**Design Optimizer – Final Project Report**

Hanavan Kuhn

[hanavan@ksu.edu](mailto:hanavan@ksu.edu)

CIS 625

November 11, 2020

**Goals**

The goal of this project is to allow users to easily input specifications for a problem into a program and have it solved without needing to work with the equations themselves.

**Design Goals**

* The program should be able to perform minimization, maximization, and also find a specific value for a problem.
* The scripting language should be easy to use and require as little of a learning curve as possible, making it easy for non-programmers to use.
* The solver should provide automatic unit verification and unit conversions for expressions and properties, so as to prevent mistakes in calculation.

**Implementation**

**Language**

I considered two programming languages: Java and Go. I am more familiar with Java, but I decided to use Go for the project because I wanted to add the extra difficulty of learning a new language on top of the original design goals. I also considered using two development environments: Notepad++ and Visual Studio Code. VS Code was more difficult to set up, but provided compile checking and syntax highlighting, which made development much easier. Notepad++ has syntax highlighting as well, but it is not as powerful, so I ultimately decided to use VS Code.

**Scripting Language**

To make the scripting language, I had to first decide the overall style I wanted to use. I decided on modelling the language after C, using curly braces to indicate scope, and semicolons to indicate the end of statements. As a language functionality goal, I wanted to provide the ability to specify all of the configuration options for the optimizer directly in the script, without the use of command-line arguments or flags. This helped achieve the design goal of making it more user-friendly.

**Scripting Language Syntax**

In order to provide all of the functionality for the optimizer, five structures were provided to allow users to specify the needed information in the scripting language.

* Unit structure – allows the user to specify a unit, either as a base unit (e.g. “m”) or as a composition of other units (e.g. “m/s”).
* Property structure – allows the user to specify a property of an object (e.g. “mass”), as well as the units of the property (e.g. “kg”). Optionally, if the value of the property is known, it can also be specified. Additionally, if the value of the property is a function of other property values, an expression can be used to define the value (e.g. “width \* height”).
* Assembly structure – allows the user to organize properties into groups. This is useful when defining multiple properties that are all derived from the same object (e.g. a baseball, which has properties like “mass”, “radius”, “volume”, etc.)
* Enumeration structure – allows the user to create a set of items with similar properties (e.g. a material, which has a density, cost, buoyancy, etc.).
* Solve structure – allows the user to specify parameters and type of optimization to use to solve the design problem.

**Program Structure**

The program has three main stages, which fit together in a pipeline structure. The output of the first stage is used as the input to the second stage, and so on.

* Tokenizer – takes in a stream of characters (in this case a file), and converts them into a list of tokens.
* Parser – takes a list of tokens and converts them into a parse tree.
* Solver – takes a parse tree and examines the structure to produce an output.

**Tokenizer Functionality**

The tokenizer can take in a stream of characters to construct a list of tokens. An example of this is shown below.

Input string: property area : m ^ 2 = width \* height;

Output token list:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Property | area | : | m | ^ | 2 | = | width | \* | height | ; |

**Parser Functionality**

The parser accepts the previously generated token list, and converts it into a parse tree.

Input token list:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Property | area | : | m | ^ | 2 | = | width | \* | height | ; |

Output parse tree:

Property: “area”

Units

^

m

2

Value

\*

width

height

**Solver Functionality**

To perform the optimization, the parse tree generated from the previous step is used to identify which type of optimization to use. This is done through by locating the “solve” structure inside of the parse tree, which is not shown in the above tree. Next, the solver will perform gradient descent/ascent in order to determine which values could be used. In this case, “width” and “height” are properties that do not have a value, and so they will be used as inputs to the function. The gradient descent algorithm will then slowly modify their values until the “area” property is maximized or minimized.

**Features Implemented**

* All language features (unit, property, assembly, enumeration, and solve) are implemented and functional.
* Minimization and maximization functionality is implemented.
* All properties can have units defined.

**Difficulties**

Naturally, some difficulties arose throughout the development of the project. Two major difficulties arose that made it intractable to solve given the time constraints of this project.

* I attempted to implement a gradient descent algorithm by hand, which was initially a success. However, upon testing with functions that had a very small gradient, the algorithm’s convergence was very slow. I ended up finding a 3rd party library which worked much better, since the authors had much more time to think about and implement a more robust solution that I had for this project.
* Unit conversion and checking became difficult while trying to implement, due to the amount of logic required to compare the parse trees of the units with the parse trees of the expressions to verify that they were a match. Additionally, the calculation of the conversion factor required a similar amount of logic, which was impossible to implement given the remaining time available to finish the project.

**Features Not Implemented**

Because of the aforementioned difficulties, a few features that I had initially planned on implementing did not get come to fruition.

* Units are not checked or converted. Even though the units must be specified on every property, their values are not used during calculation. They are properly parsed, however they are not used ever after the parsing stage.
* Enumerations work internally, but they cannot be used properly in the script. An explicit value for a property that has an enumeration type can be specified by hard-coding its value into the code, but not in the script itself.
* Solving functions for an exact value is not implemented. Finding the zeros of a function is more difficult than performing gradient descent, but when I attempted to implement it, the amount of math involved was much greater than I anticipated.

**Future Ideas**

Throughout development, I came up with some ideas that weren’t part of the original goals, but would be nice to implement later on.

* Providing an “include” functionality, similar to C’s preprocessor “#include” macro. This would allow for users to define commonly used units in a file that could be included, making scripts more reusable.
* Creating an abstraction system that would provide a way to easily reuse commonly used objects. For example, an abstract “circle” assembly could be created that could be extended by another assembly if it has circle-like properties. Similar to Java’s class system.
* Equality and inequality constraints, which would allow for many more types of optimization problems to be solved. This would allow users to specify things like “constraint mass : kg <= 5;”, which would make sure that the mass must stay less than or equal to 5 kg.
* Property names can be resolved without needing to be fully qualified. The current implementation requires that any expression that references another property must be fully defined. This would make scripts easier to read and easier to write.