Blueprint

Benjamin Wilen

0.1 Introduction

Blueprint 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 primarily 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 Blueprint 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 Blueprint 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). 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

```
Example 1
Run with command: "dotnet run example_1.bp"
Output: Nothing
(Because there are only type definitions and not actually instances,
nothing is generated. This is equivalent to a program in python that
is just x = 5)

Example 2:
Run with command: "dotnet run example_2.bp"
Output: Produces main_floor.svg
Inside of main_floor.svg:
<svg width="400" height="110">
  <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)
    <text x="0" y="10" font-family="Verdana" font-size="55" fill="blue"> Living Room </tex
  </rect>
</svg>

Example 3:
Run with command: "dotnet run example_3.bp"
Output: Produces main_floor.svg
Inside of main_floor.svg:
<svg width="400" height="110">
  <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)'
    <text x="0" y="10" font-family="Verdana" font-size="55" fill="blue"> Living Room </text
  </rect>
</svg>
```

0.4 Language Concepts

The primitive values in Blueprint are strings and numbers. The combining forms are Furniture, Rooms, and Levels. 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 Blueprint follow a similar form and usage as other programming languages, I'll discuss combining forms, variables, and type definitions in this section. Furniture, Rooms, and Levels are initialized with attributes, which can be strings or numbers. These are used to define characteristics such as name, id, dimensions, size, shape, etc. 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.

Variables and 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 Blueprint program with the exception that a type definition currently cannot be defined inside another type definition (this might change in the future). 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.

0.5 Formal Syntax

```
<Blueprint> ::= <expr>+
<expr> ::= <typeDef> | <level>
<typedef> ::= type <var>(<pars>){<instances>}
<instances> ::= <instance>*
<instance> ::= <level> | <room> | <furniture>
<children> ::= <child>*
<child> ::= <room> | <furniture>
<level> ::= Level(<attributes>) {<children>}
<room> ::= Room(<attributes>) {<children>}
<furniture> ::= Furniture(<attributes>)
<attributes> ::= <attribute>*
<attribute> ::= <key>=<value>,
<key> ::= "<string>"
<value> ::= "<string>" | <num> | <var>
<var> ::= <char>+
<pars> ::= <var>*
<num> ::= <float> | <int>
```

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

*Blueprint is whitespace insensitive
```

0.6 Semantics

Syntax	Abstract Syntax	Type	Prec/Assoc	Meaning
n	Number of int	int	n/a	n is a primitive
”string”	EString of string	string	n/a	”string” is a primitive
x	Variable of string	string	n/a	stored variable identifier
”id”=”ex”	Attribute of Expr * Expr	’a ->’b ->string	n/a	Attribute of an object such as location or size
Furniture()	Furniture of Expr list	’a list ->string	2, N/A	a furniture object that returns the svg of the furniture item
Room(){}	Room of Expr list * Expr list	’a list ->’b list ->string	2, N/A	a room object that returns the svg of the room item
Level(){}	Level of Expr list * Expr list	’a list ->’b list ->unit	2, N/A	a level object that returns the svg of the level item
type test(){}	Assignment of Expr * Expr	’a ->’b ->Expr	1, N/A	a type definition for resuable object collections

0.7 Remaining Work

There is one feature neccessary to the programming language, and three cool extensions (I’ll hopefully be able to implement 1 or 2). First, I need to implement the evaluation of Levels, Rooms, and Furniture. This will involve converting those objects into SVGs. For this project, I expect everything to be denoted as rectangulars, but a cool future extension would be to enable images for furniture. The three extensions not neccessary for the MVP are:
1) Lexical scoped variables and type definitions (to enable nested type definitions).
2) Importing type definitions from other programs (would allow me to make a standard library of objects).
3) Expressions for analysis of blueprints.