McGill University

COMP 520 - Compiler Design

Winter 2018

GoLite

Milestone 1 - Group 02

Name: Jérémie Poisson
McGill ID: 260627104
E-Mail: jeremie.poisson@mail.mcgill.ca

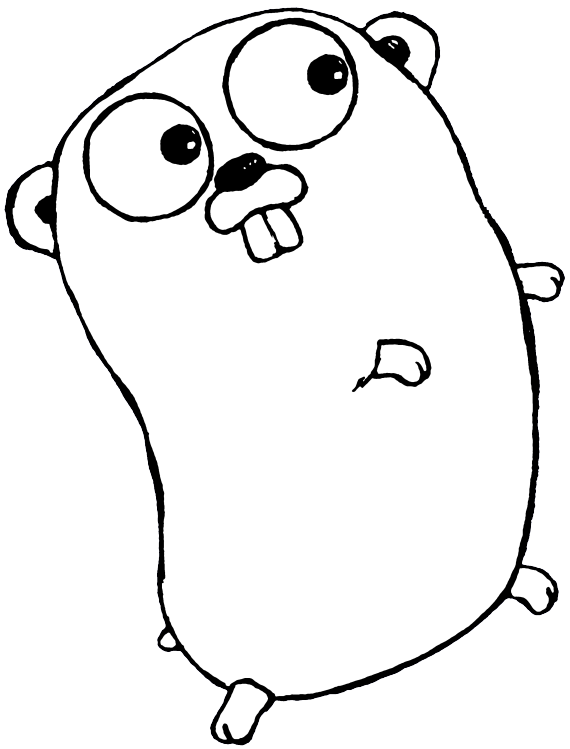
Name: Amir El Bawab
McGill ID: 260645260
E-Mail: amir.elbawab@mail.mcgill.ca

February 28, 2018

Contents

1 Introduction

In this report we discuss GoLite, a programming language that is a subset of Golang. GoLite language covers the necessary syntax to build a meaningful program. The process of building GoLite project is composed of several phases: Lexical analyzer, Syntax analyzer, Semantic analyzer including type checking, and finally Code generation. For the first milestone, only the Lexical and Syntax analyzers were implemented, the last two phases will be developed in future deliverables.



2 Lexical Analyzer

2.1 Choice of a scanner

Several open source libraries are available for scanning text input and converting it into tokens. In this project we use Flex for that purpose because it is written in C/C++, the programming language of choice for our team.

2.2 Flex

Flex library uses regular expressions and deterministic finite automata (DFA) to identify sequence of characters and converting them into their corresponding tokens. Both regular expression and DFA, in addition to optional semicolon feature are discussed in the following sections.

2.3 Regular Expressions

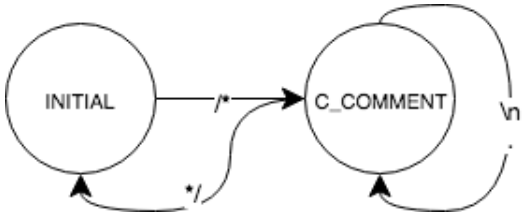
The majority of GoLite syntax is defined using regular expressions. Reserved keywords, operators, literals, white-spaces, new lines, inline comments are all examples of syntax presented to Flex using regular expressions. The only exception where regular expression was not used, but could have been used, are block comments and strings. The latter were chosen to be implemented using a DFA because it is simple to understand and maintain compared their equivalent regular expression representation.

2.4 Deterministic Finite Automata

Compiling Flex files converts regular expressions into a DFA. However it is also possible to construct a DFA directly in a Flex file. In fact, building DFA manually turned out to be useful in our project for identifying block comments and strings.

