/*.gold \
                | cut -d '/' -f 2 \
                | cut -d '.' -f 1
        exit
fi

# test all functions.
if [ "$target" == "--all" ]; then
        # implement doofus.
        funcs="$(ls test \
                | grep .gold \
                | cut -d "." -f 1)"

        make clean >> /dev/null
        make >> /dev/null

        for func in $funcs
        do
                printf "######## $func ########\n"
                ./compile.sh test/"$func".gold >> /dev/null 2&>1
                test/"$func".exe
        done
        make clean >> /dev/null
        cd test && ./clean.sh >> /dev/null
        exit
fi

# build
make clean >> /dev/null
make >> /dev/null

# compile
./compile.sh test/"$target".gold

# execute
test/"$target".exe

# clean
make clean >> /dev/null
cd test && ./clean.sh >> /dev/null
```

10.2 Demo File

```
func print_num(int num) void {
      print(sprint("%d", num));
}

func main () int {
      int dim = 8;
      int board_size = dim * dim;
      int k;
      //background
      int bkg = 1;
      //foreground
      int frg = 8;
      //Player
      int plr = 4;
```

```
    int [64] board;

    for (k = 0; k < dim * dim; k = k + 1) {
          board[k] = bkg;
    }

    srand();
    int p_spot = dim * (dim-1) + 3; //3 is arbitrary
    board[p_spot] = plr;

    int i;
    int j;
    int arr_len = dim * dim;
    bool collide = false;
    bool victory = false;
    for (k = 0; collide == false && victory == false; k = k +1) {


          //Shifting blocks down
          for(i = dim-1; i > 0; i = i - 1) {
                for (j=0; j < dim; j = j + 1) {
                      int spot_above = (i-1)*dim + j;
                      int spot = i * dim + j;
                      if (board[spot_above] == frg) {
                            if (board[spot] == plr) {
                                  collide = true;
                            }
                            board[spot] = frg;
                      } else {
                            if (board[spot] != plr) {
                                  board[spot] = bkg;
                            }
                      }
                }
          }

          int change_spot = rand() % dim;

          //Changing top row
          for(i = 0; i < dim; i = i+1) {
                if (i == change_spot) {
                      board[i] = frg;
                } else {
                        board[i] = bkg;
                }
        }


        //PRINTING BOARD
        println("CLIMB THE STAIRS TO YOUR LEFT!");
        for(i = 0; i < dim && collide == false; i = i + 1) {
                print("| ");
                for(j = 0; j < dim; j = j + 1) {
                        int spot = (i * dim) + j;
                        print_num(board[spot]);
                        print(" ");
                }
                println("|");
        }


        //GETTING PLAYER INPUT
        if (collide == false) {
                print("Move (2=l,3=r):  ");
                int move = atoi(input());
                println("");

                board[p_spot] = bkg;
                if(move == 2) {
                        p_spot = p_spot - 1;
                }
                if(move == 3 && p_spot != (dim * dim - 1)) {
                        p_spot = p_spot + 1;
                }
                if (board[p_spot] == frg) {
                        collide = true;
                } else {
                        board[p_spot] = plr;
                        if (p_spot == 0) {
                                victory = true;
                        }
                }
        }
}
```

```
      if (victory == true) {
            println(sprint("CONGRATULATIONS  -  YOU HAVE WON IN  %i 
MOVES!", k));
      } else {
            string c_spot = sprint("Collision in: (left-to-right, 
top-to-bottom) (%i, %i)", (p_spot % dim + 1), (p_spot / dim + 1));
            println(c_spot);
            println(sprint("GAME OVER  -  %d ROWS TO GO!", (p_spot / dim) + 
1));
      }
      return 0;
}
```

10.3 Test Files

```
//alphanumeric-names.gold
func main() int {
    //int a2;
    //a2 = 1;
    int a2;
    a2 = 1;
    if (a2 == 1) {
        println("SUCCESS");
    }
    return 0;
}
```

```
//array-element-change.gold
func main () int {
      int[6] a = {0,1,2,3,4,5};

      int i;
      for (i = 0; i < 6; i = i + 1) {
            a[i] = a[i] % 2;
      }

      for (i = 0; i < 6; i = i + 1) {
            println(sprint("%d", a[i]));
      }
    println("SUCCESS");

      return 0;
}
```

```
//array-string.gold
func main() int {

    int len = 3;
    string[3] blah;
    blah[1] = "hello";
    int i;
    for (i = 0; i < 3; i = i + 1) {
        blah[i] = "hello";
    }

    print(sprint("%s", blah[1]));
    println("SUCCESS");
}
```

```
//array-with-len-int.gold
func main() int {
      int[4] a;
      int[4] arr;
      arr = {1,2,3,4};
      int[3] b = {0,0,1};

      if (b[1] == 0) {
            //print("b[1] == 0");
      }
      if (b[2] != 0) {
            //print("b[2] != 0");
      }
      print("SUCCESS");
      return 0;

