Design Document
Group 9

(Since we didn't finish weeding and pretty-printing properly in milestone 1, the design document details the weeder and pretty printer extensively. If you're more interested in only the type checking part, go directly to type-checker design.)

Weeding

The weeding is done through a single function weed_ast, which does it by visiting all nodes in the provided ast. The entire structure mirrors the declarations in the ast file (including order for readability) and nodes translate to visiting functions of said nodes. All these functions are mutually recursive, and most pass around a linenum argument for proper error message display.

Six errors not caught in the parser are being dealt with in the weeder:

1. Using _ as a value
2. Using wrong expressions as left values
3. Using a short variable declaration in the for loop's third section
4. Having more than one default case in a switch statement
5. Having break statements and continues outside of loops
6. Not having adequate return statements

The first one includes rejecting _ in the right side of single variable declarations, short variable declarations, when using it as a custom type (var x _ = 1) and finally when used inside an expression passed to a function or a type cast. The implementation uses a flag reference to indicate a recent discovery of the blank id, that is checked inside the appropriate statements and thus at the end of the traversal of the relevant expression.

The second one also uses a "global" flag, to determine if the most recent exited expression node was a left value or not. Only ids and indexed expressions are always being treated as left values, while select expressions (for structures) does not update the flag, that is, is only a left value if its expression was also a left value. All other expressions are marked as non left values.

The loop declaration problem was similarly solved, a flag to determine if the short variable declaration is allowed is used. It's different from the previous solutions in that it is the declaration itself and not the loop that's in charge of error detection and reporting. To deal with default statements inside switches, a local reference is used instead of a global one (as the number of switch cases is unknown). Then, every default increases this local reference by one, and upon return to the switch node, the error is reported.

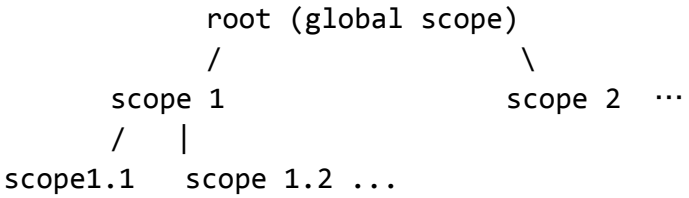
Also, the break and continues problems was fixed with yet another global reference indicating loop nesting. Every loop increases the value on enter, and decreases it on exit. Thus, break and continue nodes can easily check their nesting level, and when 0, report the error. Lastly, returns were handled by visiting all statements in a function declaration and making sure at least one was a return statement, or had all paths covered by them, checked with recursive calling, as well as a default case check in switches, and a condition presence check in loops.

PrettyPrinter

- Indentation is done by adding a "level" argument to all functions and increment the level each time we get into a new scope.
- Block structure (variable declaration block, type declaration block, struct block) is seeing as a list of multiple declaration, where each contains a list of single declaration.
- This way, we can print the blocks exactly as it is in the source code.
- In case of an if_statement without “else”, an empty else block is printed.
- Escaped characters are sorted out from strings and rune so they can be printed as it is.
- Semicolons and commas in lists are printed in top-level functions so the inner printing functions don’t have to worry about it.
- Expression types are cached in the expression nodes once type checking has been done. Therefore printing expression types is trivial. We print the types of all the subexpressions as well, and the type is printed as a comment right next to the expression.

Type checker design:

The symbol table is implemented as a tree of scopes. Each scope has a reference to its parent, and contains a hashtable of all the identifiers declared in the scope.



The hashtable links each symbol to an entry that holds its type and the line number it was declared in, as well as a backword link to the scope. Opening a scope involves adding a new Scope node to this tree, while closing a scope only does some cleanup work (for example printing the symbol table). It should be noted that closing a scope does not destroy the

node. Once built, the symbol table is persistent. The initial_scope function returns the root scope, which is preloaded with the identifiers 'true' and 'false'.

We introduced a reference field in the identifier node of the AST. Once an identifier is looked up, the symbol table entry is cached in the node. In the code generation phase of the compiler, this entry will become the unique identity of the identifier, and will facilitate proper variable name generation.

We introduced a similar reference field in all the expression node, where we cache the type of an expression once we determine it. This helps increase performance in future traversals of the AST (for example pretty printing).

The most important part of the typechecker is getting the type of expression statements - because that is where all the types live. This is done in the resolve_exp_type function.

I mention a few points about the type checker design:

Representing types: Types of golite are represented as a datatype gotype defined in symtable.ml. There is a type called GoCustom that holds its type - this is used for variables that are of custom types. The Struct type holds inside it a map of field names to their respective types.

Type Declarations: New declared types are added to the symbol table as any other identifiers, but the identifier has a special type - NewType of whatever the type is declared to be.

Variable Declaration: When a variable is declared with no explicit type (for example) Function Calls vs Type Cast: Since out AST gets confused about casting to custom-defined types, type checking function calls involves looking up the function name in the symbol table, and if it resolves to a type name, doing a check for types instead.

Void Function Call: One annoying issue was that void function calls don't have types, and therefore it is illegal to have then nested inside other expressions while it is legal to have them as a lone expression statement. Currently, this is handled as a special case - typechecking a void function raises a void function call exception, which is then inspected at the statement type check level and a decision is made about whether the function call is valid or not.

