# Documentation: Blueprint

Benjamin Wilen

## 0.1 Overview of Minimal Working Version

Given Blueprint's purpose of enabling users to create blueprints via a program, there are two main features that are essential to the functionality of the program. 1) Users should be able to define reusable types. For example, if an architect reuses a kitchen layout, they might want to save that as a custom type. 2) Users can use instances of built in types and custom types to produce Blueprint objects. Although the 2nd feature is the more critical one for the output of this language, for the minimially working version I implemented the 1st feature Because I think it is the building block of the language and should be used internally as well. For example, for the final project, I will have a standard libraries of objects such as couch, fireplace, sink, staircase, etc. which internally can be implemented as custom types.

To implement type definitions, I treat type declerations as non-scoped function declerations. A type definition has a name, optional formal parameters, and returns a Expr. As a result, I treat functions similar to how we store variables, using an env map that holds the type name as the key, and the value is a TypeDef object which includes the parameters and the children objects. Lastly, I do not consider scope for function implementation because these are simple functions in the sense that you can't do logic or computations within the function, only return children objects.

For the final version of this project, the output will be SVG files for each level specified in a program. However, because the second feature isn't implemented in this version, I currently print the env returned from evaluation to check that the type definitions were properly specified and stored in the environment.

## 0.2 Minimum Formal Grammar in BNF

```
<Blueprint> ::= <expr>+
<expr> ::= <typeDef> | <instance>
<typedef> ::= type <var>(<pars>)[<instances>]
<instances> ::= <instance>*
<instance> ::= <level> | <room> | <furniture>
<level> ::= Level(<attributes>) [<instances>]
<room> ::= Room(<attributes>) [<instances>]
<furniture> ::= Furniture(<attributes>)
<attributes> ::= <attribute>*
<attribute> ::= <key>=<value>,
<key> ::= <string>
<value> ::= <string> | <num> | <var>
<var> ::= <char> | <var><char>
<pars> ::= <par>*
<par> ::= <var>,
<num> ::= <d><num> | <d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<char> ::= {'a'...'z'}
```

## 0.3 Minimal 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 o |
| Furniture() | Furniture of Expr list | ’a list -¿ string | 2, N/A | a furniture object that returns the svg of the fu |
| Room()[] | Room of Expr list * Expr list | ’a list -¿ ’b list -¿ string | 2, N/A | a room object that returns the svg of the ro |
| Level()[] | Level of Expr list * Expr list | ’a list -¿ ’b list -¿ string | 2, N/A | a level object that returns the svg of the le |
| type test ()[] | Assignment of Expr * Expr | ’a -¿ ’b -¿ Expr | 1, N/A | a type definition for resuable object coll |

## 0.4 Next Steps

For next steps, there are obvious extensions and improvements that need to be made to this working version. First, as mentioned earlier, the implementation of actually created blueprint SVGs. This will require a more complex evaluation function, however, I believe the current parsing function already includes this feature. The AST needs to be cleaned up a little bit to be less flexible with certain nested layers. For example, given the current AST, a program can be just a number, but a program should only be a Sequence of Assignments, Levels, Rooms, and Furniture. Finally, the grammer parsing should be cleaned up and improved slightly for readability, most noticably allowing whitespace flexibility.