Song Yang, Charles Gingras, Gregory Theos
Group 05
April 10, 2019
COMP 520

Writing a Compiler on a Subset of the Go Language

Introduction

In the context of a university course, we created a working compiler for the GoLite programming language, a subset of the Go programming language designed by Google. There are major differences between the two, namely in the program encoding, pool of basic types, grammatical constructs, language semantics and language built-in functions.

In GoLite, the program is encoded in ASCII, whereas, the Go programs are encoded in Unicode. GoLite has also a much smaller set of available basic types to use and more limited grammatical constructs. Mutual recursion is not allowed in GoLite, but possible in Go. GoLite does not allow declaring grouping type literals to variables, but Go does.

In Go, functions may return many values, like in Python. In GoLite, at most one return value is allowed. In Go, pointer types exist, but not in GoLite. There are features in Go like channels that are not supported in GoLite. Lastly, only a few of the built-in functions are available in GoLite.

The main section of this report summarizes the development of the lexer, parser, AST, weeder, typechecker and code generator. Across all modules, test files are written and added to a general test set. Some sections of the report will feel repetitive to a reader who has already read the milestone reports. For sections where a comment from the instructor was given, a special response is added.

Language and Tool Choices

Before talking about each individual module of the compiler, we'd like to thank and point out the existing compiler tools we used.

All of our compiler is written in C. The tools we used for the lexer and parser are *flex* and *bison*. The chosen C compiler is the GNU *gcc*. All of the chosen tools are free software. The main reason for choosing these tools is that they are also the ones showcased in class, which eases learning and finding help.

For running our compiler, we solely focus on Linux systems available at McGill university's computer science servers. As such, there are Bash and Makefile scripts supplementing the main C source files, mostly for building and testing. The writing of the test suite is done both by hand and by hand-written Python scripts. Git is the selected tool for version-control, because GitHub provides remote hosting of repositories free-of-charge and because the course requires all milestones to be handed in on a private GitHub repository.

The target language for our GoLite compiler is C, again. Despite large differences between GoLite and C, this is the language we are the most familiar for this course. Furthermore, C is relatively fast to compile and run, which decreases the time to run the test suite. It is worth pointing out that C has no object-oriented support and poor run-time errors, which increases the difficulty of this project.

Modules

Lexer

No changes were made to the lexer since the second milestone.

The lexer was implemented using flex. It identifies the GoLang keywords and operators, integers in decimal, octal, and hexadecimal format, floating point numbers with at least one number on either side of the decimal point, standard and raw string literals, and individual characters (i.e. runes).

A token is returned for all matches unless the match is a single character operator (e.g. +,*,&,!). We opted to convert the integers received in octal and hexadecimal formats into the corresponding decimal value stored inside the token %union. This avoids confusion in the later stages of the compiler, particularly in the parser and AST, and fewer grammar rules need to be written as a result and fewer node constructors need to be defined.

Likewise, raw strings are converted to strings by being parsed and converting escape characters to their formatted counterparts (e.g. \n becomes \\n). The raw string delimiters (`) are also converted to double apostrophes. We keep the double apostrophes inside the string since it will avoid us the task of generating them back during the codegen phase.

Multi-line comments gave us some troubles. We initially attempted to recognize them using a simple regex expression (“/*”.*”*/”). This does not output unclosed block comment errors properly. Instead, we had to go about declaring explicit DFA states to better recognize the blocks. Upon encountering a “/*” expression, the lexer enters an exclusive “IN_COMMENT” start condition. Note that since flex prioritizes longer matches, if “/*” appears in the context of a string, the start condition is not activated. If it ever is, then it discards all inputs and exits the condition on the first “*/” match. Encountering the end of a file while in it results in an error (unterminated comment).

Due to the design decision of the original Go language, a block comment is transformed into a semicolon by the lexer when the preceding token triggers the semicolon generation rule, as described in the specifications.

Another challenge that was mostly tricky to notice had to do with the ending of a file. If no new-line was present, then no semicolon is generated even if one should be (e.g. if the file ends with a closing function curly brace ‘}’). This is dealt with using start conditions again. An

“EOFCHECK” start condition is entered at the very beginning of the file and allows us to match on the end of file pattern. When this occurs, we simply return a semicolon if necessary, while making sure to leave the start condition.

Parser

The parser was implemented using bison. It consists of a context-free grammar centered around certain groups of non-terminals mostly aligned with the major constructs of an imperative programming language like GoLite.

All previous attempts to avoid parser conflicts by increasing the grammar were replaced by more precedence directives. New directives are for struct field access and parenthesized expressions.

