Milestone 1: Go Parser

Xavier Denis

February 16, 2015

1 Design Decisions

For this project I opted to use the Haskell. Haskell itself offers many advantages such as it's very power type system which allows for many errors to be caught in a static manner. Additionally, the AST can be easily represented using standard Haskell types. While the type system and core language of Haskell are very powerful and useful, the primary reason for using it is the library ecosystem. Parsec, a monadic parser combinator library provides a set of simple, powerful tools which allow for the lexer and parser to be combined in a legible manner. Parsec also provides utilities to parse entire expression trees very rapidly. The primary downside of using Haskell is the learning curve that is associated with monads and other abstract constructs. While this hindered me at first, I was able to rapidly adapt and pick up more advanced and efficient patterns.

2 Implementation Decisions

The code is split into several modules which encompass the different subjects of the language. Within the top-level 'MGC' (Montreal GoLit Compiler), there are two other modules.

The 'Syntax' module contains a definition of the AST and all related data types. It also handles representation of the AST, that is, pretty-printing. Pretty-printing is handled in the 'MGC.Syntax.Pretty' module, it contains an abstract implementation of a simple pretty-printing typeclass and the concrete implementations for each type of the AST.

The ‘Parser’ module consists of 4 parts. The first, ‘Prim’, contains the basic blocks of the language and many helper methods, it handles parsing of literals, parentheses, braces, white-space and comments. The ‘Expression’ module handles the parsing of all Go expressions. The ‘Type’ module parses all go types including user defined types. Finally, the main ‘Parser’ module contains the actual definition of the language putting together the previous module to form the statments, variable and type declarations that make up go.

There is a single area of concern in the structure of the implementation. The ‘Type’ and ‘Expression’ modules are mutually dependent and required a special trick to allow the Haskell compiler to compile the modules. While this shouldn’t impact future development, it is worth bringing up.

Seperately from the main source code directory is the ‘test’ directory. This contains tests for many (but not all) of the codebase. Some tests were not defined due to time restrictions or simply were found to be unnecessary (not testing useful properties).