Charles Gingras, Gregory Theos, Song Yang Group 05
COMP 520

Milestone 2

1. Introduction

In the second milestone, we implemented mostly the symbol table and type checker. Modules from the first milestone were also modified to handle extra requirements and to fix some errors. Test files for this milestone are written by hand and by automated scripts in Python.

2. Modifications since last milestone

Our compiler was not error-free. There was a handful of issues with the parsing phase and a few with the scanner. The weeder was enhanced to cover new requirements in this milestone and to cover parsing errors too hard to fix in the *bison* file. After the creation of an automated test script for the pretty printer, the pretty printer is updated to solve some of the detected errors.

- Scanner

Starting with the scanner, there was a bug causing single line programs to not be properly terminated by a semicolon which raised a syntactical error in some instances. Also, hexadecimal integer literals were allowed to occur as only a “0x” string. All were fixed within the scanner.

- Parser

As for the parser, syntactical errors were thrown in cases were the semicolons were present, but only the condition expression was present, or nothing at all was. This was easily fixed by adding those cases into the grammar. A similar issue also occurred with if and else if statements when only a semicolon was present without any preceding assignment, quick declaration, or function call.

Our parser was not matching on the built-ins (append, cap, len) to make sure the right number of arguments were passed. Instead, we were planning on doing that check during the type checking, just like for all other functions. It was an easy fix by simply letting the non terminal for factors to expand to those specific calls, with a predefined number of arguments.

One of the trickier fixes that was brought up was that a variable can be declared to have type struct, followed by a declaration of the structs contents. This was easy to fix by moving some of the rules related to structs to the non terminal for types. This impacted the AST structure so some changes had to be made to better reflect the structure of the input program.

Function identifiers can now be wrapped in parentheses in function calls.

Lastly, parser conflicts were eliminated, thanks to the teacher's help, and the grammar rule for expression was collapsed into one by adding new precedence directives.

- Weeder

The new requirement that a non-void function ends in a terminating statement is added to the weeder. The previous use of one boolean return value for recursion calls is not enough and is changed to a struct of booleans. This allows the keeping track of terminating statements, breaks and default cases for verifying the termination in switch statements.

After the release of the blank identifier document on the course webpage, the weeder also checks for the blank identifier used as a struct field access (*e.g.* a._ = 3).

Short declarations were not properly handled by the parser in the previous milestone. Since changing the grammar to solve this issue creates 10+ parser conflicts, we decided to let the parser accept more than it should and reject invalid short declarations in the weeder. Previously, short declarations in our grammar can have expressions on both sides of the assignment operator. But after checking with the solution set, we realized that the left hand side can only have a list of identifiers and enforced this restriction in the weeder.

- Pretty Printer

The pretty printer could not pretty print its own output. Syntax errors and segmentation faults were thrown by running the pretty printer on already pretty-printed *.go* files. Many bugs related to this are fixed. However, the pretty printer output is aesthetically different from the reference and sometimes, also syntactically different from the reference compiler. Lastly, the pretty printer invariant seen in class does not hold with our pretty printer. Parentheses are nested each time an expression is pretty printed. This leads to a file that grows ever bigger each time the pretty printer is run.

3. Symbol Table

The symbol table is implemented in a way that distinguishes explicitly between declared variables, types, and functions. It does so by referencing a separate array for each of them. This would likely be inadequate for a real life compiler since this triples the memory cost from arrays. However, it helps keep the code somewhat cleaner by avoiding an excess of flags and makes it easier to reason about.

At the very top level of the program, the table is initialized with types *int*, *float64*, *string*, *bool*, and *rune* inside of it. The boolean values true and false are also added as constant variables. A value, in any array of the table, is represented by symbols: a symbol is a construct that holds the identifier of the value it is representing, contains a reference to its

type (using the same type nodes as in the AST) and contains a reference to a parent symbol. The parent symbol refers to the type symbol of the current symbol's type: in plain English, this means that in "*var a num*", the symbol for variable *a* refers to the symbol of type *num*. The same applies for types. Symbols for structs will not have such a reference.

The way type nodes are constructed lets them be linked in a "chain" by using the parent symbol to walk through the type definitions.. This allows easy support of type declarations and type checking. Here is an example:

```
type num int
type numlist []num
```

A type node with identifier "numlist" would be labeled as a slice type and would point to the "num" type node. The "num" type node would be labeled as a base type and would point to the "int" type node. This way, we have a type hierarchy that contains information about a type and its "parents". These linked lists are partially constructed by the parser as it builds the AST. At this point in the compiler, the type nodes only ever point at "dummy" nodes. In our previous example, the "num" dummy node would not point to the "int" node. As the symbol table is constructed and as type declarations are encountered, the symbol phase takes care of updating the pointers types have to their "parents" by looking up the corresponding parent symbol in the table.

The tricky part is handling type redefinitions, such as redefining the type "int". All predefined types point to a single empty type whose identifier is " ", a space. We can thus distinguish type "int" from a redefinition of type "int" by looking at which type each inherit from. We further support this by keeping a static global reference to the 5 base types' symbols when the symbol phase begins and henceforth. Type equality is determined purely by parent symbol equality (*i.e.*, pointers to the type symbol are equivalent). Since no explicit symbol is created in the case of slice and array types, some inspection of the type AST node is also necessary to determine equality: the full declared type (referenced by the symbol) is stored into a string and then string comparison determines equality.

When a type is used (*e.g.* in the context of a variable or type declaration), but is not found inside the table, an error is thrown. Likewise, when an identifier is used, but not found in the table, an error is thrown. Hence, the compiler does not support circular function calls. Upon creating a new symbol, a check must first be done to ensure that the identifier of this symbol is not already used across all 3 arrays of the table. Only the local scope is verified for the symbol kind that is being used (var, type, or function, despite the fact that we do not support nested functions), but all above scopes are verified for the other 2.

