COMP 520 - Final Report (GoLite)

Group 14
Alexandre St-Louis Fortier (260720127)
Stefan Knudsen (260678259)
Cheuk Chuen Siow (260660584)

April 15, 2016

1 Introduction

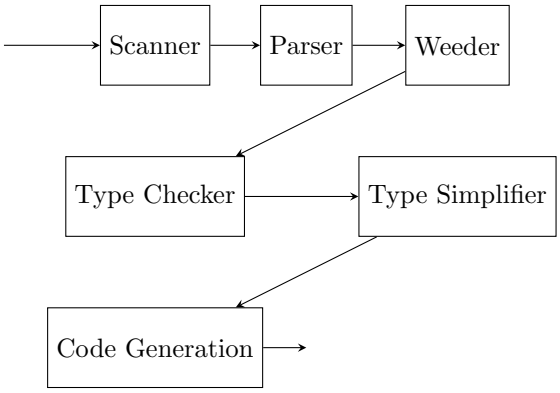
New programming languages are being conceived every other year to tackle various kinds of problems. The structure of a computer program are dependent on the syntax and semantics of a given programming language, and the correctness of the program has to be exact for a compiler to translate it to a machine language. Since the number of programming languages is on the rise, the study on compiler design is imperative to actually understand the tools and how the whole compilation process works. Our group decided to work on the GoLite project as the source-to-source compiler.

We chose OCaml [1] as our implementation language for compiler construction because the recursive nature of the language allows for easy manipulation of abstract syntax tree (AST). In addition, the pattern matching feature that OCaml provides is particularly helpful in constructing the compiler and has been applied to most parts of the compiler. The tools that we used for building the scanner and parser are `ocamllex` and `Menhir` [2,3,4]. The maturity of the parsing and scanning tools were another reason for the choice of OCaml. What's more, the lead TA had pointed out that OCaml is a good language for such a project.

As for the target language, we have chosen to generate WebAssembly (or `wasm`) [5], a low-level programming language that attempts to be more efficient than JavaScript for the web browser. In particular, our compiler generates WebAssembly [6,7]. Since WebAssembly is currently in the experimental stage, only a handful of documentations about the language exist and even some of them are not complete. So far we have only encountered 2 compilers that translates a particular language to WebAssembly: emscripten [8] from LLVM to JavaScript (`asm.js`, the predecessor of `wasm`), and ilwasm [9] from C# to `wasm`. As such, our compiler that takes GoLite programs and generates WebAssembly is a timely project.

The next section is about the design of the compiler structure and all its phases, followed by some examples comparing GoLite programs and the generated WebAssembly code, and ended with a brief discussion and future work.

2 Compiler Structure



2.1 Scanner

The lexical analysis phase of the compiler scans the source code (GoLite program) as a stream of characters and searches for valid tokens that are defined in `lexer.mll`. Programs with any tokens that are not recognized by the scanner such as invalid literals or block comments will be rejected by the compiler, otherwise it will proceed to the next phase.

For inserting semicolon token according to Go semicolon insertion rule 1, we added a variable `insert_semic` for each of the tokens that define the insertion condition. Upon reaching a newline or end of file tokens, our program checks whether the previous token that satisfy rule 1 has updated `insert_semic`, and if it evaluates to `true` it will trigger the insertion of the semicolon token.

We also made a decision to handle types as identifiers and not include the primitive types as tokens. This way our parser will be able to accept both primitive types and type aliases when specifying types. We also decided to defer the check for invalid use of primitive types in variable identifiers to the weeding phase.

2.2 Parser

The syntax analysis phase of the compiler parses the tokens generated by the scanner and checks whether the combination of the tokens form a syntactically valid grammar. This phase is needed due to the limitations of regular expressions—context-free grammar is able to recognize balancing tokens for example. Production rules are defined in `parser.mly` which generates a parse tree for the subsequent phases.

