Creating a Compiler and Executer in Haksell for Goal, a Simple Programming Language Based on Google Go.

Luke Jackson

April 12, 2015

Contents

Chapter 1

Introduction & Overview

1.1 Introduction To Project

This project focuses on creating a compiler in Haskell for a simple language based on Google's relatively new language, Go. As part of this project I decided to create my own language, Goal, which is a simplified version of Go but contains all of the features I was interested in implementing. I then created a compiler and executor for Goal.

The aim of this project was never too create something that can handle Go exactly as Google designed it. It was to create something that could handle a simpler language of my own design that will get most of its ideas, features and syntax from Go, although it's scope will be much smaller. Thus making the main focus of this project implementing a compiler and executer for a modern programming language. Though I also ended up exploring some aspects of programming language design.

Another important focus of this project is creating a parser. Rather than just using a basic approach to this I chose to use monadic parser combinators.

This document is split into five main sections; Designing Goal, Parsing, Code Generation & Intermediate Representations, Code Execution and Testing. In each of these sections I will go into more detail about how I approached each of these problems in my project.

These next few pages will give a brief introduction to compilers and programming languages, also giving more information about Go and the motivation behind the project. I will also briefly discuss any development methodologies used and give a justification for certain technical decisions.

1.1.1 Introduction to Compilers & Executors

To understand what a compiler is, you need to first understand what a program is and, more importantly, what a programming language is. Programs have become a fundamental part of human existence. From performing simple calculations to creating applications that allow me to create documents such as

this, they exist in every aspect of our life. The simplest definition of what a program is, is to think of a program as a recipe.

You have a list of resources that you need, followed by a series of instructions telling you what to do with these resources. Then, to run your program, you grab all the resources you need and follow the instructions.

Programs that run on computers have to be represented in someway that both humans and eventually computers can understand. Therefore programs are written in programming languages, a good definition for programming languages can be found in [Aho et al.(2007)Aho, Lam, Sethi, and Ullman, p, 1] where they state;

> Programming Languages are notations for describing computations to people and machines ... But, before a program (in this format) can be run it first must be translated into a form which can be executed by a computer.

So this brings us nicely to being able to define what a compiler is, a compiler is something that takes programs written in one programming language and translates it into another programming language. Often taking a program written in a high level language, that humans write code in, and translating them into lower level languages that computers can understand and then execute.

Finally an executer is something that executes code written in a given programming language. You will usually only see executors created to execute low level languages given the smaller, and more precisely defined, instruction set. This highlights the need for compilers to translate high level languages into more easily executable lower level languages.

Looking at these explanations it is clear to see the importance of compilers in the modern world. They allow people to create complicated programs and applications efficiently without having to concern themselves with low level details.

1.1.2 Introduction to Goal

Goal is the language I have created my compiler for. It is a language I have created specifically for this project as a means of exploring compilers. It draws heavy inspiration from Go and uses the same basic syntax where possible. The main features of Goal are the ability to perform function recursion and to create and run concurrent programs.

The idea for Goal came when i was picking features from Go I was interested in compiling then decided it would be a nice idea to bring them together to create a more complete language. In my opinion during the course of this project Goal has evolved from a means to explore compilers and language design into a simple standalone language with it's own uses. I will discuss more about the design and functionality of Goal in chapter 2.

1.1.3 Introduction to Go

Go is an object oriented programming language created by Google relatively recently in 2007. It is used by Google for many different applications, most notably it powers dl.google.com, a service which contains the source for Chrome and The Android SDK. A good place to find out more about the history of Go and it's development is in the documentation section on golang.org.

In many ways Go is similar to C in terms of syntax and that it is also statically typed. There are however some interesting features thrown in, such as how Go handles concurrency and even some newer ideas such as splices, which we will talk about later.

In summary I would say Go (also referred to as golang) is a new language with quite a bit of potential, which can be seen by it's growing popularity. Although it is not revolutionary, it's mixture of simplistic syntax and more exciting features makes it a good language to emulate.

1.2 Motivation

The biggest motivation behind this project was a desire to learn more about compiling and executing modern object oriented programming languages. I was also curious about programming language design and wanted to take the chance to really look at a programming language analytically.

1.2.1 Why Haskell

I chose to implement my compiler using Haskell. Haskell is a purely functional language with strong static typing. Functional languages are often seen as good platforms to use when creating a compiler because some of their features make it easier to handle tree data structures, which can be important during parsing and creating an intermediate representation of language. Also the use of pattern matching and efficiently being able to recurse makes Haskell a good choice for implementing a compiler and a virtual machine.

Another reason I chose Haskell was quite simply that it is a language I enjoy working with, and more importantly for a project like this, would like to learn more about. I was keen with this project to not only create something interesting but also learn more about functional programming. I hoped to use this project to explore some unique functional approaches to some of the problems creating a compiler can throw up. A good example of this is how I handled parsing and my use of Monadic Parser Combinators.

1.2.2 Why Create A New Language

There are so many great programming languages out there with fantastic supporting documentation that would be excellent choices to make compilers for. Which really begs the question, why did I go through the hassle of designing my own language to compile? The answer to this is pretty simple, with a project

like this if I was to create a compiler for C++, for example, I would never have the time to create something that covers all of C++'s functionality within 9 months. Therefore no matter how well I had done with the project it would always feel a little incomplete, but more importantly if someone was to use my compiler they could very well end up trying to use features my compiler didn't support which would be frustrating for me and anyone who wishes to use my compiler in the future.