Declarations of top-level variables, types, and functions are grouped into a top-level program definitions node.

Statements are grouped into simple statements (which can occur before the condition in certain other statements such as “for”) and plain statements, which can themselves be simple. Simple statements are assignments, increments/decrements, and expressions (such as function calls). Plain statements also include prints, breaks, continues, returns, ifs, fors, switches, and declarations.

Most of these major groups have block forms which can occur in a variety of contexts. In the case of identifiers and expressions, these blocks are separated by commas. In addition to blocks, declarations and assignments can be constructed to declare/assign many variables in a single line. For multiple assignments that only use one “=”, each side of the assignment operator is kept as a block, since Go semantics does not allow a simple expansion.

After milestone 2, conditional statements and loops were corrected. Some inputs were rejected because a condition or operation was missing inside an if/elif/for header, but a semi-colon was still present. An expansion of the grammar rules of if/elif/for tokens was enough to resolve these issues.

AST

The AST is made of 6 different types of nodes: program nodes, statement nodes, declaration nodes, function nodes, type nodes, and expression nodes.

Program nodes represent the top level structure of a program and keep track of package, variable, type, and function declarations.

Type nodes contain the name of the type, an enum of the kind of the type (i.e. slice, array, user type, …), and a size attribute that is only used in the case of arrays. Base types are also instantiated into a type node, since Go allows shadowing of base types. The enum in the type node allows better type checking. For user-defined structs, the type node will point

to a body of fields, which is a declaration node. For all other user-defined types, the type node contains nothing but a pointer to a previously instantiated type node.

Declaration nodes can hold any variable, type, and function declarations. Declaration nodes contain information about the declared type if any, the identifier associated with the declaration and the declared value if any. This declared value can be either an expression in the case of variables, or a function node in the case of a function declaration. For user-declared types, the name of this new type is stored in the declaration node and the underlying type referenced by a pointer to a type node.

Declarations can be linked in a list. In some quirky cases, a second pointer was also necessary to avoid the an entire list to be lost except for the head. This would occur inside the distributed declaration construct where multiple declaration node would be assembled into a list. However, the head's next field would be changed to the top level declaration in some cases, losing almost the entirety of the list.

Function nodes keep track of the function identifier, the argument count and types, its return type, and a body of statements. The arguments are stored as declaration nodes under a special enum, which signals it does not have a right hand side. Function nodes can be wrapped by expression nodes and declaration nodes depending on the context of the occurrence (function calls in the former case, function declaration in the latter). It is not always possible to specify the name/identifier of the function (e.g. "a[3]()" is valid in the reference compiler, without a clear identifier attached to it, at least from the parser's point of view). In the case of function calls, the function arguments are stored as a list of expressions, wrapped inside the declaration field of the function node.

Statement nodes store a value and can be linked in a list. Their value can be an expression, a declaration or a statement body (for block statements). It could also be more complex: in the case of conditional statements (if, elif, else, for, while), a struct is maintained to store the condition expression, optional declarations, the statement body, and following statements.

Note: The "following statements" should not be confused with the "next" statement in the linked list. Rather, it will refer to an "else if"/"else" statement in the case of "if" and "else if" statements and to the for loop statement (the one executed after each loop iteration) in the case of a for loop.

Assignment statements are stored as an expression representing an identifier and another expression representing the new value. Multiple assignments are also stored this way and are assembled into a chain of "following" statements. Short declarations are also stored in the same manner, but are assigned a different flag so that the rest of the compiler can distinguish both cases. In particular, this flag will be useful during the type checking phase to ensure short declarations contain at least 1 new variable, while no such condition exists for multiple assignments.

Print statements only need to keep track of the expressions to print and whether or not to print a new line at the end.

Switch statements are broken down into 2 components: a switch body and case bodies. The switch body keeps a reference to the condition, any optional declarations and the case bodies. The case bodies keep track of the case condition (a NULL value represents the default case). The body of statements for a case is also stored.

Lastly, expression nodes can either represent a literal, an arithmetic operation, or a function call. The type of arithmetic operation will determine if the node has 1 or 2 children. In the case of a function call, a new function node is constructed and the given arguments will be stored as described above.

Weeding

After the construction of the AST, it is traversed in order to be weeded of certain invalid constructs. These are:

- Division by 0 numeric literals
- Switch statements with more than one default clause
- All functions with a non-void return type end in a terminating statement
- Break and continue statements only occur in the correct context
- Blank identifiers do not occur in expressions or on the right hand side of assignments
- Simple statements which occur in if, for, or switch statements cannot be expressions unless the expression is a function call.

