Forest++

Simon Jenkins, Sarah Ling

1 Introduction

There are several situations in which one finds themselves in a considerable lack of trees. Perhaps they've been sheltered in a windowless bunker amidst the doomsday apocalypse, unable to look outside at or even remember the natural beauty they once took for granted. Maybe their original character needs a green (or red) forest landscape in which to frolic. It's these situations that we plan to craft *Forest++* to solve.

Forest++ is a programming language that provides the user, who may not have past programming experience, with a simple tool to create the randomly generated forest of their heart's desire. The types of trees, their sizes, their colors, and their abundance can all be changed at the user's will to create a forest tailor fit for them (or made random). This basic concept can be expanding on with more SVGs, giving a larger breadth of tools for artists who don't have a coding background.

2 Design Principles

The main principle of the language is simplicity. The user simply needs to provide the number, type, season, and size of the trees within the forest in a straightforward syntactical form: an English list. This intuitive structure gives the user enough guidance to alter the elements of the forest easily. It also abstracts most of the technical details under the hood so that the user is less intimidated and it is more accessible for people without coding backgrounds to create.

3 Examples

A sample *Forest++* program could utilize the following input:

```
dotnet run "fall: 15 birch 1.0 size, 1 pine 2.0 size, 3 oak 1.5 size"
```

which would form a randomly generated forest with 15 birches, a pine of two times normal size, and three oaks of 1.5 times normal size, all in their fall colors spread on a canvas. On the canvas, the 15 birch trees would serve as a bottom layer, followed by a layer of pines, and then of oak, since the program determines the location of the trees in the order of the list.

Here's another example:

```
dotnet run "spring: 8-10 oak random size, 3 maple 0.5 size, 8-10 oak random size"
```

This program would produce and place on a canvas a random number from 8 to 10 of oaks and in a random size (but consistent for all 8 to 10 of the oaks), three maples of half size, and another random number from 8 to 10 of oaks of a new random size, all in their spring colors. This is to show that the user can utilize a tree type more than once. In addition to using the tree type more than once, the program lets the two instances of tree type (oak in this case) have different sizes. As for randomization, for a random number of trees the user only has to provide a range of values. *Forest++* can generate a number of trees within the given range for the forest. For generating a random size, the keyword "random" must be used.

Here is a final example:

```
dotnet run "fall: 1 oak 2.0, 1 pine random, 1 birch"
```

This example showcases the leniency of our program. If the word "size" is omitted, the program can still parse that the second number is the desired size, as noted in "1 oak 2.0". Similarly, the random function can still be used when the word "size" is omitted as seen in "1 pine random". When a size is omitted completely, like "1 birch", it will default to size 1.0. A season must always be indicated.

Forest++ only takes size values in the range of 0.25 - 2.0. These values must be floats. A number of trees must always be provided. A season must always be provided.

4 Language Concepts

The key primitives involved are the type of trees, the size of the trees, and the seasons. In particular, the user should know the four available tree types (Maple, Oak, Birch and Pine) the range of potential tree sizes (0.25-2.0), and the two seasons available (spring and fall).

The central combining form is the "grove," i.e., a group of trees of the same type. A program essentially specifies a list of groves, by providing the type of tree in the grove, the size of trees within it, and the number of trees. And so, a "forest" is a combining form that is a list of groves, and specifies exactly which trees will be generated in the final image. A forest may have multiple groves of the same type with different sizes, numbers, etc. Finally, a "landscape" consisting of a forest and season will specify the trees and colors of trees that are generated on a canvas.

5 Syntax

```
<landscape> ::= <season>:␣<forest>
    | <forest>

<forest> ::= <grove>,␣<forest>
    | <grove>
<season> ::= "spring"
    | "fall"
<grove> ::= <n> <treetype> <size>

<n> ::= <int>
    | "random"
    | <int> - <int>
<int> ::= Z+

<size> ::= e
    | "random"
    | <fp>
<fp> ::= R in {0.25, 2.0}

<type> ::= "maple"
    | "oak"
    | "birch"
    | "pine"
```

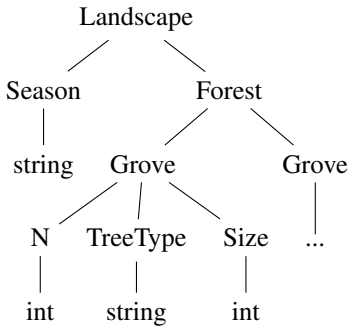
Where "e" is an empty string.

6 Semantics

The central primitives of *Forest++* are the `TreeType` of trees (Maple, Oak, Birch, Pine), the `Size` of a group of trees, the `Season` of the forest, and the `N` number of trees.

