Blueprinter

Benjamin Wilen

0.1 Introduction

Blueprinter is a programming language for producing blueprint designs of any home, building, etc. This programming language attempts to make it easy to convert a design into graphical blueprints, and even easier to iterate on those designs. For example, say someone wants to build a floor of a building, and they have just made a blueprint of the rooms. The architect decides they want to make one room larger, they shouldn't have to waste time recreating a whole blueprint, but can instead modify the program in this language to meet the new specification. Finally, this language will enable architects to reuse components of a design.

I believe this problem should have its own language because it will greatly improve the efficiency of designing spaces, but also because I believe a language for this problem already exists in oral terms. Architects primarly communicate to contractors and builders via two methods: verbally passing on instructions and through blueprints. What if these languages could be combined? If an architect wants a 150 by 300 ft living room with a central table and a fireplace, why can't a computer build a blueprint for them? This language will accomplish that, and be both general enough to allow designs of any space, while specific enough to be clear and easy to program in. The output of this program will be a handful of SVG images, each representing a floor of the space similar to traditional blueprints.

Lastly, this language will enable easier analysis of blueprints. As future steps, this language should include ways for an architect to analyze features of a blueprint, such as how many square feet a plan is, what is the breakdown of bed and baths, how to properly add plumbing and electricity, and many other features.

0.2 Design Principles

The goal of Blueprinter is to build a language that reflects how an architect thinks through the design of a space. As a result, there are three guiding design principles that Blueprinter will use. 1) There are three primitive objects in a design: levels, rooms, and furniture. Although we hope to make our language as generalizable as possible, these three objects can represent almost any structure. A level corresponds to a floor, or one SVG image. A room is a enclosed space, and can contain rooms or furniture within it. Furniture is the most atomic part of a blueprint, and represents a physical thing. 2) The language should enable nested substructures to model nested spaces. Therefore, rooms and furniture can be within levels, and rooms and furniture can also be within rooms (you can have sub-rooms). Locations will also be relative to the parent to enable easy modeling. For example, if a room is in the middle of a floor, and I want to put a furniture in the top left of the room, instead of finding that location on the entire blueprint, the furniture's location will just be (0,0) (the top left). 3) The code should be modular and reusable. A programmer should have the option to define a design, with parameters, and instanciate it throughout a design. For example, if an architect creates a spiral staircase, they should be allowed to use it in multiple places without having to redefine it. This will also be how a standard library of rooms and furniture can be defined.

0.3 Examples

To run an example, navigate to the code directory and run "dotnet run <example_name.bp>". The examples are currently stored in the code directory. All three examples produce the same output, shown below, but the first example directly defines it, the second example uses a type definition with no arguments, and the third example uses a type definition with arguments.

```
Output for All Three (inside example.svg):
<svg width="1220" height="720" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w
<rect x="10" y="10" width="1200" height="700" fill="none" stroke="blue" stroke-width="2" />
<rect x="10" y="360" width="200" height="350" fill="none" stroke="blue" stroke-width="2" />
<text x="110" y="535" font-family="Arial" font-size="20" fill="black" text-anchor="middle"
</svg>
```

```
Example 1 (directly defining room):
Run with command: "dotnet run example_1.bp"

Example 2 (using type definition):
Run with command: "dotnet run example_2.bp"

Example 3 (using type definition with arguments):
Run with command: "dotnet run example_3.bp"
```

0.4 Language Concepts

The primitive values in Blueprinter are strings and numbers. The combining forms are Furniture, Rooms, Levels, Type Instances. Lastly, primitives can be stored in variables and combining forms can be stored in type definitions. A program consists of Level objects as well as type definitions.