}
```

```
//assign-int.gold
func main() int {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
//    c = a + b;
    println("SUCCESS");
    return 0;
}
```

```
//assign-string.gold
func main() int {
    string a = "a";
    string b = "cat";
    println("SUCCESS");
    return 0;
}
```

```
//atoi-happy.gold
func main() int {
      println("Actual number = 5");
      int number = 5;

      println("Partial number = 45jk");
      string partial = "45jk";

      println("Not a number = erry");
      string none = "erry";

      println("Should add to 50");
      int new_num = atoi(partial) + atoi(none) + number;
      println(sprint("%i", new_num));

      return 0;
}
```

```
//basic-binops.gold
func main() int {
      println("Actual number = 5");
      int number = 5;

      println("Partial number = 45jk");
      string partial = "45jk";

      println("Not a number = erry");
      string none = "erry";

      println("Should add to 50");
      int new_num = atoi(partial) + atoi(none) + number;
      println(sprint("%i", new_num));

      return 0;
}
```

```
//chack-type-assign.gold
func main() int {
    int x = 0;
    println("SUCCESS");
    return 0;
}
```

```
//comparison-int.gold
func main() int {
    int a;
    int b;
    bool check_equal;
    a = 1;
    b = 1;
    check_equal = (a == b);
    if (check_equal) {
        print("a == b\n");
        println("SUCCESS");
    }

    return 0;
}
```

```
//compound-assignment.gold
```

```
func main() int {
      int a;
      a = 2;

      int b = 3;
      a = b;

      if (b == 3) {
            println("Compound assignment works");
      }
      if (a == 3) {
            println("Assignment to a var by another var works as well");
        println("SUCCESS");
      }
      return 0;
}
```

```
//float-fail.gold
func main() int {
    float y = 2.0;
    float x = 3.0;
    int z = 2;
    int duh = z + x;
    return 0;
}
```

```
//float-simple.gold
func main() int {
    float x;
    x = 1.1;
    float y = 2.2;
    x = x * 2.0;
    print("SUCCESS");
}
```

```
//for-test-fail.gold
func main() int {
      int i; //Can't declare in for statement yet if ever
      int b = 5;
      for (i = 0; i < 3; i = i + 1) {
            print("Should be 3 of these\n");
            int test = 3;
            if (b == 5) {
                  print("Inside of for loop has access to variable declared
in function\n");
            }
      }
      print("After the for loop");
      if (test == i) {
            print("Test available out of scope\n");
      } //fails

    print("SUCCESS");

      return 0;
}
```

```
//func-simple.gold
func sayBye() int {
    print("Bye");
    return 2;
}

func sayHi() int {
    print("Hi");
    return sayBye();
}

func main() int {
    int temp;
    temp = sayHi();
    if(temp == 2) {
        print("sayHi worked");
        println("SUCCESS");
    }

    return 0;
}
```

```
//func-simple-args.gold
func sayBye(int num) int {
    print("Bye");
    return num;
}

func sayHi() int {
    print("Hi");
    return sayBye(2);
}

func main() int {
    int temp;
    temp = sayHi();
    if(temp == 2) {
        print("sayHi worked");
    }

    println("SUCCESS");
    return 0;
}
```

```
//func-simple-bool.gold
func sayHi() bool {
    bool temp;
    temp = true;
    return temp;
}

func main() int {
    bool temp;
    temp = sayHi();
    if(temp == true) {
        print("Hi");
        println("SUCCESS");
    }
    return 0;
}
```

```
//func-simple-float.gold
func sayHi() float {
    float temp = 2.2;
    return temp;
}

func main() int {
    float x;
    x = sayHi();
    float y = sayHi();
    //if(x == 2.2) {
    //print("SUCCESSS");
    //}
    if (x + y == 4.4) {
    print("SUCCESS");
    }
    return 0;
}
```

```
//func-simple-string.gold
func sayBye() string {
    return "Bye";
}

func sayHi() string {
    print(sayBye());
    return "Hi";
}

func main() int {
    string temp;
    temp = sayHi();
    if(temp == "Hi") {
        print("SUCCESS");
    }

    return 0;
}
```

```
//function-basic.gold
func print_square(int dim) void {
      int i;
      int j;
      for(i = 0; i < dim; i = i + 1) {
            print("| ");
            for(j = 0; j < dim; j = j + 1) {
                  print("Hi ");
            }
            println("|");
      }

