Compiler Paper - Moises Andrade

1) Lexing and Parsing

The lexing and parsing phases are implemented with the help of `ANTLR` tools. The constructs created by the tool are used in own-implemented code to parse a golite program.

1.1) ANTLR

In a nutshell, the tool auto-generates code in the language the compiler is being implemented (in our case, `Go`) for parsing and lexing the language we are creating a compiler for (the 'target' language, `Golite`). For that, we to provide files containing the token patterns and the grammar for the target language.

The file `GoliteLexer.g4` contains the pattern for tokens in `Golite`.

The file `GoliteParser.g4` specifies the grammar for `Golite`.

The file `generate.sh` runs `ANTLR`, which generates the files `golite_lexer.go` `golite_parser.go`, `golite_base_listener.go` and `goliteparser_listerner.go`. These files contains a couple of constructs

1.2) Lexing

Main methods and types for the lexing phase are in `./lexer/lexer.go`.

The file contains wrappers for methods and types created by `ANTLR`. The `lexerWrapper` *struct* gather the main data structures need for lexing. The `Lexer` *interface* expose to public the main methods needed for lexing.

The `FillTokenStream()` method uses `ANTLR` auto-generated lexer to read all the tokens in a `golite` file and store them in the `lexerWrapper.stream` data structure. The `PrintAllTokenRules()` method prints to the screen the text content of the tokens.

For errors in the lexing phase, `SyntaxError(...)` overrides the error listener method created by ANTLR and store errors in the the `lexerWrapper.errors` slice. Errors are represented as `structs` that the line, column, and mesage for each error. The `GetErrors()` method returns the slice.

1.3) Parsing

Main methods and types for the parsing phase are in `./parser/parser.go` and in the `ast` folder.

`parser.go` contains wrappers for methods and types created by `ANTLR`. The `parserWrapper` *struct* gather the main data structures used in parsing. The `Parser` *interface* expose to public the main methods for parsing.

The `Parse()` method in `parser.go` builds the AST of a `golite` program. Building the AST involves a couple of steps, summarized below:

- `Parse()` calls the `Walk()` method created by `ANTLR`, which, in broad terms:
  - (i) build a tree for the program using the nodes defined in the `.g4` files;
  - (ii) traverse the tree from "up to down" and after that
  - (iii) call from the exit methods for each node from the "bottom to up", returning `ANTLR context` structs that represents the nodes from the tree.
- The ANTLR exit methods are overridden by our own methods defined in the `parser.go` file, such as `ExitAssignment(...)`
- The exit methods uses the ANTLR contexts to build the structs in the `ast` directory representing each node of the final AST.
- That is, after `Walk()` build parts of the tree and the respective ATNLR context, it start calling the exit methods we defined that in turn build our own `structs` for the nodes in the AST.
- All nodes are stored in the `parserWrapper.nodes` map, so the next exit method can use the results created by the previous one.

Finally, errors that occurred during the parsing phase are stored in the `parserWrapper.error` slice. Each item in the slice is a struct containing the line, column and a message and the phase in which the error ocurred. The errors can be accessed through the `GetErrors` method.

1.4) AST: structs and printing

- The structs in `ast` are the final nodes of the AST and concrete representations of the components defined in the `GoliteParser.g4` grammar file.
  - For example, the `function` struct contains a slice of `declarations`, `statements` (which are other `ast` structs) as well as fields for the `id` and `returnType`
- The structs are further divide into two "categories" by the interfaces `Statement` and `Expression` with their own methods. The list of nodes that implement each interface are in the `ast.go` file.
- After the whole tree is traversed "from the bottom to up" as described in the previous section, a `Program` struct will be created. This struct contains slices to of `types`, `declarations` and `function` structs, which by themselves contains slices for other nodes and so on.
- Each struct contains a `string()` method. The printing of the AST is implemented using the visitor's pattern as follows:
  - The `Program.String()` method is called. This will print any Program-specific string and call the `String()` methods of each `type`, `declaration` and `function` contained in the `Program` struct.
  - These will return any node-specific string and call the methods of the nodes within it. This process is repeated until all nodes are visited in a DFS manner, that is, from "the top of the tree to the bottom".
  - As a result, a string representing the AST/source is returned by the `Program` node.