Because primitive values in Blueprinter follow a similar form and usage as other programming languages, I'll discuss combining forms and type definitions in this section. Furniture, Rooms, and Levels are initialized with properties. Properties are a pair seperated by an equal sign, with the key being a string and the value being a string, num, or variable. These are used to define characteristics such as name, id, dimensions, size, shape, etc. For this implemention, x, y, width, and height are required, and name can be optionally defined if you want it to show up. Attributes are checked at runtime in the evaluation phase of a program. Rooms and Levels also have children objects, which can be Rooms and Furniture. Because of this nested property, rooms and levels have a recursive nature. A level corresponds to a single SVG image to mimic Blueprint design where each floor is a seperate sheet. As a result, Levels cannot be defined with Levels.

Type definitions enable reusable code. A type definition is similar to a simple function that only takes arguments and returns a value (no intermediary calculations). Within a type definition can be anything included in a normal Blueprinter program with the exception that a type definition currently cannot be defined inside another type definition. For implementation simplicity, I use dynamic variables where the latest instantiation is the current value. Type definitions are also dynamic as the latest type definition for its type name is the one considered when it is instantiated. Type Instances require and x and y property, and the neccessary arguments specified in the type definition.

0.5 Formal Syntax

```
<Blueprint> ::= <expr>+
<expr> ::= <typeDef> | <level>
<typedef> ::= type <var>(<pars>){<children>}
<children> ::= <child>*
<child> ::= <room> | <furniture> | <typeinstance>
<level> ::= Level(<properties>) {<children>}
<room> ::= Room(<properties>) {<children>}
<furniture> ::= Furniture(<properties>)
<properties> ::= <property>*
<property> ::= <key>=<value>,
<typeinstance> ::= <var>("x"=<value>, "y"=<value>, <args>)
<key> ::= "<string>"
<args> :: <value>*
<value> ::= "<string>" | <num> | <var>
<var> ::= <char>+
<pars> ::= <var>*
<num> ::= <int>
<int> ::= <num>+
```

```
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<string> ::= <char>*
<char> ::= {'A'...'z'}

*Blueprinter is whitespace insensitive
```

0.6 Semantics

Notes on Semantics: For Levels, a ”filepath”, ”width”, and ”height” property must be defined. For rooms and furniture, ”x”, ”y”, ”width”, and ”height” properties must be defined. Name is optional depending on if you want it to show up on the blueprint. For type instances, ”x” and ”y” properties must be defined with x defined first, then y.

Syntax	Abstract Syntax	Type	Meaning
n	Number of int	int	n is a primitive
”string”	string	strin g	”string” is a primitive
x	string	string	stored variable identifier
”x”=4	Expr * Expr	’a ->’b ->string	Attribute of an object such as location or size
Furniture()	Expr list	’a list ->string	a furniture object that returns the svg of the furniture item
Room(){}	Expr list * Expr list	’a list ->’b list ->string	a room object that returns the svg of the room item
Level(){}	Expr list * Expr list	’a list ->’b list ->unit	a level object that returns the svg of the level item
type Test(){}	Expr * Expr	’a ->’b ->Expr	a type definition for resuable object collections
Test()	Expr * Expr * Expr * Expr list	’a ->’b ->Expr	a type instance with ”x”, ”y”, and arguments listed

0.7 Remaining Work

There are four future improvements that would strengthen Blueprinter. These features enable the language to be more flexible, collaborative, and analytical. First, the language should allow more attributes to be specified on rooms and furniture, such as shape, color, filling, etc. Second, furniture should allow a filepath to an image instead of just a rectangle (for example, a couch could be an icon of a couch). Both of these improvements would enable blueprints to be more flexible and descriptive. Third, Blueprinter should allow for imports of type definitions. The way this could be implemented is adding the type definitions gathered from the imported AST and use it as the base environment for the program. This would allow standard libraries of rooms and furniture, such as staircases, common furniture, doors, etc. and would make Blueprinter a collaborative, open-source language. Lastly, Blueprinter can be extended to enable analytics on a program. Analytics could range from square footage to a room breakdown to even more complex analysis such as wasted space and flow of people.