Oregon State University
College of Engineering

CS Capstone Design Document

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
Fern Bostelman-Rinaldi
Ben Warschauer
Thomas Croll
Jackson Bizjak

Abstract

We are creating a language that professors and teachers can use to teach students the basics of programming. We will design the syntax of the language by early winter term. The language will be designed around what will help the students learn programming most easily. The scanner will be implemented in flex, once we have decided on a concrete syntax and will also be completed early into the winter term. We will then work on the parser, which will be designed using bison. The parser will be complete by the middle of winter term. Code generation will be started as soon as the parser is complete and will be finished by the end of winter term. The front end will be worked on last, after other steps have been completed. front end, and error checking will not have a definitive finished state but will continue to be expanded until the end of our capstone project.

CONTENTS

1 Introduction

1.1 Purpose

The main purpose of this project is to design a DSL that is able to represent most simple board games. The target audience for the DSL is middle school students that are being introduced to programming. Board games are both familiar to students and conceptually not too different from programs, so a board game language should make the introduction to programming and programming concepts a smoother experience. Because it's an introductory teaching tool, ease of use is crucial. The syntax shouldn't be overly complicated and students should have easy access to all the tools they need.

1.2 Scope

This project is to design and implement a programming language, but this programming language does not need to be fully featured. It does not even need to be Turing complete. The main scope of this project is to simply implement something that is easy to use, even if that comes at the cost of features. An editor should also be able to work with the project. It should be well integrated with the language and give good feedback to the user. This is the extent of the entire project.

1.3 Context

The context for this project is that there are many students who have trouble understanding computer science or any of its concepts. This programming language should be able to act as a fun and semi-familiar introduction for students. There are not many programming languages designed for learning environments and most widely used languages are too complex for students to relate to. This programming language won't have the functionality that comes with that complexity, but it also won't have the confusion, making it a good entry point into computer science for young students.

1.4 Summary

Students don't have easy access to fun and educational introductory programming experiences. Most programming languages aren't approachable and many of those that attempt to be educational end up being dry, leaving the more enjoyable aspects of programming projects out of reach for beginners. This project attempts to bridge that gap between students and computer science. To do that, we will design a DSL, implement a compiler, and integrate both into an editor. While there may be more that is possible in this area, the project is mostly to try something new and hopefully grow into something more after testing.

2 Glossary

Abstract Syntax Tree
: The tree representation of a program.

Automated testing
: A tool to run a program and check if the output is valid.

Compiler
: Converts a programming language into executable machine code.

Domain Specific Language

: A programming language that is only used for a specific context, or domain.

DSL

: Domain Specific Language.

Linear Representation

: The text representation of a program.

Machine Code

: Code that is understood by a computer and difficult to read for a human.

Manual testing

: Testing a program by having a human look for errors.

Parser

: Converts code or input from the scanner into a data structure.

Scanner

: Converts code into a sequence of symbols for the parser.

Transpile

: Convert one programming language to another.

3 Body

3.1 Identified stakeholder and design concerns

The main stakeholders of this project are professors and teachers who are going to be using this software to be teaching students the basic concepts of programming. Their main concerns are that it is functional, easy to use and highly modularized. the main target audience is students who are just tarting to program. This should be accounted for in the design in making it as easy to use as possible for someone with no programming experience. This ideally means good error messages and easy to understand interfaces.

3.2 Language Design View

Language design is the first and most time consuming step in our project. The language must be able to describe a set of board games, must be simple and familiar enough to engage middle school students, and must introduce and draw attention to fundamental programming concepts. Our design process, with the goal of achieving those three things, is broken into two sections: defining the scope of our language and designing its syntax. The end goal is to teach programming concepts, so all the design choices we make should serve that purpose. We plan to finish, or mostly finish, language design a few weeks into winter term.