Instead of having this problem I decided I would create a new language that I could provide supporting documentation for and strictly define what is and isn't possible with in it. Because the focus of this project was not on language design my approach to creating a new programming language was finding a language I liked, then picking key features I was interested in exploring. Before finally adding some extra functionality so that this new language could be useful on its own.

The language I liked the look of was Go and that is where most of the features for my language came from, with the key feature I was interested in exploring being concurrency. I decided to call the language I had created Goal.

1.2.3 Why Base This Language on Go

Go is not a language I was overly familiar with before the start of this project, but as soon as I looked into it I very much liked what I saw.

At the start of this project I was exploring different possibilities of languages I could use as inspiration for Goal. When I came across Go I discovered it had a very clean and easy to understand way of creating and running concurrent process, which was one of the key features I was interested in putting into Goal. From there I began looking into Go's design and discovered not only did it have a wealth of interesting features to look at but also a clean and understandable syntax, which would work nicely with the simplistic easy to use nature I wanted to create in Goal.

The most important factors where defiantly the way Go handle concurrency and channels but also there was an aspect of using this project as a way of familiarizing myself with another language. It also helped that Go came with such a wealth of easy to navigate online resources which was perfect for when I needed to check the exact functionality of certain expressions.

Overall I felt Go provided me with a simple syntax and good implementations of the main features I was interested in implementing. Which made it a good choice to use as the basis of Goal.

1.3 Development Process & Methodologies Used

The main method I used for development was Test Driven Development (TDD). This is a simple approach where you write your test cases first, then write your code so that is passes all your tests. I found this to be quite an appropriate approach due to the iterative nature of my development process.

I initially started work by focusing on being able to compile and execute a small subset of simple features such as if statements, variable assignments and simple arithmetic. Then using TDD I wrote tests for each new piece of functionality I wanted to add, expanding the subset of features I was able to handle. Due to the nature of TDD I was able to do this without worrying about breaking earlier features as I could just ensure all my original tests still passed.

As was mentioned before, I split the implementations of my compiler into three main sections;

- Parsing
- Code Generation & Intermediate Representations
- Execution, using a Stack Based Virtual Machine

While developing my compiler and executer I often found I would implement each feature in two steps; First ensuring I could create a suitable intermediate representation and updating my virtual machine to ensure it could handle the new feature. Then, once this was complete, I would update my parser to handle this new command.

Often when working on multiple features at a time, it would be more efficient to update the code generation and virtual machine first for all the new features. Then update the parser second, rather than adding each new piece of functionality one at a time.

I will go into more detail about the creation and running of my tests in Chapter 5.

Chapter 2

Designing Goal

This chapter deals with how I approached designing a new language and how I went about choosing the features I wished for Goal to contain. I will discuss some reasoning behind why certain features differ from how they are handled in Go and also talk about how I tried to make the language as complete as possible.

I think it is also important to note here that the main focus of this project is not on the design and creation of a new language but rather the implementation of the compiler itself.

2.1 Picking Features

As I mentioned before, rather than designing Goal from scratch I chose to instead choose features from Go I was interested in, then find a way of putting them into Goal. I did not create Goal with a target audience or with potential uses in mind, hence the lack of a specification or market research. Instead I created Goal as a means to let me implement a compiler that could handle a number of different interesting features.

Therefore you will find the features Goal can handle my seem quite varied and not completely complimentary of each other, but I do feel that Goal still has many uses.

As much as Goal was created simply as a tool for creating an interesting compiler, I did also attempt to make it as user friendly as possible. I have created supporting documentation for Goal that has full examples of the syntax it uses and detailed explanations of how to use each of it's features.

A good summary of my approach to creating Goal is that I filled it with features I wanted to explore, then added extra functionality to try and make the language as easy to use, complete and useful as possible.

2.1.1 Syntax

A language's syntax can be said to describe the form that commands and expressions in a language must take [Terry(1997), p. 72]. In the case of programming languages it defines how you must write your code so that it can perform the computations you wish.

The syntax for Goal is pretty simple and almost identical to that of Go, with some minor differences. I decided to follow Go's syntax rules not only for simplicity but also so it was easy to see where Goal got it's functionality from. I felt it would make sense to give it syntax rules close to the language it was emulating.

A key part of Goal's syntax is that each command must end with a semicolon, including if statements, functions and for loops. A more detailed outline and examples of valid Goal syntax can be seen in the accompanying documentation for Goal with examples of syntactically correct Goal [**?**].

2.1.2 Types

2.1.3 Basic Commands

2.1.4 Functions

2.1.5 Concurrency

2.2 Differences From Go

2.3 Possible Uses

Chapter 3

Parsing

Bibliography

[Aho et al.(2007)Aho, Lam, Sethi, and Ullman] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers; Principles, Techniques & Tools*. Pearson, 2nd edition, 2007.

[Terry(1997)] P. D. Terry. *Compilers and Compiler Generators; An Introduction With C++*. Thompson Computer Press, 1st edition, 1997.