The weeder is implemented by a top-down traversal of the AST. Recursive calls carry information about previously seen patterns. The return value from the recursion tells whether a certain pattern was found in an AST subtree overall. For example, the rewinding of the call stack is used to confirm the presence of a breaks in switch statements when determining the presence of terminating statements.

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.

Song Yang, Charles Gingras, Gregory Theos
April 10, 2019

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.

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 that is being created (var, type, or function, despite the fact that we do not support nested functions) to support nested scopes and shadowing.

When a function declaration is encountered, a few things need to be done before the symbol is placed into the symbol table. Notably, the function arguments are stored into the symbol for future reference as symbols themselves. Then a subscope is defined, in which the function arguments are stored. A last subscope (to the current subscope) is defined which will be the scope of the function body. This allows the function arguments to be redefined within the function body.

Similarly for structs, the struct fields are also assigned to symbols, but these are not stored in a symbol table. They are simply there for future reference in the typechecker to allow code reuse in the case of symbol compatibility.

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.

Type Checking

Most of the type checking is on expressions. 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 used for representing 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 short declarations are well typed. In the case of switch statements, the switch condition is typechecked and then the cases are typechecked against the condition. If there is no switch condition, the cases are typechecked against the boolean type.

Code Generator

Our chosen target language is C, mostly because of our familiarity with the language. C and GoLite, however, have significant differences which required special care when dealing with certain constructs.

First and foremost, any new declaration will use a new identifier that consists of the string concatenation of a constant string “CODEGEN_BINDING”, a number which is incremented every time it is used, and the original identifier of the declaration. This allows us to avoid the use of a reserved keyword in C, but that is not enforced in GoLite (e.g. restrict). Likewise, it covers the case of shadowing variables, by simply declaring a new variable in its stead.

In GoLite, arrays and slices support bounds checking. This is done by wrapping arrays and slices into a struct which holds a “size” and “cap” field (only slices have a “cap” field). The array of values is always given the identifier “val”. The type name given to this struct will be

determined in a similar way to declarations (see above). Likewise, struct types cause a new struct to be defined with the types corresponding to the struct's definition. This is done whether the struct type is first defined through a type declaration or in an "inline" manner. These definitions are stored by creating a new symbol that represents the original type (array, slice, or struct) and is assigned the typename in a "bindingName" field. This symbol is stored in the symbol table construct, in a list separate from the actual symbol table. This allows us to check for pre-existing bindings of types while at the same time keeping them in a similar hierarchy to the symbol table.

At the beginning of the codegen phase, all the declarations at the top level are parsed to check for two things: init functions and functions in general. All encountered functions will have a header printed so that the generated C code can compile when functions are not defined in a specific order. Init functions will also assemble into a list, which will be used to make all the init calls at the beginning of the main function. Before printing out a function header, the return type and the arguments of the function need to be inspected so that type declarations are made for slices, arrays, and "inline" structs. For this to be doable, we also need to look for type declarations (specifically structs) so that the C compiler has a reference to these user-defined types in case that they show up in the function signature.

Variable declarations are followed up by initialization of the declared variables. In the case that a RHS value was given, the new variable is assigned this value. Otherwise, it is assigned a default value. Some helper functions were defined for the cases of structs, slices, and arrays. In the case of structs and arrays, every element is assigned by value. In the case of slices, the "size" and "cap" fields are assigned by value, but the underlying array is assigned by reference. Since this array changes in size through append calls, it is declared as a pointer. Its default value is the null value. As append calls are made, the underlying array of a slice struct is allocated memory proportional to its new capacity. When present at the global scope, variable initialization/assignment is delayed until we enter the main function since the C compiler does not support them otherwise.

Type declarations are only handled in the case of structs. Otherwise, variables will be declared with their resolved underlying type. This is not an issue since the typechecker already determined the correctness of the program and the user defined types do not change the execution of the program.

Assignment statements and short declarations are dealt with in a similar fashion to variable declarations. First, temporary bindings are declared and are assigned values on the RHS of the statement. Then, the expressions on the LHS are parsed and sequentially assigned their corresponding temporary bindings. In the case of structs, arrays, and slices, and element-wise approach is used (see the variable declaration section for more details). In the specific case of short declarations, a declaration will be made if necessary.

If and else if statements are always nested in a block statement to maintain the behaviour of short declarations before the condition. These short declarations are placed inside the block statement, before the if/elif statement.

All loops are converted to C while loops. Short declarations used for init statements (in “for” loops) are dealt with by being placed in a block scope before the actual loop, just like if/elif statements. We deal with break and continue statements by creating labels at the end of the loop to which we jump. Before an actual continue or break statement is executed, the post-loop statements are executed, if any. The labels are arranged in a way such that if execution proceeded without ever encountering a goto, it would encounter the continue labels before the break ones, if any. This would lead to execution of the “post-loop” statement, if any, and then execution of the continue statement, instead of the break. Note that a continue statement is always placed for this reason.