2) Semantic Analysis

Semantic analysis is implemented via the visitor's pattern using methods and structs in the `ast`, `types` and `symboltable` packages.

In broad terms, sintactic analysis proceeds as follows:

1. In the parsing phase, the user-defined types are populated in the `Program.UserTypes` (more below)
2. A first pass through the AST is done to build the symbol tables
3. A second pass through the AST is done for type checking

Below more details.

2.1) Types

- Each basic `golite` type is represented as a **single** struct, initialized in the `init()` method of `types.go`.
  - For example, the concrete representation of a "boolean" is the `BoolTySig` created at initialization. Any boolean found in the source code is mapped to this variable.
  - This allows to compare types through the addresses of these structs
- Each *user-defined* type is mapped to a unique `types.StrucType` struct that implements the `types.StructType` interface.
  - These structs are created in the **parsing** phase. In the exit method of the type declarations, the `Program.UserTypes` map is populated with the user-defined types that are found by the parser.

2.2) Symbol tables and First pass

The program's symbol tables are divided in three categories: `Globals`, `Functions` and `Structs`. These are concrete representations of the `SymbolTable` generic struct. They hold useful information about the global variables, functions and struct definitions of a program, respectively.

The symbol tables are populated in a first pass through the AST as follows:

- `Program.BuildSyumbolTable(...)` is called. This calls the `BuildSymbolTable` methods of `declarations`, `functions` and `typedecl`
- During the calls to `BuildSymbolTable`, any syntactic errors found are appended to an `errors` slice, which is passed among the method calls.
  - For instance, the `function` call to it's build symbol table will populate the errors slice if it founds a duplicate declaration of a local variable name or parameter.
- After all methods return:
  - The `Globals` symbol table will contain info about global variables relevant for syntactic analysis
  - The `Functions` ST will contain info relevant about function declarations
  - The `Structs` ST will contain info relevant about user-defined types
  - All symbol tables are stored in the `Program.SymbolTables` field.

2.3) Stactic type checking

Stactic type checking is realized by the `TypeCheck` methods of each `ast` node using the visitor's pattern.

- The method for each node looks for syntactic errors that can occur at the level of the node and calls the `TypeCheck` method of it's subnodes.
- A slice of `errors` (the same from the first phase) is passed around methods and populated with a string describing an error, if any.
- For example:
  - the method in `binary` checks if logic comparisons between variables have consistent types; any error is appended to `errors`
  - the method in `assignment` checks if a integer defined variable is being assigned to a struct;
  - the method in `function` looks for conflicting names of functions to other types, functions and variables; etc

Therefore, static type checking proceeds as follows:

- `Program.TypeCheck(...)` is called.
- This will do any Program-lelvel type checking and call the type check methods of `declaration`, `typedecl` and `function` which will check for node-level errors and call `TypeCheck` of subnodes.
- After the whole AST is traversed by the `TypeCheck` calls in a DFS manner, the `errors` will be populated with all syntactic errors and returned by the `Program` node.

During sintactic analaysis, the SymbolTables are accessed for duplicate declarations, retrieval of types, etc. This is done by passing to the `TypeCheck` methods the `st.funcEntry` of the callee function and all symbol tables built in the first phase.

2.4) Control flow checking

The check that all paths have a return is done in the `TypeCheck` method for the `function` node, which calls `HasReturn()` method of all *statement* nodes using a mix of the visitor's pattern and recursive logic.

