Oregon State University
College of Engineering

CS Capstone Technology Review

May 16, 2020

Fall 2019

Code in Sheep's Clothing

Prepared for

Oregon State University

Martin Erwig

Prepared by

Group 41b
Tom and Friends
Johannes Freischuetz

Abstract

The main role of this project is to be able to be able to give students an easy way to learn programming concepts without having to know the complicated syntax of a more complicated programming language. Reaching younger students who don't have as much experience with problem solving has also been challenging in the past. The project should develop a programming language that is able to not only act as a transition into programming but give students something that they are able to enjoy and relate to. This project consists of developing a syntax for this programming language and then implementing it. The main aspects that the author focuses on are implementing built in functions, the parser's concrete syntax, and designing the syntax for the language. The built in functions implementation this will be dependant entirely on the target language, which as stated in the paper should be C++. The concrete syntax can be implemented in either Haskell or C/C++. The concrete syntax is relatively ambiguous to the language and C++ seemed to be the best choice. The main decision for this project are the language design decisions. This will influence most of the other decisions of the project. For this, we will use an abstract syntax tree.

CONTENTS

1 Goal

The main problem with teaching programming to new programming students today is that there is no easy way to introduce students to programming, especially for students who are younger. The goal is for students as early as middle school to be able to understand the topics and be able to begin working on projects independently after short lessons. When attempting to do this with current programming languages, the burden of learning all the required material is often overwhelming and requires too much time to learn. For example, teaching the concepts of C or C++ to a middle schooler so that they can learn basic topics in computer science is not time efficient. There are languages that attempt to make it easier for people who are new to programming to learn, such as Scratch, but the issue with this is that many of the concepts do not transfer well to programming languages that are used in industry.

This project proposes to develop a Domain Specific Language for students to be able to define simple board games that could be played. Students are often introduced to programming through games. To bring these two ideas together the DSL would focus on defining a language for students to be able to define the rules of a board game. This does not necessarily mean that every board game would be definable, but rather a set of core board games must be able to be defined. The constructs used to define these games would allow for a wide variety of custom games that students could also easily define. This would allow the students to be able to relate to the programming that they are doing as well as learning core programming topics without needing to understand abstract concepts. Because of the target audience, one of the main goals of the language is to be able to provide good error message to provide the user with easy to debug syntax errors. These previous restrictions would also allow for better error messages and simplified instructions.

Once defining the Domain Specific Language, a specialized editor would allow for more restrictions or visual editing that is not currently possible in other languages. This will significantly help new students who have no experience. It would allow for faster feedback as well as guide them in making the correct decisions for the design of their projects. One way to imagine this is only showing the student logical paths that they can take from where they are. While this leads to some restrictions of creativity, this is acceptable as the teaching is more important than the outcome.

2 Components

2.1 Built In Components

One of the main components of the programming language is the ability for many board game rules to be built in. The students who are using the language should not be required to be to implement these concepts to be able to make a simple program. These allow the user of the language to not know about the backend representation of the programming language and still be able to easily operate with the language. This also can allow students to develop something that would normally be far outside of their programming skill level. An example of one of these built in construct would be counting the number of elements in a row of a given type. This can be used to implement the win condition for tic tac toe by counting the number in a row to see if it is three. This clearly greatly simplifies the logic required for the user of the language. There will be many different types of built in components that need to be implemented, but they are all high-level concepts like the in a row construct.

2.2 Parser Concrete Syntax

For the language there will need to be a concrete syntax that can convert our DSL to a programming language that can be compiled to assembly. This is an essential part of any parser. The efficiency of this part of the project is not very important, but it must be functional for all edge cases and allow for giving good error messages to the compiler. This concrete syntax will likely be designed based on the language that is chosen for the implementation. There are many parts of the concrete syntax that may be difficult to implemnet without shift reduce conflicts.

2.3 Language Design

Designing the language is one of the most important parts of the project. The project will go through many iterations of the actual implementation of the language. For this reason, it is important to try to design the most important parts of the design first, so that work can start development on other parts of the project. Because this part may also change in the future, it is also important to understand how to design so that the concrete syntax will be able to handle the changes of the language. For this reason, these parts will in many ways be developed together.

3 Technologies

3.1 Built In Components

Implementation language for the built-in components are dependant on the implementation of the parser. It would not make sense to implement them in a different language than the target language as they will likely be used in the parser. The ease of development for these built ins will help decide though on which language to use for the parser. Both C++ and Haskell allow for fast development of the language. Both languages should be able to easily be implemented into the parser. The only advantage is for C++ where it is able to integrate directly into the parser as it can easily be either compiled directly to assembly and used that way or it can be first transpiled to C++ then directly integrated [1]. The main benefit of using C++ over Haskell is that it can be compiled to a binary through another step. For this reason I think the project should use C++. [2].

3.2 Parser Concrete Syntax

The abstract syntax tree design is somewhat unrelated to the language that is chosen for the parser. This will be implemented similarly for most language choices. For both C++ and Haskell the implementations of the abstract syntax tree are almost identical. The most important characteristic for this part of the implementation is that it is easy to change. This means that the symbols should ideally be highly modularized and easy to change at any point during the project. Doing this will allow for almost at will changes to be made to the language design. The scanner will of course still need to accept new symbols, but any changes that are made at the grammar level will be already made.

Both C++ and Haskell have very good support for development of programming languages, but of course they have their tradeoffs. One of the biggest tradeoffs that must be understood first is the amount of support that each language has. C++ has a large community of people that are very dedicated to designing programming languages through flex and bison [1]. Theses are very powerful resources with decades of support. Haskell also has a large community, but one of the main tradeoffs, is that there are programming restrictions that come with Haskell that C++ does not have [2]. While many times these restrictions are good practice it can be difficult to work around them sometimes. Another consideration, for converting to assembly, C++ has amazing support for this, where Haskell is less developed [3]. This means that if the project ever needs to be able to generate standalone executables, it will be much easier in C++. While Haskell is weaker in this sense, it is known for being much faster at developing languages quickly and quickly modifying them which is a very useful aspect for our project. One more consideration is that most of the members in our group have a very strong background in C++ using flex and bison. C++ overall is the more powerful and flexible language which seems to point to this being the best language for this aspect of the project, however for this portion of the project the language is a lower priority than other aspects. It is a higher priority to choose a standard language across the project. For this part specifically however, C++ makes the most sense.

3.3 Language Design

The language design is one of the most important parts of the project. It must be easy for the students to understand as well as being very powerful. There is no real technology for this part other than abstract syntax trees. The project project must be able to be parsed by a compiler and this is a requirement. This does mean that there are some restrictions, but this more comes from the rigidity of a programming language rather than restrictions of the parsers. This means that the language will likely be closer to other programming languages which has many advantages for teaching the programming language. There are different kinds of decisions that can be made when designing a language and taking influences from successful languages is always good practice. For this reason abstract syntax trees are the obvious choice.

References

[1] C. verBurg, "What are flex and bison?," Dec 2018.

[2] A. Cowley, S. Diehl, M. Kiefer, and B. S. Scarlet, "llvm-hs: General purpose llvm bindings," Sep 2019.

[3] G. team, "4.11. ghc backends."