}


func main () int {
      int dim = 8;
      int board_size = dim * dim;
//    int [board_size] current_board;

      print_square(8);
    println("SUCCESS");

      return 0;
}
```

```
//hello.gold
func main() int {
      print("Hello world!");
    println("SUCCESS");
      return 0;
}
```

```
//input-string.gold
func main() int {

      print("Please enter a string: \n");
      string s = input();
      println(s);
    println("SUCCESS");

      return 0;
}
```

```
//keyword-else.gold
func main() int {
    if (4 < 3) {
        print("4 < 3\n");
    }
    else {
        print("3 < 4\n");
        println("SUCCESS");
    }
      return 0;
}
```

```
//keyword-if.gold
func main() int {
    if (3 < 4) {
        print("3 < 4\n");
        println("SUCCESS");
    }
      return 0;
}
```

```
//negation.gold
func main() int {
      int a;
      int b;
      a = 1;
      b = -a;
      if (a + b == 0) {
            print("a == -b\n");
        println("SUCCESS");
      }

      return 0;
}
```

```
//not.gold
func main() int {
      bool a;
      a = false;
      if (!a) {
            print("!a == true\n");
        println("SUCCESS");
      }

      return 0;
}
```

```
//out-of-order-decl.gold
func main() int {
      print("Starting test");
      int a;
      a = 1;
      int b;
      b = 1;
      if (a == b) {
            print("Success at declaring out of order");
        println("SUCCESS");
      }

      return 0;
}
```

```
//pointer-change-value.gold
func main() int {
      string test = "Hey";
      string test2 = "Please";
      string ~ptr = @test;


      ~ptr = "Changed";
      string ~new_ptr = @test;

      println(~new_ptr);
      //Testing for overflow
      println(test2);

    println("SUCCESS");

      return 0;
}
```

```
//pointers-simple.gold
func main() int {
    int a = 2;
    int ~A = @a;

    if (a == ~A) {
        println("SUCCESS");
    }
    return 0;
}
```

```
//scoping-simple-blocks.gold
//Should throw an error
func main () int {
      int a = 1;
      {
            int test = 1;
            {
                  int i = 0;
            }
            //i = 1;
      }
      test = 2;
      if (a != test) {
            print("Test didn't fail, but should");
      }

    println("SUCCESS");
      return 0;
}
```

```
//scoping-simple-int.gold
// Expected Output:
// 3
// 4
// 3
func main() int {
      int a = 3;
      if (a == 3) {
            print("a == 3 before");
      }
//    print(sprint("%i\n", a));
      {
            a = 4;
//          print(sprint("%i\n", a));
            print("a is changed to 4 in the block");
      }
//    print(sprint("%i\n", a));
      if (a == 3) {
            print("a == 3 after\n");
      } else {
            if (a == 4) {
                  print("a == 4 after");
            }
      }

    println("SUCCESS");
      return 0;
}
```

```
//scoping-simple-string.gold
// Expected Output:
// rip
// rekt
// rip
func main() int {
    print("Expected Output:")
    print("rip")
    print("rekt")
    print("rip")
    print("Actual Output:")
    string a = "rip";
    print(sprint("%s\n", a));
    {
        a = "rekt";
        print(sprint("%s\n", a));
    }
    print(sprint("%s\n", a));

    println("SUCCESS");
    return 0;
}
```

```
//sprint-float.gold
func main() int {
      string s = sprint("I am %f years old", 21.34);
      println(s);
      return 0;
}
```

```
//sprint-int.gold
func main() int {
    int a = 1;
    int b = 2;
    int c = a + b;

    print("a is: ");
    print(sprint("%i\n", a));
    print("b is: ");
    print(sprint("%i\n", b));
    print("c is a + b\n");
    print("c is: ");
    print(sprint("%i %i \n", c, c));

    print("SUCCESS\n");
    return 0;
}
```

```
//sprint-string.gold
func main() int {
    string a = "a";
    string b = "cat";

    print("a is: ");
    string c = sprint("%s\n", a);
      print(c);

    print("b is: ");
    string d = sprint("%s\n\n", b);
    print(d);

    print("SUCCESS\n");
    return 0;
}
```

```
//string-comparison.gold
func main() int {
    string word = "a";
    if (word == "a") {
        println("string compar works");
        println("SUCCESS");
    }
    return 0;
}
```