Codegen of switch statements is approached as was described in class. Cases are checked in an if statement and execution flows to an else branch, possibly containing more cases, when the case is evaluated as false. Break statements are converted to a goto statement which directs execution outside of the if else body.

Codegen of expressions is generally straightforward with a few exceptions. A forward check need to be done for string concatenation, append calls, and slice/array access. String concatenation is dealt with if encountered in the forward check and the result is stored in a new binding. Likewise for append calls, the slice’s underlying array is potentially extended to increase capacity and the resulting slice is stored in a new binding. Slice/array access consists of verifying that the access is within the bounds of the construct. In the append and element access cases, special care is given so that expressions are not executed more often than they are supposed to: to do this, we store the evaluated expressions (notably indices, but also array/slice references) inside temporary bindings and access these bindings when we need access to the values.

Struct and array equality require some specific helper functions so that pairwise equality between each field can be verified. String comparison uses the C function “strcmp”. In the case of slice/array access, care must be given to insert “.val” before the [] operators.

Generating print statements is generally straightforward, except for float expressions. C is supposed to support the following format `printf(“%+1.6e”,x)` which should print x in a scientific format with 6 decimal places, a ‘+’ if the number is positive, a ‘-’ otherwise, and the exponent with 3 significant digits. However, in one of our test cases (initialize.go), printing 2 floats on the same line cause the second ‘+’ to not print. Using a different test script, the ‘+’ would now print, but the second exponent would now only have 2 significant digits. Hence we wrote a custom algorithm to print out floats: it stores the float value in a temporary binding. It divides it down or multiplies it by 10 until it is within the range (10,-10), while keeping track of the exponent. Then it prints out the float, prepended with the right symbol +/-, and followed by its exponent, also prepended by the right symbol. This caused the behavior of the compiled “initialize.go” file to remain the same.

Testing

Test files for the first, third and fourth milestone are written by hand. Only the second milestone contains files automatically generated by scripts.

Because of the milestone guidelines, 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

The semantics test files are located in *programs/3-semantics/valid*, while the benchmark test files are located in *programs/3-benchmark/valid*.

Semantic tests are based on the tutorial slide posted by the instructor. Nearly all of them are edge cases of the Go language that would be problematic to translate in C. Benchmark tests are computationally expensive programs designed to run 5 seconds on the executable generated by the standard Go compiler. Our compiler generates inefficient code, however, which led us to reduce the size of the input of the benchmarks in order to see the code finish.

There are overall 700 test files written for the entire test suite. They do not cover all cases to be caught, but they cover a sizeable portion of all cases that a well-written compiler might fail. In fact, some of these programs have exposed failures in the reference compiler and elicited interesting conversations about the Go language's design.

Conclusion

This 8-week long project is one of the largest academic projects for most of us, or probably the largest. In terms of team work, we learned about the importance of planning and communication. Our team dynamic is very individualistic. We did not have any meetings or any planning. The work separation was done so that the least possible coordination was necessary. As such, there was minimal design conflict, or just conflicts in general, and a lot of non-coding time was saved. However, this approach led to unfair work distribution, poor design and probably not the best end product possible from our team.

On the more technical side, we learned what makes a good programming language good. From the syntax to the semantics and ease of use, Song rules that Go is a language for writing short scripts, like Python, not for large programs. Although it is unlikely that we will build another compiler, the tools we used and the experience gained from the compiler project gave us better appreciation for programming languages in general and for the underestimated field of compilers.

Contributions

Contributions in the entire project are unfairly divided.

- Song:
    - Almost all of the test files (690/700)
    - Weeder
    - Revision changes to the scanner and parser
    - Test scripts, Makefile, C's main file, test suite organization and execution scripts
    - GitHub issue opener and editor
    - Proof-reading on entire reports and writing on selected sections
        - Introduction
        - Language and tools justification
        - Weeding
        - Testing
        - Conclusion
        - Contributions
- Charles:
    - Scanner
    - Major revision changes to the parser
    - AST and $$ constructor calls in the parser
    - Revision changes to the weeder
    - Symbol table
    - Type checker
    - Code generator
    - Writing on selected sections of the reports:
        - Scanner
        - AST
        - Symbol table
        - Type checker
        - Code generator
- Gregory:
    - Parser
    - Small initial draft of type checker and code generator
    - Remaining test files (10/700)
    - Writing on selected sections of the reports:

- Parser