Note that we took the token definitions from `parser.mly` and placed them into a separate file `tokens.mly`. Along with the flags defined in `myocamlbuild.mly`, this enables the main program to display a list of tokens from the tokenizer.

2.3 Weeder

The weeder checks for invalid programs that is difficult to implement in the parser. In particular, we check for

- Invalid use of basic types (`int`, `float64`, `bool`, `rune`, `string`) as an identifier, since we didn't specify those basic types as keywords.
- Invalid use of underscore.
- Negative index for arrays.

- Invalid use of expression as expression statement or left value.
- Multiple `default` in a switch statement.
- Invalid use of `break` and `continue` outside of loops.

2.4 Symbol Table

Initially, the symbol table consisted of a list of maps (as in dictionaries). We briefly considered the alpha-renaming approach, but since hash tables are typically used for symbol table because of its performance, we ended up changing the symbol table to utilize hash tables instead.

The way we design the symbol table is to implement a cactus stack of hash tables, as had been presented in class. The root represents the global scope with a hash table which stores the information for top-level declarations. Whenever a block is entered, a new frame is generated. This frame (`F`) has a new hash table associated with the new scope, and also has a pointer to the parent frame. This allows for the lookup of a declaration starting from the innermost scope that `F` resides, and recursively traverse towards the root node through the parent frame, but not the neighboring nodes as the scopes of these nodes and `F` are disjoint.

We went a little off of what was suggested to handle some of the primitives. This can be seen with our extended type system which includes `TKind` types. We use `TKind` as the type of variables. So all base types have type `TKind(TSimp("#"))` where `"#"` is the type at the top of the type hierarchy. The use of `TKind` allowed us to handle type checking as the function call of a `TKind` identifier. Accordingly, `type` statements add a symbol of type `TKind` to the symbol table.

Since our AST doesn't encode the last position of a node, it was difficult to print the last line of a scope. Instead we provide an alternative view of the symbol table with the `-smartsymtab` flag which gives a better view of how some symbols shadow others.

2.5 Type Checker

Type checking closely followed the `typechecker.pdf`. New scopes are added at the beginning of every block, and at the end of every block, the top hash table is popped. At variable, type, and function declarations, a check is made to see if the `id` has been declared in the current scope. A type error is raised if it's already been declared.

Because of the initial idea of using an immutable map rather than a mutable hash table, type checking consisted of "threading" the context through lists, that is, mapping while passing each context onto the next element of the list. Since we decided not to use the map, this is no longer needed, (simply mapping suffices).

Because type declaration is allowed in GoLite, the approach of matching the types and operators that we had seen for the `minilang` compiler wouldn't work. We took the approach of pattern matching on the operator and check if two types were unifiable. If their upper bound were belonged to the class of types allowable for the operator, then the upper bound was the type returned.

2.6 Type Simplifier

Before passing the annotated AST to code generator, we convert type aliases to their base types. This is to simplify the code generation phase as WebAssembly has only numbers as primitive types and does not support type declarations.

2.7 Code Generation

There are two kinds of formats that represent WebAssembly code: WebAssembly TextFormat (`.wast`) code which uses an AST representation in S-expression syntax, and WebAssembly binary-encoding (`.wasm`). Current tools that deal with these files include `sexpr-wasm-prototype` [10] which translates WebAssembly TextFormat to binary-encoding, and `binaryen` [11] which is a compiler that parses and emits WebAssembly files. Our compiler generates the WebAssembly TextFormat code.

Below are some of the constructs that are uncommon and worth mentioning here. We follow the specification as defined in [6,7].

2.7.1 Local variables

WebAssembly requires that variable declarations in a function to be defined at the top of the function body (`local`) before they are used. As such, all variables local to a function has only a single scope. To overcome this limitation, we append the GoLite type and scope level to the variable names. We use `set_local` and `get_local` instructions for any operations that deal with global variables.

