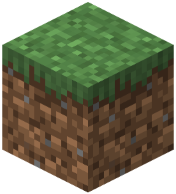
GENERATING CASTLES FOR MINECRAFT USING Haskell

TIM WILLIAMS
OCTOBER 2019

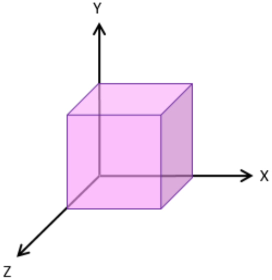
THE BASIC IDEA

- A Domain-specific language (DSL) that targets Minecraft "mcfunction" files and "setblock" commands.
- A *compositional* language that makes it easy to assemble complex structures from simple ones.
- A *shallow embedding* inside Haskell, leveraging Haskell's expressiveness and abstractions.



A DOMAIN-SPECIFIC LANGUAGE

- DSLs offer naming, semantics and abstractions that match the problem domain.
- This one is hopefully usable by anyone familiar with basic functions and 3D Cartesian coordinates.



DATA TYPES

- The basic atom in Minecraft is the block.
- All blocks have coordinates and a kind (e.g. air, cobblestone, water).
- Coordinates assumed to be relative.

```
data Block = Block
    { _blockCoord :: Coord
    , _blockKind  :: String
    }

data Coord = Coord { _x :: Int, _y :: Int, _z :: Int }
    deriving (Ord, Eq)

makeLenses ''Coord
makeLenses ''Block
```

- Minecraft structures are represented as an ordered list of blocks.
- Use a newtype to hide the underlying representation.

```haskell
newtype Blocks = Blocks { unBlocks :: [Block] }
    deriving (Semigroup, Monoid, Show)

mkBlocks :: [Coord] -> Blocks
mkBlocks = Blocks . map (\c -> Block c cobblestone)

-- | A block of nothing (air) at the origin (0,0,0)
zero :: Blocks
zero = Blocks [Block (Coord 0 0 0) air Nothing]
```

We set the kind of block using an infix `#` operator:

```haskell
-- | Set the kind of all blocks
infixr 8 #
(#) :: Blocks -> Kind -> Blocks
(#) blocks k = mapKind (const k) blocks

mapKind :: (Kind -> Kind) -> Blocks -> Blocks
mapKind f = mapBlocks $ over blockKind f

mapBlocks :: (Block -> Block) -> Blocks -> Blocks
mapBlocks f = Blocks . map f . unBlocks
```

A NON-COMMUTATIVE MONOID

- Blocks are combined using a monoid instance, derived using the underlying list instances.
- The `Blocks` monoid is *non-commutative*, the right-hand-side overrides the left.

```
zero <> (zero # cobblestone) -- results in a cobblestone block at (0,0,0)
(zero # cobblestone) <> zero -- results in nothing (an air block) at (0,0,0)
```

LENSES FOR DIMENSIONS

- Abstract over dimensions using lenses.
- Any function that requires both reading and updating a dimension needs only one parameter.

```
type Dimension = Lens' Coord Int
```

```
view :: Lens' a b -> a -> b
over :: Lens' a b -> (b -> b) -> a -> a
set  :: Lens' a b ->       b  -> a -> a
```

REPETITION AND LAYOUT

To build composite structures, we use combinators that provide us with repetition and layout:

```haskell
-- | Repeat structure 'n' times with function 'f' applied iteratively.
repeat :: (Blocks -> Blocks) -> Int -> Blocks -> Blocks
repeat f n = mconcat . take n . iterate f

-- | replicate structure 'n' times with a spacing 's' in dimension 'd'.
replicate :: Dimension -> Int -> Int -> Blocks -> Blocks
replicate d s = repeat (move d s)

-- | Move blocks by 'i' in dimension 'd'.
move :: Dimension -> Int -> Blocks -> Blocks
move d i = mapBlocks $ over (blockCoord . d) (+i)

-- | Translate blocks by the supplied 'x, y, z' offset.
translate :: Int -> Int -> Int -> Blocks -> Blocks
translate x' y' z' = move x x' . move y y' . move z z'
```

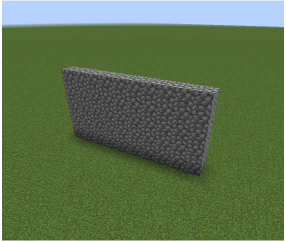
WALLS AND FLOORS

```haskell
-- | Create a line of cobblestone blocks with length 'n' along dimension 'd'.
line :: Dimension -> Int -> Blocks
line d n = replicate d 1 n zero # cobblestone

-- | A wall of cobblestone with width 'w', height 'h', along dimension 'd'.
wall :: Dimension -> Int -> Int -> Blocks
wall d w h = replicate y 1 h $ line d w

-- | A wooden floor with lengths 'lx' and 'lz'.
floor' :: Int -> Int -> Blocks
floor' lx lz
    = replicate x 1 lx
    . replicate z 1 lz
    $ zero # oak_planks
```

wall x 9 4



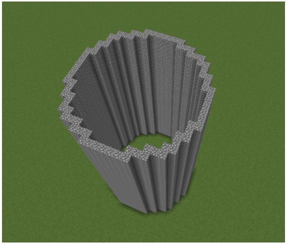
CIRCLES

```haskell
-- | A circle of radius r in the plane formed by dimensions (d, d'),
-- centered on the origin.
circle :: Dimension -> Dimension -> Int -> Int -> Blocks
circle d d' r steps =
    mkBlocks [ set d x . set d' z $ Coord 0 0 0
             | s <- [1..steps]
             , let phi = 2*pi*fromIntegral s / fromIntegral steps ::Double
                   z   = round $ fromIntegral r * cos phi
                   x   = round $ fromIntegral r * sin phi
             ]
```