3.3 Language Design Viewpoint

This project is meant to introduce students to algorithms, to introduce the idea of a programming language, that different syntax is needed to communicate with a computer, and to introduce common programming constructs like loops, conditionals, functions, and arrays. An ideal language scope would be narrow enough to never distract students from those target concepts, but also broad enough to never be limiting. To find a compromise that best fits the project goals, we've grouped potential games by what they would require of our language to be implementable. Tic-Tac-Toe, the simplest game we've come up with, hardly requires anything of our language but still touches on all the important concepts. More complicated games, battleship for example, which requires hidden information, multiple game stages, and multiple kinds of pieces, could complicate the language without adding anything concept wise. However, if the language is kept so simple that only Tic-Tac-Toe can be implemented, students might not engage creatively with the product, making the learning experience less effective.

These are the constraints we are considering placing on the language: there are only two players, the board is a rectangular grid, there is only 1 phase to a game or 1 kind of move, players can only have one kind of piece, a space on the board is either occupied by a player or empty, and pieces cannot be moved or removed only placed. We think all of these constraints together are too restricting, especially considering that the cost of removing constraints, specifically access to multiple kinds of pieces and the ability to move and remove pieces, is not that high. The syntax for any potential constraint can be made completely optional through default values. Users would only have to use the more complicated, potentially distracting syntax if they wanted to take advantage of it. If they didn't, they could use the simpler syntax and the constraints would be applied by default without them having to think about them. This wouldn't be using two separate syntax, but one syntax in which complexity is optional.We will keep most of the constraints constant, but we'll use that approach to make 1-3 of them optional.

Syntax design will take up less time than defining the scope. Our primary concern involves constructs. Our language will involve conditionals, functions, loops, and some data types, but we don't want to giver students free access to them. The goal is to introduce them without overwhelming. Data types will exist in the form of pieces, players, and the board, but they won't be customizable. We want to include if statements, but we don't want students to ever write for or while loops. Loop functionality does need to be included, we just want it to be masked or implicit. The game itself, for example, is a while loop that passes between turns and checks the end game condition, but that all happens behind the scenes. We just ask students to define what happens on a turn and when the game ends. There are other helper constructs we've considered, like an in a row construct. Some of them will end up being necessary, but we're going to try and use as few as possible. The final syntax issue is effectively word choice. Our programs could read like English, like something as removed from English as C, or like something in between. Because we want to get students out of their comfort zones, to make other programming languages feel less alien, we are leaning towards the removed from English side of something in between. This design will be competed within the first weeks of winter term.

The completed design will be tested with students to see how easy they believe the language is to use. This will include surveying the students as well as watching them use it. This will likely require refinement to the language from the feedback received. This can be done for a number of cycles to ensure ease of use for the students in the compleated project.

3.4 Scanner View

The scanner is the first step in compiling, it recognizes the syntax written, and turns it into a list of symbols rather than just text. As an example, the statement “length = 10”, before being sent to a parser, would be turned into symbols that would look something like VARIABLE ASSIGN NUMBER. From this point it becomes easier for the parser to match patterns and parse the code. In our design of the scanner, we want to create a tool which will allow our concrete syntax to be implemented and will also have helpful error messages.

3.5 Scanner Viewpoint

For our scanner, we should use flex, a tool which allows you to create a scanner using C or C++. Using flex will allow us to support any language design we decide on with our scanner, while still allowing for us to create it easily. Almost all of the decisions on what the syntax should look like should be aimed at making the language easy to use. A few things though, will be influenced by the limitations of the scanner. Using flex will allow us to have a lot of resources to help build our scanner, and will enable us to create a product that is usable by brand new programmers.

Because the language is designed for middle school students, helpful error messages are very important to the functionality of our programming language. The overall goal for the language is a programming language which will introduce students to writing concrete syntax in an intuitive and helpful way. In order to create specific error messages to display to users, the compiler must have tools to detect errors. Error detection can be highly complicated depending on the robustness of the error checking. Compiling errors can occur in the scanner or parser. For the most basic detection in the scanner, any word not recognized as a valid statement in our language could be shown to the user. Therefore, if some word isn’t recognized by our scanner, we could let the user know exactly which word didn’t match. We can also aim at adding more advanced error checking. We can check for common errors in the scanner and show a specific error message for the issue rather than just showing the incorrect word.