Here is the relevant block of code:

```
var b bool
warnings := make([]string, 0)
hasReturn := false
for _, stmt := range f.statements {
    warnings, b = stmt.HasReturn(warnings)
    hasReturn = hasReturn || b
    // After a return is reached, all following statements are unreachable
    if hasReturn {
        warnings = append(warnings, SemErr(stmt.GetToken(), "Unreachable
code."))
    }
}

// If function doesn't has return for all control paths && is not void, error
if !hasReturn && funcEntry.RetTy != types.VoidTySig {
    errors = append(errors, SemErr(f.token, "Function "+f.id+" missing return
statement."))
}
f.HasReturn = hasReturn // this is used in the LLVM phase
```

Notice: I do not characterize dead code (i.e., code that comes after a function has fully returned) as syntactic error; it is appended to a `warnings` slice though.

More specifically:

- Each `statement` node implements the `HasReturn()` method, which returns:
  - a slice of strings (placeholder for warnings of dead code)
  - a **boolean = true if the statement has a return and false otherwise**
- Regarding the boolean:
  - `ret` always return true.
  - `conditional` return true if (i) has **both** `if` and `else` branches **AND** both have a return or path that leads to a return.
    - That is, returns `true` only when the if-else is "closed" / guaranteed to return.
    - Note: `conditional` only with one branch always return false, since `if` branch may not be taken.
  - `block` returns true if at least one statement returns true.
  - All other statements return `false`

How this works:

- A `block` statement starts with `hasReturn=false`. And then call `hasReturn` for it's statements.
  - We update `hasReturn` by making `hasReturn = hasReturn || statement.HasReturn()`
  - This means that once one of the flows **completely returns**, that block does not have a dead end (but may have dead code)

    - Notice `statement.HasReturn()` will only return true if effectively a `return` statement is found OR a completely "closed" if-else flow is found.
        - "closed" = **both** the TRUE branch and the ELSE branch have a return statement. Emphasis on the **both**; if only one branch ⇒ no return
- In `conditional`, `hasReturn` starts as true. It is updated by doing `hasReturn && block.HasReturn()` for the blocks of **both** branches (notice it is an `and`, not `or`).
    - That is, we call the `block.HasReturn()` method, which in turn returns true only if there is a flow that completely returns.
    - If no such flow is found, `conditional` returns false, closing the recursion.
- Finally, the `function` node similar to `block` starts with `hasReturn=false` and update it by doing `hasReturn || statement.HasReturn()`. Hence if a flow **completely returns** the function has a return. Then it checks if the function is `void` to decide on a semantic error or not (for instance, `hasReturn` can be false just because the function has nothing to return to start with)

Obs: if a program has `return` for functions that does not specific a return a semantic error is also thrown. This check occurs in the `ret.TypeCheck()` method and is separate from the control flow check.

---

3) LLVM

The translation of source code to `LLVM` uses the `ast` package; the `types` package (for type translation to LLVM); and the symbol tables and `hasReturn` from the syntactic phase. It also introduces the `ir` package.

The translation occurs in two phases:

- In a first pass, the AST is converted into a group of LLVM instructions and stored in the `function.cfg` field (a basic block).
- In the second pass, the `llvm` code is printed using the `function.cfg` block of each function
- In both phases, the visitor's pattern and recursive logic is used to do a DFS in the tree / basic blocks (more below)

3.1) Main elements of the `ir` package

- the `instruction` interface represents an LLVM instruction. Each file in the package contains a `struct` that represents an LLVM instruction and implements the interface.

For the files that not map directly to LLVM instructions:

- The `Operand` interface represents `llvm` registers and literals, each implemented by a specific struct (Global registers, Fstr registers, etc)
- A `BasicBlock` struct holds a slice of instructions and pointers to other `BasicBlock`s. These are created throughout the conversion and used to print the `llvm` code.
- The `eofInstructions` struct holds instructions that go in the end of the file in the LLVM and ARM representations.

