Design Document

Milestone 1

COMP 520: Compiler design

Maxence Frenette (260 685 124)
Maxime Plante (260 685 695)
Mathieu Lapointe (260 685 906)

Feb. 27, 2018

Scanner - An OCamllex Love Story

Matching runes

Problem: How to match rune values
Solution: Create a regex to match a single string character (including escape characters) that can be reused to match a series of runes (a string) and a single rune. The string regex has an additional `\″` character.

Multiline comments

Problem: Match multiline comments in the same way as golite specs (for example, `/* /* */` is valid, but not `/* /* */ */`
Solution: Create a regex that starts with a /* and ends with */ and that is never allowed to match /* in between the two. To do so, the regex is allowed to match any character but *, then

Semicolon injection

Problem: The scanner must add semicolons at the end of lines according to the golgang specification. If there is a comment at the end of a line, the comment must be ignored.
Solution: We create global variable `possible_semicolon` that is true if the last matched character by the scanner is one where there should be a semicolon added if a end of line is encountered (example: an identifier). If a newline is encountered while `possible_semicolon = true` then we manipulate the scanner buffer to inject a semicolon token in the output[1]. Otherwise, if any other token then a newline is encountered, the `possible_semicolon` variable is set to false and no semicolon is injected.
Single line comments (//) problem: After implementing this solution, we encountered a problem with comments. For example, a semicolon should be inserted at the end of a line even though there is a comment at the end of the line.
Single line comments solution: This is where the `possible_semicolon` is very handy. Thanks to this variable, we are not forced to insert a semicolon directly after detecting that it should be inserted. Indeed, we can match on a comment, ignore it, and then insert the semicolon if `possible_semicolon` even if the last match before a new line is a comment.
Multiline comments (//) problem**: Another problem came from multiline comments. For example, if a line ends by a starting multiline comment, but the multiline comment ends on a new line. The scanner needs to detect that a semicolon should be inserted.
Multiline comments solution: Almost same as single line comments, but the scanner needs to check if the matched multiline comment contains at least one new line. If it does and the `possible_semicolon` is true then a semicolon token should be inserted in the scanner output.

---

[1] For more information on buffer manipulation, see
https://realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html

Line counting

Problem: Doing line counting while scanning is usually pretty simple (increment the line number on new line characters). However, since we have a regex matching multiline comments, some new lines inside comments may get lost during scanning.
Solution: When matching a multiline comments, we count the number of newline character occurrences in the matched string and increment the line number accordingly.

Hexadecimal and Octal values

Problem: What should the scanner do when encountering an hex/octal values? Should it store is as an hex/octal or should it directly convert it to integer values?
Solution: After analysis of the languages grammar and typing, we determined that the fact that a value was defined in hex/octal does not affect the behavior of the value. Thus, we decided to convert them to int value tokens during the scanning.

Parser - A Menhir Love Story

AST

Problem: One of the main reason we chose OCaml over another language such as C is how convenient it is to build and analyze tree structures. Indeed, algebraic data types offer an easy way to define our AST that only models valid states. The problem was to build a good structure that is precise enough so that a good portion of invalid programs can't even be represented using our tree structure.
Solution: We decided to build our AST structure before starting the parser so that we'd already have a clear idea of how the parser would look like. As such, we spent quite a lot of time reading the specifications and modifying our AST to be as precise as possible but general enough to support all the edge cases that could arise.

Distributed var keyword

Problem: The first problem we encountered was how to define parsing rules to support the "distributed" var notation for variable declarations.
Solution: We overcame this issue by defining a top level rule that contains the **var** keyword and is followed by either one declaration (which can be several formats) or a list of such declarations surrounded by parenthesis.

Simple statements

Problem: Another issue we had during parsing was that we realized a bit late that we should break down the statements into 2 categories: simple statements and statements. A simple statement is a statement but the converse is not true. This decision came from the fact that the

Go documentation makes the same distinction which means that if we don't, we will have to write several rules to make sure we handle all sorts of special cases.
Solution: This was achieved by re-defining our AST so that we too have 2 different kind of statements. Of course we also had to change some parsing rules but this was not a problem.

AST - A Functional Programming Love Story

Types

Problem: We wanted a good way to represent different types in GoLite that is statically typed.
Solution: We created the `typesDef` discriminated union that recursively represents a type definition. We separated the slice and array types to make sure no invalid type was permitted in the AST. We also created the `typesRef` discriminated union that defines the reference to an existing type. It is different from `typesDef` because it can't represent and anonymous struct. This made sure we can't support anonymous struct (they are invalid in GoLite).

Variable Reference

Problem: How can we represent nested variable reference such as `myvar.array[2].element`?
Solution: We created a `kind` node that represents a list of `kind_elem` each representing an identifier (`myvar`) or an array access (`array[4]`). In the code, each `kind_elem` is separated by a point (`myvar.array[4]`). `kind_elem` cannot represent an array access without an index (that's why we cannot reuse `typesRef` for `kind_elem`).

Else If

Problem: Is there a way to handle `else if` in a simple way?
Solution: We decided to make represent `else if` in our AST as a simple `else` with an `if` statement in its body. This reduces the complexity of the AST since we only had to support if() {} [else {}] statements.