We will create most of the scanner in a short period of time. Therefore, the more important part of testing will be manual testing where we make sure that all statements are accurately recognized, and that errors are caught and displayed in a helpful way. We should try all the statements individually, as well as full blocks of code to make sure statements are not only recognized, but also that only the desired statement is recognized. We can use automated testing to make sure that the previous functionality of our scanner does not change when we add new features. Because the scanner turns code into symbols with no input occurring while the program is running, we can just have simple scripts with input and expected output. The scanner is needed to make a complete parser, so the scanner should be built right after the language is designed to allow the parser to be developed. This means we should aim at creating the first iteration of a scanner immediately after we decide on concrete syntax early next term, or at the end of this one. The scanner should continually be worked on until it supports our concrete syntax. It should not take more than a few weeks to create the scanner itself, but adding extensive error messages to the compiler will be something that always has room to be expanded.

3.6 Parser View

The main problem for the parser is to be able to take the linear representation of the language and put it into an abstract syntax tree. The parser should be able to take any legal string and place it into this representation so that the compiler can then take this and either convert it to machine code or transpile it to another language that is possible to

compile. There are a few challenges that come with this, but the largest is representing the abstract syntax tree. This will somewhat depend on the language that is chosen.

3.7 Parser Viewpoint

This part of the project has a few options for languages, but the main choices are between Haskell and Flex and Bison implemented using C/C++. We have chosen to use the Flex and Bison implementation. the reason for this is it allows for easier conversion to assembly if wanted later, as well as easy transpilation. This part of the project has a few key aspects that are part of the parser.

The first major aspect of a parser is that it must be able to read input from the scanner into the parser. This will allow it essentially read in information from the scanner and therefore read the linear representation that the user has input. This is one of the simpler aspects of the implementation as it will simply need to read the symbols that the scanner sends it. While there are some minor challenges with this aspect, it mostly falls on the scanner to be able to do implement these. This aspect of the project should be completed slightly after the scanner, near the beginning of winter term.

The next part of the project is to generate a internal representation that the parser can use. This is the most difficult and only major part of the parser. This is often represented through a grammar which can then be converted to some sort of data structure later for the next stage of the compiler to use. This can be somewhat difficult as the grammars that can be used for this part is limited. This can lead to errors if these rules are not followed. After the parsing has bee completed the data structure will be generated. The data structure often mirrors the grammar, so this aspect should not be complicated. Finally once this internal representation has been made it simply needs to be passed to the next stage of the compiler. This aspect of the project should be completed a few weeks into the winter term. Testing this part of the project is relatively easy and can be done by printing out the generated tree and seeing if it matches the linear representation that is passed into the parser.

3.8 Code Generation View

Code generation represents the most significant portion of this project. After input has been run through the scanner and parser, internal representation is generated. This is usually in the form of an abstract syntax tree (AST). In the code generation stage, this abstract syntax tree is converted into some other target language. This is often assembly or machine code in the case of a true compiler, or some other higher level language in the case of a transpiler. Another crucial element of a compiler that occurs during this stage is error checking that can not be performed in the parsing stage. Errors of this form can not be captured by the grammar of the parser and must be extrapolated from the AST.

3.9 Code Generation Viewpoint

The compiler creation tools flex and bison will be used for generating the scanner and parser respectively. These tools are heavily integrated with the C and C++ programming languages. As such, it is a natural extension to write code generation code in either C or C++. This language choice would allow for maximum integration with the parser and scanner.