    - Example: `declare i8* @malloc(i32)`; creation of register for the `printf` statements.
    - There is single a global instance of this struct that is populated throughout the translations by the proper node/instruction. Ex: the structs responsible for `printf` fills the `eofInstructions` with instructions to create registers for `printf` statements.
- The `comment` struct is used to print comments in the `llvm` file.

3.2) First phase: creation of instructions and CFG

For the first phase the visitor's pattern is used through the `ToLLVM` method of each AST node and a recursive call in the basic blocks. The process is as follows:

- The `Program.ToLLVM(..)` method is called. This will create a slice of basic blocks and call the `Program.ToLLVM(..)` of the `typeDecl`, `declaration` and `function` nodes
- The `ToLLVM` of `typedecl` and `declaration` populate the symbol tables with information relevant for the `LLVM` conversion.
    - Example: registers for the global variables; registers for nil values of a struct; etc
- The `ToLLVM` of `function` nodes take the slice of basic block as inputs and:
    - add an entry block for the function
    - create function-specific `llvm` instructions and add to the entry block. Example: allocation of memory to parameters.
    - Calls the `ToLLVM(..)` method of all it's statements passing the last basic block of the CFG (see below)
- A `statement.ToLLVM` method takes a basic block, do a DFS and returns a basic block filled with new instructions. Specifically, the method:
    - Fills the block with node-specific llvm instructions
    - Call the `ToLLVM` methods of the sub-nodes, which fills the block with node-specific instructions and call `ToLLVM` of sub-nodes and so on.
- For the `conditional` and `loop` statements, besides filling the original basic block, new blocks are and linked creating the CFG of a function
    - In `conditional`, the `hasReturn` of the semantic analysis phase is used to decide on the creation or not of an exit block. If both branch return ⇒ no exit block. This avoids creation of empty labels.

3.3) Second phase: printing

For the printing of LLVM code, a DFS is performed using the visitor's pattern in the AST nodes and basic blocks:

- The `Program.PrintLLVM(..)` method is called. This writes any program-level `llvm` instruction to a buffer (example: the target triple) and calls the `PrintLLVM` of `declarations`, `typedecl` and `functions`
- The `function` method writes any function-level instruction (e.g: registers for parameters) and calls the `BasicBlock.PrintLLVM(..)` of it's CFG created in the first phase
- The `PrintLLVM` of the basic block prints any BasicBlock-level instruction (e.g.: it's label), `PrintLLVM` of all it's instructions and recursively calls the `PrintLLVM` of all it's BasicBlock successors.

- After all calls, `Program` calls the string method `eofInstructions` to write the instructions that goes at the end of LLVM files (such as registers for printfs, etc) and returns a string containing the `llvm` code.

---

4) ARM

The ARM implementation using the visitor's pattern through the `TranslateToAssembly` methods of each `llvm` instruction of the `ir` package. The `codegen` package contains methods for writing code and counters the ARM registers.

A stack-based approach was used.

The process is as follows:

- `program.TranslateToAssembly()` is called
- This writes any program-level arm instruction (eg: code for the architechture, global variables, etc) and calls the `TranslateToAssembly()` method of each function.
- `Function` allocates space to the stack and populate the symbol table with locations for all registers and calls the `TranslateToAssembly()` method of it's CFG basic block.
  - Obs: during the LLVM phase, the function symbol table's will be populated with temporary registers used in the LLVM translation. But their locations in the stack are determined here.
- The `BasicBlock.TranslateToAssembly()` prints any block-level arm instruction, calls the `TranslateToAssembly` method of all it's instructions and recursively call the `TranslateToAssembly` of all it's BasicBlock successors.
- The `TranslateToAssembly` method of instructions are direct translation of an LLVM instruction to arm. In some cases, more than one ARM instruction is created by an LLVM instruction (eg: if a global have to be accessd in an `add` instruction)
- All instructions are returned as strings. I.e., there is no indirect structs as in the previous phases. `TranslateToAssembly` methods passes buffers among themselves containing previously written instructions.

