Design Optimizer - Project Design

Hanavan Kuhn
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.

Functional Description

The program will have a single mode of operation, which is to take in a program written by the user, and produce an output that matches what the user asked for in the program. If the program is syntactically correct, the requested information will be printed to the console, and if the program is not syntactically correct, the first error encountered will be printed to the console.

Input Program Syntax

The programming language that is interpreted by the optimizer has several language constructs. Each of these constructs must be specified in the following order:

1. Units
2. Enumerations
3. Assembles/Subassemblies
4. Summarize(s)
5. Solve

Below are more in-depth descriptions of each language construct.

1. Unit

A unit is a label that indicates the units a value was measured in. In the program syntax, a unit can be defined in 2 different formats. The first format defines that a unit exists, and that it is a *base unit*, which means that it is not defined as a composition of other units.

```
unit unit_name;
```

The second format defines a unit based on one or more previously defined units, as well as a conversion factor, expressed as a floating-point number.

```
unit unit_name = <num> <uexp>;
```

`<num>` is the floating-point conversion factor, and `<uexp>` is a unit expression, which consists of other units and only the *, /, or ^ (exponentiation) operators.

2. **Property**

A property is a component of an enum or an assembly, and is used to define a property's value and associated units. A property can be defined in 3 different formats. The first format defines that a property exists, and that it has a particular unit associated with it.

```
property property_name : <uexp>;
```

`<uexp>` is a unit expression defining the units of the property. The second format defines that a property exists, and that it has a constant value and a particular unit associated with it.

```
property property_name : <uexp> = <num> <uexp>;
```

`<num>` is a floating-point value, and `<uexp>` are both unit expressions defining the units. The units of the property and the value are defined separately because the optimizer will automatically convert between units based on what units are given, provided that the units are convertible.

The third format defines that a property exists, and that it can be calculated by evaluating an expression. This expression can contain references to previously defined properties.

```
property property_name : <uexp> = <exp>;
```

`<uexp>` is a unit expression, and `<exp>` is an expression consisting of constants and previously defined properties.

3. **Enumeration**

An enumeration is a structure that defines a list of *n* properties $P_1 \ldots P_n$, as well as a fixed set of named tuples $V_1 \ldots V_n$ containing values corresponding to each property. The format for an enumeration is given below.

```
enum enum_name {
    property property_name1 : <uexp>;
    property property_name2 : <uexp>;
    …

    value value_name1(<num> <uexp>, <num> <uexp>, …);
    value value_name2(<num> <uexp>, <num> <uexp>, …);
    …
}
```

`<uexp>` are unit expressions, and `<num>` are floating-point numbers. Each value V_x must have *n* comma-separated values, with $V_{x,y}$ corresponding to the value of P_y.

4. Assembly

An assembly has a list of properties, as well as a set of subassemblies contained within. This allows for separation of different parts of the problem into a hierarchical tree of assemblies. The format of an assembly is given below.

```
assembly assembly_name {
    assembly subassembly_name1 {
        …
    }

    assembly subassembly_name2 {
        …
    }

    property property_name1 : <uexp>;
    property property_name2 : <uexp> = <num> <uexp>;
    property property_name3 : <enum>;
    property property_name4 : <enum> = <evalue>;
    property property_name5 : <uexp> = <exp>;
}
```

`<uexp>` is a unit expression, `<num>` is a floating-point number, `<enum>` is the name of an enumeration, `<evalue>` is the name of a value in the enumeration `<enum>`, and `<exp>` is an expression.

5. Summarize

The summarize directive tells the optimizer to include the value of a particular property in the final result output. When the optimizer program is run, if it is successful in finding a solution, the value of this property will be printed to the console's output.

```
summarize fully.qualified.name;
```

A period is used to separate identifiers, in which each identifier refers to the name of an assembly, property, or enum property.

6. Solve

The solve directive tells the optimizer to try to find a suitable value for the given property, provided the property's value is not constant. The solve directive can be specified in 2 different formats. The first format is used when maximizing or minimizing a property.

```
solve [maximize|minimize] fully.qualified.name;
```

The second format is used when finding where a property equals a particular value.

```
solve [set] fully.qualified.name = <num> <uexp>;
```

`<num>` is a floating-point number, and `<uexp>` is a unit expression.

Sample Input Program

Below is a simple program that could be used to calculate the height of a model rocket nosecone based on the fact that the nosecone’s mass needs to be 300g. It would also substitute every value for the “material” enum to find the height of a nosecone made out of ABS plastic and PLA plastic.

```
unit kg;
unit m;
unit s;
unit g = 0.001 kg;
unit cm = 0.01 m;

enum material {
     property density : kg/m^3;

     value PLA(1.24 g/cm^3;);
     value ABS(1.04 g/cm^3;);
}

assembly rocket {
     assembly body {
          property radius : m = 4 cm;
     }

     assembly nosecone {
          property mat : material;
          property length : m;
          property mass : kg = mat.density * (pi * length *
rocket.body.radius ^ 2) / 3;
     }
}

summarize rocket.nosecone.mat;
summarize rocket.nosecone.length;

solve set rocket.nosecone.mass = 300 g;
```

Implementation

The implementation of this program will be written in Go (golang). The program will consist of 3 parts, a tokenizer, a parser, and a solver. The tokenizer is responsible for converting the input file into a list of tokens, and reporting errors such as malformed numbers or illegal characters. The parser is responsible for turning the list of tokens into a parse tree, as well as reporting any errors such as malformed language constructs. The solver is responsible for taking the parse tree and using that to solve the problem, as well as reporting information like undefined units, mismatched units, undefined properties, etc.

The parser will parse expressions using the shunting yard algorithm or the recursive descent technique. Some fallbacks of the shunting yard algorithm may imply that recursive descent may be a better choice.

The solver will be implemented by using gradient descent/ascent, which can be used to find a local maximum or minimum within a continuous multi-variable function. This means that the optimizer is not guaranteed to be able to find a global maximum or minimum, but will only find the one closest to where the descent algorithm's starting point.

Parse Tree Structure

The parse tree structure has two major portions, the expression tree structure and the parse tree structure. The expression structure is almost identical to the expression grammar in languages like C, except for when units can be specified after a constant.

On the next page is the structure of the parse tree. Names in square brackets indicate optional parameters. Nodes that can have multiple children of a single type (such as enums with multiple properties) are only shown as having a single child of that type for the sake of brevity.

```
~ root
   ~ unit
      ~ name : string
      ~ [multiplier] : float
      ~ [units] : expression
   ~ enum
      ~ name : string
      ~ property
         ~ name : string
         ~ [units] : expression
         ~ [value] : expression
      ~ value
         ~ name : string
         ~ argument
            ~ value : float
            ~ units : expression
   ~ assembly
      ~ name : string
      ~ subassembly
      ~ property
         ~ name : string
         ~ [units] : expression
         ~ [value] : expression
   ~ summarize
      ~ property_name : string
   ~ solve
      ~ property_name : string
      ~ [value] : float
      ~ [value_units] : expression
```