2.4.1 Block Comment DFA

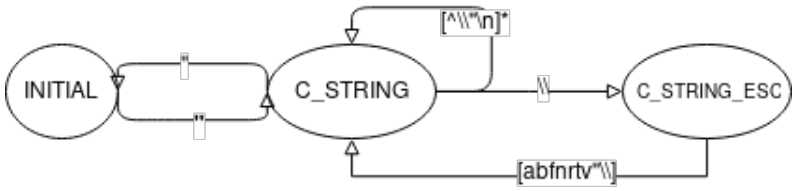
```
"/*"              { BEGIN(C_COMMENT); /*Go to C_COMMENT state*/ }
<C_COMMENT>.      { /* Comment can contain any char */ }
<C_COMMENT>\n     { /* Comment can contains new lines */ }
<C_COMMENT>"*/"   { BEGIN(INITIAL); /* Go back to initial state */ }
```



2.4.2 String DFA

```
<INITIAL>\"  {  /*...*/   BEGIN(C_STRING); }
<C_STRING>[^\\"\n]*  {  /*...*/  }
<C_STRING>\\  {  BEGIN(C_STRING_ESC); }
<C_STRING>\"  {  /*...*/  BEGIN(INITIAL); }
<C_STRING>\n  {  /*...*/  }
<C_STRING_ESC>[abfnrtv"\\]  {  /*...*/  BEGIN(C_STRING); }
<C_STRING_ESC>\n  {/*...*/  }
<C_STRING_ESC>.  {  /*...*/  }
```

NOTE: For full DFA code please refer to the project files



2.5 Optional Semicolon

GoLite allows users to optionally apply semicolons at the end of their statements. Archiving this behavior requires manipulation of the list of tokens generated by Flex where semicolon are auto-injected before new lines and after some predefined set of characters. The latter are determined by Golang under the first rule of the optional semicolon section: https://golang.org/ref/spec#Semicolons. Several methods can be used for injecting tokens follow a set of rules, in this project optional semicolon concept is implemented by keeping track of the last token before a new line. If the last token is part of the predefined characters, then a semicolon is injected. It is important to note that the grammar has to align with those rules as well. In other words, when the parser analyzes the tokens, it would not accept optional semicolons, in fact they are mandatory.

3 Syntax Analyzer

3.1 Choice of a parser

In this project our team decided to use GNU Bison as the GoLite language parser as it works in harmony with Flex scanner.

3.2 Conflicts

Building GoLite grammar is a challenging task. While structuring the grammar productions is difficult, not many conflicts were available, and if appeared they were simple to solve. In Left-to-right Rightmost (LR) parsers, there are two types of conflicts: Shift/Reduce and Reduce/Reduce. Shift/Reduce conflicts were not encountered during the development of the grammar as the language forces the user to use delimiters (e.g. {, (, []), between tokens making it clear for the bottom-up parser what its decision must be. Reduce/Reduce conflicts were more common. The two cases were Reduce/Reduce conflicts appeared were, first when two productions are identical and the parser would not know to which one it should reduce, and second when a production represents a particular derivation of a more generic production.

Example of the first type of conflict is presented below.

```
function_args
    : expressions
    | %empty
    ;

expressions_opt
    : expressions
    | %empty
    ;
```

In this example, clearly both productions are identical, but they were initially developed in parallel as their names were descriptive when used in productions. This conflict was solved by removing *function_args* production and using *expressions_opt* instead where applicable.

Following is an example of the second type type of conflict.

```
simple_statement
    : expressions tEQUAL expressions
    | ...
    | identifiers tDECLARATION expressions
    ;

expressions
    : tCOMMA expression
    | expression
    ;

identifiers
    : tCOMMA tIDENTIFIER
    | tIDENTIFIER
    ;
```

The problem in the above example is that *identifiers* production can be generated from the *expressions* production, the former is a particular derivation of the latter. This ambiguity may result in the grammar generating *expressions tDECLARATION expressions* which cannot be reduced to *simple_statement*. The conflict is less clear to detect and even the parser would not detect it at compile time, in fact the runtime behavior would arbitrary select a production (usually the first one to occur). Unlike the first conflict, deleting a production does not completely resolve the issue. In fact, the solution applied in this project allows the input to form an illegal syntax, and later in the code a weeding pass is performed to verify the correct usage of the language specifications. So in the example above, the *simple_statement* production has been modified as follow:

```
simple_statement
    : expressions tEQUAL expressions
    | ...
    | expressions tDECLARATION expressions
    ;
```

After building the AST, the left expressions of the declaration are checked to make sure they represent identifiers. If not, an error is reported.

3.3 Abstract Syntax Tree

Building up the abstract syntax tree requires us to model the language component with different data structure. In this section we explore how we designed the main classes and how they model the problem of representing a program in a abstract syntax tree form.