5) Answers to specific questions

Lexing and Parsing: Describe how your compiler lexes and parses a Golite program?

- Answered in Section 1

Static Semantics: Discuss how your compiler implements static type checking and how it checks to ensure all control-flow paths have a return statement.

- Answered in Section 2

LLVM Intermediate Representation: Discuss the structure used to implement the LLVM instructions and how they are store after converting the AST nodes to LLVM instructions.

- Answered in section 3

Optimizations: State the specific optimization implemented and how you implemented the optimization in your code. Discuss the representation of the program as a set of control flow graphs with LLVM instructions. If you did not implement an optimization then discuss how you would modify your compiler to include one. Talk specifically about what packages you would need to add, how your LLVM representation would need to change, etc.

I did not implement optimizations. But a direct improvement would be a graph-coloring of registers to remove the unnecessary loads and stores that comes with the stack-based implementation.

I believe it would be relatively direct to create the sorted super block given my LLVM representation. In my implementation, `function` contains a single `cfg` block that points to the successors which are accessed via a DFS (as in the vanilla implementation of the topological sort)

More packages and methods would be needed for the construction of the graph of registers and the coloring.

- Each `Instruction` struct for an LLVM instruction would need to have fields and methods to compute the live ranges.
- I would add a new package `graph` containing a struct definition for the graph of a function. It would contain a `node` struct to represent the virtual registers created in the LLVM phase. I would also have a `stack` struct for the final phase of the algorithm
- Populating the graph and computing live ranges could be done during the translation to LLVM.
  - Nodes could be created in the `ToLLVM` methods of each instruction, similar to how I populate the function stack in the current implementation.
  - For the live ranges; the start could would be set when the register is created and the end could be incremented whenever the register is used during the translation.
  - To fill the adjacencies, I am not sure but I believe I would need another pass using the populated graph and live ranges to connect the nodes.
- After this, I could use the populated `graph` and the `stack` struct to execute the algorithm in step 4 of the lecture.
- In the end I would have the same `StackRegisters` map of registers in the `FuncEntry` for that function in the symbol table, but much smaller. And this could be used in the same way I use in the stack-based approach.

Code Generation and Register Allocation: Discuss any interesting aspects of your translation from LLVM to assembly.

Discussed in section 4.

Misc.: Discuss other aspects of your compiler that you find interesting. Be proud that you implemented you very first compiler and talk about your struggles and what you learned in the process.

1. I believe the main struggle (and learning) from the process was anticipating what decisions in terms of software design I should take at step T so that:
    - They are specific enough to implement the required features of step T
    - They are general enough so that implementation of new features in phases T+j can be relatively swift
- In some points, I had to refactor code a couple of times for taking bad decisions.
- In other points, I took more time to write something more general initially, but was happy when collected the benefits in following phases.
- In all situations, I learned that planning was good, but sometimes you have so few knowledge of what you will have to implement that you just have to get started. Then re-iterate and refactor.
- After seeing how the whole process works, I would refactor many things in the code for even for the phases that are working fully
2. Another one is the challenge of "remembering" code when it increases in complexity and one has to task switch.
- This seems a bit specific, but as I stopped working in the compiler to deal with other assignments, retaking the work on it took some time. Specially because of the many recursive/nested structures.
- I learned along the way make specific notes so that I could restart work faster
3. Turning abstract representations into concrete ones. This was true in may situations: when turning the source code into "nodes"; creating "blocks" and the CFG; etc.
4. A personal challenge (and source of proudness for myself): I find/found particularly difficult to deal with recursion. In this course, I had to overcome it so many times. And I was a bit proud of myself when I could come with the recursive solution to the control flow check (which I found very elegant)
5. Finally, I would like to have been able to fully implement the ARM phase and optimizations. I found it to be more difficult to go from source code to IR and machine code. Nonetheless, I believe I learned all the necessary tools to do so (which I plan to).