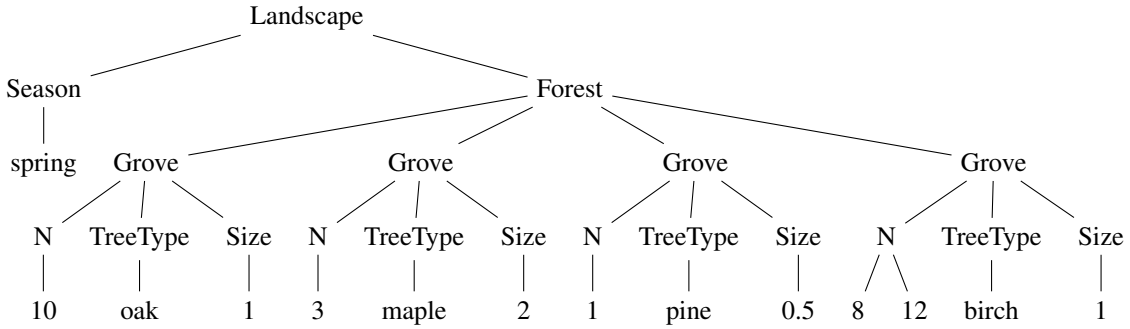
These primitives are combined into a key combining form, the `Grove`, which is a group of a `N` trees with a given `Size`. `Grove`s are made of `Tree`s, which aren't necessarily able to be explicitly stated by a user as input, but make up the user-specified `Grove`s. The next higher combining form would be a `Forest`, which is a list of `Grove`s. Finally, the `Landscape` would be a `Forest` with a certain `Season`.

This tree shows how the types within the AST of the language fit together:

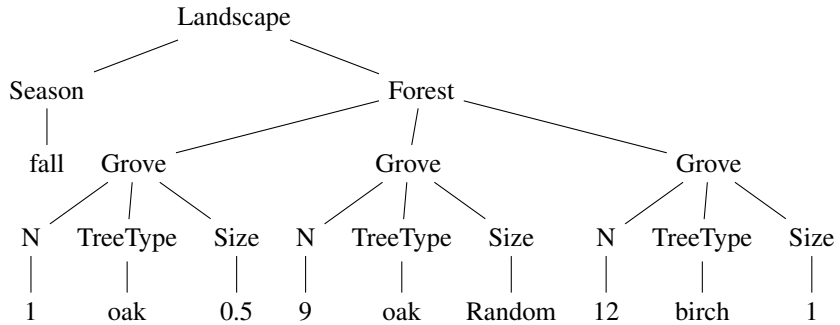


These sample abstract syntax trees for the above three examples show how the AST elements fit together:

```
spring: 10 oak, 3 maple 2.0 size, 1 pine 0.5 size, 8-12 birch
```

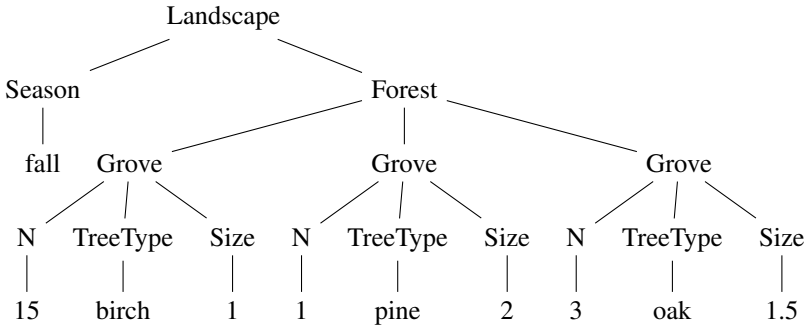



```
fall: 1 oak 0.5 size, 9 oak random size, 12 birch
```



Forest++ programs read a string describing the landscape to be generated, with the syntax described above. The output of evaluating the program will be an SVG file, with a forest of trees generated to the specifications of the input.

To go through an example in more depth, consider this program: `fall: 15 birch, 1 pine 2 size, 3 oak 1.5 size`. This program has the following AST:



Using post-order traversal, evaluation of this AST would first involve storing the season `fall`. Then, it would evaluate the three groves within the forest. When evaluating the first birch grove, the position of 15 trees on the canvas would be randomly generated. Then the sample birch tree SVG for the given season (fall) would be assigned to the positions. Finally, since the size is 1, the trees would not be scaled. To complete the landscape, the other two groves will be evaluated in the same manner. The final output will be an SVG file with the final landscape.

In terms of how the parse allots and manages different variable types, the parser can interpret the input since there are certain key characters that act as separators for each variable. The ":" character separates the Season from the Forest, so when a ":" is read, the parser knows the season is on the left-hand side and the forest is on the right-hand side. Once the Season is parsed, the "," character acts as a separator in the Forest List, denoting when the instructions for one grove ends and another beings.

7 Remaining Work

Currently *Forest++* brings life and joy with its brightly colored foliage in spring or fall attire! However, the randomization mechanism does not account for overlap. So when creating some forests, a grove of trees may simply cover others depending on the order of groves! Further improvements would be minimizing this overlap. This task, if done naively, could raise the computational power to n^2, since in order to check overlap, each tree would have to compared to the next tree. It would also require a list of locations to be stored and continually updated at the creation of more trees. This would be slightly challenging, but would enhance the visual aesthetic of *Forest++* greatly.