- **Program**: represent the whole program at a global level. It is currently designed as a singleton, which makes much sense as GoLite is only concerned with a single program. As per the program specification, a program is only made of a collection of declarable (no statements are allowed at the program-level). For example, a simple program containing a simple main function will only contain a declarable for the main function.

- **Declarable**: represent every declarable statement that can be made within the language. In GoLite, one can declare many things: types, functions, variables, structure. When such statement is parsed, a declarable object is created and is later referenced throughout the program AST, as needed. For instance, a type declaration will be referenced from a function call statement later on in the program as the parsing phase continues.

- **Statement**: represent every general statement within the program that do not declare new structure in memory to later reference in the program. Those constitute everything from variable-expression assignments to if and return statements. A statement can also be a block of statement, that is simply a list of statement bundled toghether —i.e. the code bundled within a if statement is a block of statement. Built-in functions such as $println(...)$ and $print(...)$ are modeled directly as separate statement classes as they are reserved keywords as per the specifications. It is thus impossible for a user to redeclare such functions.

- **Expression**: are a special computational statement which are used to return a certain result. In GoLang, expressions are generally either Unary (acting on a single sub-expression), Binary (acting on two sub-expressions, usually a left hand side and a right hand side) or Primary which represent the base case for the recursive definition of both Unary and Binary expressions. Those comprises the basic arithmetic operations such as addition, multiplication, increment, decrement, etc. A special case of expressions are what we call "Primary" expressions. Those are simply expressions representing other structure declared within the program. As part of primary expressions, one find identifiers, function calls (along with its set of passed arguments), array-like structure access (along with the specified index), literals (such as plain integers, strings, etc). Like the *println*(...) and *print*(...) functions, the *append*(...) function is modeled directly as a separate expression sub classes. This is because it is a reserved keyword that cannot be overridden by users but also returns a value.

A complete and elaborate UML design of all the concrete classes is available as an appendix to this report.

3.4 Pretty Print

The pretty print implementation for this milestone was mostly straightforward. We simply added a pure virtual *toGoLite*(*intident*) method in every **Statement**, **Declarable** and **Program** classes such that every subclasses implement its own pretty print method. For pretty printing the whole program, we simply have to traverse the abstract syntax tree constructed and call the *toGoLite*(*int*) method with the appropriate identation level as parameter.

4 Testing

For the scope of this milestone we designed GoLite programs that tested the ability of our compiler to parse and scan both valid and invalid programs. GoLite is a flexible language and supports a lot of custom types and structures. Our methodology for designing such tests is mainly based with our exprience of developing the scanning and parsing phase of the compiler and the many intricacies of the GoLite language. For both valid and invalid programs, we established about 4-5 programs for each component of the language. About half of those programs are testing basic functionnalities (right syntax, usual use case, etc.) whereas the other half focuses on testing edge cases (unusual syntax, nested structures, etc.). Some of those tests also integrate multiple components toghether and verifies their compatibility as per the official specifications of the GoLite language.

So far, our test set comprises of about 100 invalid programs and 50 valid ones. Our compiler successfully passed the whole test set. We are confident we covered most of both the usual and edge cases of the GoLang syntactic issues.

5 Team

We started off by discussing the main components of the GoLite language and how they would be implemented. Abstracting the language components into an object-oriented paradigm was fairly straightforward at the beginning, but we realized that our design required a major refactoring later as we read the full GoLang reference specifications documentation and compared it to the GoLite specifications sheet.

Amir, being more experienced with compiler design, undertook the grammar part of compiler at first while Jeremie created the initial code implementation. After reviewing our work as a team and adjusting to the major design modifications, we figured it would be better to have only one person working on the implementation at first so that the codebase would be uniformly designed for the next milestones. We think it is much better to have a solid and homogenenous codebase before

adding in new features. It was also much more easy for us to sync up and work efficiently. We decided that Amir should work on building the Abstract Syntax Tree while Jeremie was busy working on writing test cases and on the report. We also carefully synced up every now and then to make sure we both took part in the design decisions and that we both are proficient in writing actual code.

We think this first milestone was a very good start for the GoLite project. We are confident that our compiler will do well when evaluated and is easily maintainable for the subsequent milestones of the GoLite project.

5.1 Unified Modeling Language