Some specific checks are done when encountering definitions and assignments to avoid putting the blank identifier into the table and looking it up, respectively. Rules for functions main and init are enforced upon placing the symbol in the table.

4. Type Checking

The bulk of the type checking phase consists of determining expression types. When the type of an expression is determined, if it is correct, the type is stored in the expression node for later use, and a symbol is passed back. The symbol will usually be a symbol for a type, but in the case of identifier expressions, a variable symbol is passed back instead.

Determining type compatibility in the case of binary expressions is the same as determining type equality between both branches. An extra check will need to be performed so that the types (or their underlying types) are valid given the operator of the expression.

Trickier cases involve specifically dealing with underlying types, especially when those underlying types are structs, arrays, or slices. These come up often especially in the context of the built-in functions append, cap, and len.

To properly do this, we need to follow the linked list of types/symbols and accept if we encounter a valid type (*e.g.,* slice for append) and raise an error if we encounter an invalid type (*e.g.,* array, struct, or something other than slice for append). In the specific case of append, we must also make sure that the second argument is of a proper type given the first argument. This is done by comparing their parent references and comparing the string of their types, as was previously explained for type equality. Note that this also applies in the case of expression resulting from struct member access or array/slice indexing.

When the type checker encounters a type cast, it first treats it as a function call (since the parser cannot distinguish the two). If the identifier is not found (in the function table), then a lookup is made in the type table. A successful lookup begins the type checking for the type cast:

-first we make sure a single argument is passed, otherwise an error is raised;
-the type of the argument is determined;
-a check is performed which first tries to find the underlying type of the cast and the argument and then will see if those underlying types are base types (*e.g., float64*,*int,...*). Type cast validity is determined by hard coding which base type can be cast to which other;

If instead the type checker finds a function symbol for the current function call expression, it iterates through the arguments of the symbol and those given to make sure the types match. If so, the returned symbol is the type which the function is declared to return.

For comparable expressions (==, !=), a helper function deals with traversing a type/symbol linked list to determine if the type is comparable.

At the top level, the type checker goes through declarations. As it encounters variable declarations, it makes sure to update the symbols of the variables which were declared without a type. Upon encountering function declarations, their statements are typechecked.

When type checking statements a reference of the function symbol is kept so that return statements may be flagged as returning the wrong type. Type checking other statements consists of properly propagating type check calls and making sure conditionals, assignments and quick declarations are well typed.

5. Testing

The test files for the weeder are in *programs/1-scan+parse* within a *weeder* directory. They test for terminating statements.

The type checking test files are organized in the *programs/2-typecheck* directory as follows:

- *valid/*
  - *milestone1/* : valid programs from milestone 1
  - *weeder/* : valid programs to test the weeder in *1-scan+parse*
  - *shadowing/* : valid variable, type and function shadowing
  - *initialization/* : initialization with *var* (var x int = 3)
  - *short_declaration_assign/* : short declaration followed by a valid assignment to the same variable
  - *assign/* : valid declarations followed by assignments
  - *assign_literal/* : valid declaration followed by assignment of a literal
  - *blank_id/* : valid usage of blank identifiers in terms of typing
  - *call/* : valid function calls with correct types passed to the arguments
  - *multi_assign/* : valid assignment with many variables to the left and expressions to the right of the assignment operator
  - *multi_short_declaration/* : valid short declaration with many identifiers to the left and expressions to the right of the operator
  - *return/* : valid assignment of returned values
  - *binary/* : valid usage of binary operators
  - *if/* : valid usage of if statements
  - *op_assign/* : valid usage of operator assignments
  - *print/* : valid usage of print and println
  - *unary/* : valid usage of unary operators
- *invalid/*
  - *undefined_var/* : undefined variable
  - *undefined_functions/* : undefined function calls
  - *initialization/* : initialization with *var* with conflicting types and literals
  - *short_declaration_assign/* : short declaration followed by an invalid assignment to the same variable

    - *assign/* : invalid assignment of declared variable to one another
    - *assign_literal*/ : invalid assignment of literals
    - *blank_id/* : invalid usage of the blank identifier (some are already caught by the weeder)
    - *call/* : invalid function calls due to mismatch of argument types
    - *multi_short_declaration/* : invalid short declaration with many identifiers to the left and expressions to the right
    - *redeclaration/* : redeclaration of variables, types and functions in the same scope
    - *return/* : invalid assignment returned values
    - *special_functions/* : invalid usage of *main* and *init*
    - *undefined_type/* : using previously undefined types
    - *binary/*: invalid usage of binary operators
    - *built_in/* : invalid usage of built-in functions
    - *field_selection/* : invalid field selection in structs and other types
    - *for/* : invalid for statements
    - *if/* : invalid if statements
    - *increment_decrement/* : invalid increment and decrement
    - *index/* : invalid indexing of variables
    - *op_assign/* : invalid usage of operator assignments
    - *print/* : invalid usage of print and println
    - *switch/* : invalid usage of switch statements
    - *typecast/* : invalid usage of typecasts
    - *unary/* : invalid usage of unary operators
- *generate.sh* : automatically generates a large subset of the test files
- *destroy.sh* : automatically deletes the files generated by *generate.sh*
- many other Python scripts that create or destroy test programs

6. Teamwork

Charles: symbol table, type checking, parser bug fixes from milestone 1, bug fixes in general.
Song: weeder, parser conflict resolution, grammar refactoring, test programs, error reporting
Gregory: Minor symbol table, type checking