2.7.2 Global variables

WebAssembly does not support global variables. The way we implement global variables is to make use of linear memory. For each top-level declarations, we enclosed them in functions which will be called at the beginning of the start program, and assign a location to the memory address to store the declaration. This information is stored in a hash table for later usage. We use `store` and `load` instructions for any operations that deal with global variables.

2.7.3 Types

WebAssembly supports 32-/64-bit integer and 32-/64-bit floating point as the base types, but not the others. We take the 32-bit integer to represent the integer in GoLite, whereas rune, bool, and string are represented as 32-bit integer in WebAssembly. Strings are stored in memory, so integer type for strings represents the memory location.

2.7.4 Control flow structures

WebAssembly has basic control flow structures including `block`, `loop`, and `if` statements. We make use of `block` and `label` to implement `for` loops and `switch` statements. For nested statements with multiple scoping, we can make a reference to a named label that is defined by the closest outer enclosing construct, so labels in different scopes but have the same name do not have conflicting results.

2.7.5 Print statements

WebAssembly does not have native support for printing to console. A way to include printing is to import a module from `spectest`, but that also has its limitations in that it can only print WebAssembly integers and floats verbatim. Alexandre implemented proper printing mechanism in `binaryen` in his forked copy of the repository [12], and we use that program to parse the generated S-expression code.

3 Examples

4 Challenges Faced

This project is challenging not only because WebAssembly is a low-level programming language, but also the scarcity of information and documentation about WebAssembly that makes it much more difficult for us to implement the code generator, especially for certain constructs such as GoLite types and string operations. Even `ilwasm` [9] has not implemented proper generation of arrays and prints. Time is needed for the development of WebAssembly to mature. Even then, the speed of the development will only result in changes to the formal specification.

5 Conclusions

At the end of the project, we have come to know that building a compiler from scratch is no small feat. We have to devote our time in designing the architecture of the compiler from parsing to code generation, while maintaining the correctness of the syntax and semantics of the generated code. Even though it is a difficult process especially when dealing with a low-level programming language that has certain limitations over a high-level programming language, we have learned much from this project on what defines a computer program. We assert that the study of compiler design is crucial for programmers to understand the tools that they are using to build a computer program.

This project serves as an excellent example for the community at large to refer to a compiler that generates WebAssembly. There is still much work to be done if we were to extend GoLite to Go instead, but we hope to see that more such compilers exist so as to allow support for more programming languages to be compiled to WebAssembly code. This in turn will move the development of WebAssembly forward.

6 Contributions

Scanner Alexandre laid the ground work for defining the tokens. Cheuk Chuen and Alexandre defined the regular expressions for the literals.

Parser Cheuk Chuen laid the ground work for defining the grammar and AST. Alexandre made important changes and thorough bug checks.

Pretty printer Stefan worked extensively on the pretty printer.

Weeder Alexandre wrote the weeder.

AST Alexandre modified the AST so as to add annotations.

Symbol table Cheuk Chuen wrote the symbol table. Alexandre made changes according to the modified type checker.

Type checker Stefan laid the ground work for the type checker. Alexandre made changes according to the modified AST.

Pretty printer for types Stefan modified the pretty printer to work for the new AST and to print the types of typed expressions.

Testing Alexandre wrote shell scripts to automate testing of the existing valid and invalid programs. Cheuk Chuen wrote the majority of the tests for type checking.

Code generation All three of us contribute to various parts of the code generation.

7 References

1. https://ocaml.org/
2. http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html
3. https://realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html
4. http://pauillac.inria.fr/~fpottier/menhir/manual.pdf
5. https://webassembly.github.io/
6. https://github.com/WebAssembly/spec/blob/master/ml-proto/README.md
7. https://github.com/WebAssembly/design/blob/master/AstSemantics.md
8. https://github.com/kripken/emscripten
9. https://github.com/WebAssembly/ilwasm
10. https://github.com/WebAssembly/sexpr-wasm-prototype
11. https://github.com/WebAssembly/binaryen
12. https://github.com/astlouisf/binaryen