Function Arguments

Problem: How to represent correctly `fun(a, b int)`?
Solution: We decided to take care of this in our parser step. The parser will take the type and add it to every argument it belongs to (the original function becomes `fun(a int, b int)` in the AST. This helps simplify the AST and will make later phase of the compiler easier to implement.

Weed - A Justin Trudeau Love Story

Blank identifiers

Problem: We need to parse the whole AST to look for '_', this turned out to be harder than expected because many types for the AST are mutually recursives. Furthermore, because we have so many constructs to cover, it becomes hard to figure out a way to parse the tree every statement and expression in three without forgetting something.
Solution: We broke down the problem into smaller functions, a function that find blank identifiers in statements, one finds them in expressions and so on. We then built a function that start at the root of the AST and call these smaller functions recursively.

Looking for continue and break statements

Problem: An *if* node in our AST stores its body statements in a list, such that those statements do not have a reference to their parent node. This was a problem in the weeding phase because we wanted to know if a break statement node was a child either a for loop or switch statement, but nothing else. But since it may not be a direct child of a loop or switch, this means that we should be able to start from a break statement and go up in the AST to know if it's the child of a for loop or switch statement. But without any reference to its parent node, we were forced to use a top-down approach.
Solution: We overcame this problem by using a recursive function where one of its arguments is whether it has encountered a *for* node or *switch* node in its path. Once (if ever) we encounter a *break* or *continue* statements, all we have to do is check if a *for* node or *switch* node was found on the current path.

Pretty Printing - An OCaml Love Story

Distributed Declarations

Problem: How to handle distributed declarations?
Solution: We decided to separate the distributed declarations into several single line declarations. This was making the printing easier while keeping the program valid.

Else Ifs

Problem: Is there a way to handle `else if` in a simple way?
Solution: We decided to make represent `else if` in our AST as a simple `else` with an `if` statement in its body. This made pretty printing easier since we only had to support if() {} [else {}] statements.

Tooling - A Javascript Love Story

OCaml

OCaml is arguably the best language for writing a compiler. It isn't by chance that Rust and Go, two recently developed, modern programming languages both started out with an OCaml compiler. OCaml is garbage-collected which eliminates the need for tedious memory management that is needed in C. It is also memory-safe which prevents any segfault from happening. Finally, the biggest advantage of OCaml is the language support for statically types discriminated unions and pattern matching. The support of these features allows to expressively define and manipulate our AST and their statically-typed nature prevents many mistakes akin to forgetting a 'case' in a 'switch' statement in C. All in all, these features allowed us to develop faster, create less bugs and have less lines of code to maintain overall. The only downside for this was a minimal decrease in performance.

Javascript

When we started the project, it was obvious that we needed a good scripting language to define our various tests. We started with a python script that tested the compiler on various files in a fixed directory structure. The first problem we encountered was that we wanted to have a more flexible directory structure. We didn't want to put all our parsing tests in the same /parsing/valid and /parsing/invalid folders, for instance. We also found it nice to have a language with a good unit-testing framework and test runner. In this regards, very few come close to Facebook's Jest. Finally, Maxence is very comfortable with the language.

The Test Script

All in all, javascript allowed us to easily implement nested folders for our tests. We also have a way to programmatically ignore certain test files. The program automatically switches between expecting valid and invalid programs according to the folder names. About 10% of the repo's code is javascript and none of it superfluous.

Jest

Running all tests cases at once is great, but very inefficient when a project includes a couple hundred tests. Jest brings these kinds of features to the table and many more. With jest, we are able to ignore certain tests with command line options, rerun only failed tests, see a progress bar when running lots of tests, split the tests in test suites and see a nice summary of tests that ran. This greatly accelerated the development of the project by allowing Maxime and Mathieu to not lose time with testing.

Lots of Tests

This flexible test setup allowed us to easily create, maintain and organize more than 60 tests. With just these, we were able to find one potential bug in the reference compiler and ~10 bugs in our own compiler. We also use tests from public past COMP 520 GoLite project. We simply added them as git submodules and configured our test script to run their tests and to expect the proper output for the proper commands. Right now, we have ~400 tests that we use to test for regression in our scanning and parsing phases. If we enabled more tests from those repositories, we would have a potential ~4000 tests that we could use. This allowed us to find 4-5 potential bugs in the reference compiler including a segfault that looks to be in the compiler's weeding phase. On our compiler, it allowed us to find dozens of bugs that might not have been covered by our homemade test suite.

Group Organization - A Bromance Story

Since we knew that the testing was going to be a really important part of this project, we decided that one team member should be on testing full-time. Meanwhile, the two other members were going to work on the scanner/parser/weeder/pretty printer and solve the bugs reported by the team member who is testing. This ended up being an accurate separation of tasks since everybody worked equally on this release.

Tasks

- Maxence
    - Setup of an automated testing tool
    - Creation of valid and invalid tests
    - Aggregation of tests coming from past projects found on GitHub
        - Removal of typecheck and symbol table related tests (for now)
    - Reporting bugs found by tests
    - Reporting bugs to the reference compiler
    - Validating repository structure for submission
- Maxime
    - Creation of the initial OCaml repository
    - Contributed to the scanner
    - Contributed to the parser
    - Creation of the pretty printer
- Mathieu
    - Contributed to the scanner
    - Contributed to the parser
    - Creation of the weeder