For the purposes of this project, it is not very useful to write a full compiler that generates assembly or machine code. While this could be more efficient than a transpiler, it would be more difficult to write and less portable. For these reasons, the compiler will generate C++ code. This allows it to use the g++ compiler for generating executables. Using the a separate compiler to build executables from the target language ensures portability and ease of code generation.

Parts of this stage of the project will likely need to be done after the parsing and scanning stages are already complete. This is due to the heavily integrated nature of these stages. It would be difficult to test and complete code generation without other portions being completed. Additionally this step in compiler creation is typically the most complicated and involved, and is expected to take the greatest amount of time. Code generation will be completed in stages, beginning with simple target language generation, and moving on to more complex features such as type checking and error messages. This would ensure a functioning prototype can be created in a reasonable amount of time, and would allow for more easy testing of features. Type checking and comprehensive error messages will be finished last because they will take the most amount of time and effort. This will lead to this section of the project being finished around the middle of the term.

Testing the code generation should be relatively straightforward. The compiler should be generating code that can be compiled into functioning executables. Thus, verification of the compiler will involve inputting programs in the DSL and ensuring that they produce the desired functionality. To this end, various testing programs can be created. It is expected that the langauge be able to produce simple board games such as TicTacToe and Connect4. Examples of these games will be included for demonstrating language syntax, as well as verification.

3.10 Front End View

The front end for this DSL needs to be intuitive and easy to understand, as the two main users of it will be middle school students and teachers who will have little to no prior experience with the language or front end. The majority of users will have never experienced this language or front end before their first use in a classroom.

The front end will need to serve both as an introductory learning experience and also as a tool for those with more knowledge. It needs to be far more user friendly than a lot of other text editors, unlike Atom or Notepad++. These users will have little to no experience with programming prior to this, so this front end needs to appeal to a user who has never used a traditional coding editor before.

There needs to be an extensive and easily accessible documentation for the users to reference and be able to use when they need to problem solve, as unlike other languages there won't be online resources they can use. The documentation will need to completely cover all aspects of the language and give suggested solutions to commonly found problems. Besides that the front end will also need to offer a gradual introduction into the language that will allow user to gradually learn without becoming overwhelmed or confused.

3.11 Front End Viewpoint

Using C# we'll create a front end that's a portable application, allowing it to run on a multitude of systems with different specs. It will allow Linus Pauling Middle School to run it on whichever system they choose. This will also allow creation this front end in parallel with creating the actual DSL, so we can begin development on it before the language is entirely completed.

By mid February a working text editor will be developed that will read and parse code written in our DSL. This editor will allow for code to be written and ran to create a board game such as tic-tac-toe or connect four. This will just be a basic code editor, similar to notepad++ without any syntax highlighting or in-depth error messages.

By mid March the editor will include syntax highlighting and error messages. These will aid in helping users learn the DSL and create running and functional board games. Syntax highlighting will help the users catch small syntax errors, such as missing punctuation or undefined terms they may be trying to reference. Error messages will display

what line the error happened and what the program may have been expecting or what happened to cause it to error, allowing users to narrow down within their code where it is running into errors.

Finally, by the end of March there will be a full documentation contained within the front end that will be easy for users to access and reference while programming with the application. It will contain documentation on all of the DSL and examples on how to use the editor, including help sections to troubleshoot common errors a user may run into. It will also include example board games a user can recreate within the application as a tool to better understand and learn the DSL.

To test this portion of the project, prototypes will be given to the middle schoolers at Linus Pauling Middle School as well as the client. They will be tracked using the tool as well as surveyed after using the tool to help guide development of the project. Their feedback will be used to test the implementation as well. This will allow the project to develop in a way that is guaranteed to be a good fit for the students.

3.12 Design rationale

the main point of all of these designs it to ensure that it is easy for a middle schooler to use this software. That means that as much as possible needs to be hidden from the user as possible. For example the compiler is not something that the user needs to be able to understand, but the user interface needs to be very easy to use. In many ways the design choices of everything is made for ease of use for the user, but for things that the user cannot see, ease of development is the main priority.