Redeclared Variables: Since we store the line number of each declaration, we can easily print the line number of first declaration when we have a redeclared variable.

Line number handling: The line number handling in our codebase is not very cleanly designed - only certain nodes have line numbers attached to them. Therefore in the type checker, type checking those two nodes involves catching all the errors of the children nodes and printing an appropriate error message with line numbers.

Dumping scopes: When a scope is dumped, we print the name, type and line number of declaration of each symbol.

Testing the type checker

The type checking rules are almost direct translations of the spec to OCaml code, and enumerating the rules will amount to pasting the table of contents of that file here. Here is a list of test files we use to test the typechecker:

append1.go : id is not of type Slice
append2.go : expr is not type T
assign_stmt1.go : assign wrong type to struct field
assign_stmt2.go : assign wrong type to int
binary_exp1.go : '||' arg2 not bool
binary_exp2.go : '&&' arg1 not bool
binary_exp3.go : '==' arg1, arg2 not the same type of comparable type
binary_exp4.go : '!=' arg1, arg2 not the same type of comparable type
binary_exp5.go: '<' arg1 not ordered
binary_exp6.go: '<=' arg1, arg2 not the same type of ordered type
binary_exp7.go : '>' arg2 not ordered
binary_exp8.go: '>=' arg1, arg2 not the same type of ordered type
binary_exp9.go: '+' args not string or num
binary_exp10.go: '-' arg1 not numeric
binary_exp11.go: '*' arg2 not numeric
binary_exp12.go: '|' arg1 not integer
binary_exp13.go: '&' arg2 not integer
block.go : statement ill-typed
field_selection1.go : expr is not of type Struct
field_selection2.go : id is not a field in struct S
for_loop1.go : statement doesn't type check
for_loop2.go : expression not type bool
for_loop3.go : init ill-typed
for_loop4.go : increment expression ill-typed
func_call1.go : wrong arg type
unc_call2.go : too many arguments
func_decl1.go : extra return value
func_decl2.go : duplicate function argument
func_decl3.go : variable already declared

func_decl4.go : missing return value
func_decl5.go : statement ill-typed
if_stmt1.go : init ill-typed
if_stmt2.go : expression not bool
if_stmt3.go : statement body has the wrong type of x
if_stmt4.go : expression in 'else' not bool
if_stmt5.go : statement body in 'else if' has wrong type
if_stmt6.go : statement body in 'else' doesn't type check
indexing1.go : expr is not of type Slice or Array
indexing2.go : index is not int
op_assignment1.go : '+=' expr of left ill-typed
op_assignment2.go : '&=' expr on left ill-typed
op_assignment3.go : '/=' expr on both
op_assignment4.go : '<<=' expr on left is blank_id
op_assignment5.go : '%=' expressions on both side not the same type
op_assignment6.go : '&^=' expressions on both side not the same type
print1.go : can't print struct field
print2.go : one of the expression ill-typed
println.go : expression ill-typed
return_stmt1.go : return value not the same as expected
return_stmt2.go : return value ill-typed
short_var_decl1.go : ill-typed value on right hand side
short_var_decl2.go : no new variable on left hand side
short_var_decl3.go : try assigning a variable to a different type
switch1.go : init does not type check
switch2.go : expr not well typed
switch3.go : case e1, e2 not bool
switch4.go : case e1, e2 not well-typed
switch5.go : case e1, e2 well-typed but not the same type then expr
switch6.go : statement not well typed
type_cast1.go : type is not any of the four allowed (nor their aliases)
type_cast2.go : expr is ill-typed
type_cast3.go : expr well-typed but if type not allowed
type_decl1.go : need type cast
type_decl2.go : type already declared in current scope
unary_exp1.go : expression not numeric after unary '+'
unary_exp2.go : expression not numeric after unary '-'
unary_exp3.go : expression not bool after logical negation
unary_exp4.go : expression not numerical after bitwise negation
var_decl1.go : expression is not of type T
var_decl2.go : expressions on right side not well-typed
var_decl3.go : variable already declared in the current scope

Contributions

Omar: implemented the weeder and tested it, fixed some parser issues
Wei: fixed some parser and ast issues and prettyPrint, created test files and tested type checking.
Deepanjan: implemented type checking and return statement weeding. fixed some prettyPrint issues

Appendix:

Scanner:
Semicolon insertion is done by matching a list of last_token for tokens just before an end of line that requires a semicolon. Everytime the scanner sees a newline, it checks if the last token is matched and insert a semicolon if needed.

Parser:

Lvalue was very difficult to implement in the parser, as it was causing many conflicts to have its own definition, so we decided to use primary_expression on the left side of an Assignment statement and filter out invalid primary expressions in the weeder.

If_statements: in cases where we have “else if”:

```
if⋯ {
}else if {
}else if {
}else{
}
```

it is parsed as

```
if ⋯ {
}else..{
        if . {
        }else {
                if {
                }else{
                }
        }
}
```

so everything after a “else” is included in a new if_statement.

Operator precedence is implemented by defining 5 levels of left associative rules, overwritten by unary operators and parentheses.

AST:

All for_ loops are converted to the three-clause loop.
All if_statements are converted to the two-clause statements