CYLINDERS

```haskell
-- | A hollow cylinder of radius r in the plane formed by dimensions (d, d')
-- and with length along dl.
cylinder
    :: Dimension -> Dimension -> Dimension -> Int -> Int -> Int
    -> Blocks
cylinder d d' dl r h steps =
    replicate dl 1 h (circle d d' r steps)
```

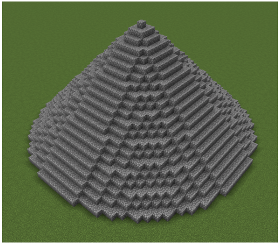
```haskell
cylinder x z y 10 40 500
```



CONES

```haskell
-- | An upright hollow cone in the (x,z) plane, with radius r and height h,
-- centered on the origin.
cone :: Int -> Int -> Int -> Blocks
cone r h steps = mconcat
    [ move y y' $ circle x z r' steps
    | y' <- [0..h]
    , let r' = round $ fromIntegral (r*(h-y')) / (fromIntegral h::Double)
    ]
```

cone 20 20 1000



SPIRALS

```haskell
-- | An upward spiral in the (x,z) plane with radius r and height h
-- using rev revolutions, centered on the origin.
spiral :: Int -> Int -> Int -> Int -> Blocks
spiral r h revs steps =
    mkBlocks [ Coord x y z
             | s   <- [1..steps]
             , let phi = 2*pi*fromIntegral (revs*s) / fromIntegral steps ::Double
                   z   = round $ fromIntegral r * cos phi
                   x   = round $ fromIntegral r * sin phi
                   y   = round $ fromIntegral (h*s) / (fromIntegral steps::Double)
             ]
```

```haskell
-- | A spiral staircase in the (x,z) plane with radius r, thickness t
-- and height h using rev revolutions, centered on the origin.
spiralStairs
    :: Int -> Int -> Int -> Int -> Int
    -> Blocks
spiralStairs r t h revs steps = mconcat
    [ spiral (r-i) h revs steps
    | i <- [0..t-1]
    ]
```

spiralStairs 10 12 80 6 1000

GRID LAYOUTS

A grid layout combinator is particularly useful, especially for castles.

```
grid :: Int -> [[Blocks]] -> Blocks
grid spacing = f z . map (f x)
  where
    f :: Dimension -> [Blocks] -> Blocks
    f d = foldr (\a b -> a <> move d spacing b) mempty
```

RENDERING

Finally, we need a "render" function for generating the command file:

```
data CoordKind = Relative | Absolute

render :: FilePath -> String -> String -> CoordKind -> Blocks -> IO ()
render minecraftDir levelName functionName coordKind (prune -> blocks) = ...
```

Scaling up to castles

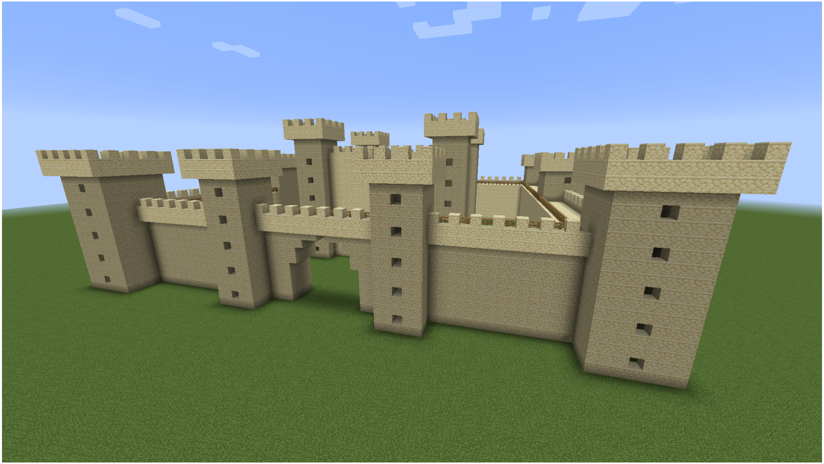
- Castles are just monoidal compositions of the aforementioned components.
- Start with abstract components. e.g. solidCircle, then make more concrete specific variants, e.g. circularFloor.
- Higher-order functions useful to parameterise components, e.g. the style of turret.
- Components are more reusable when sizes have been parameterised, e.g. widths, lengths, radii.

```haskell
englishCastle :: Blocks
englishCastle = mconcat
    [ castleWall 100{-width-} 10{-height-}
    , grid 50 {-spacing-}
        [ [ t,  t,  t]
        , [ t,  k,  t]
        , [ t,  g,  t] ] ]
  where
    t  = circularTurret 4{-radius-} 15{-height-} 20
    t' = circularTurret 3{-radius-} 15{-height-} 20
    k  = castleKeep t' 24{-width-}  15{-height-}
    g  = move x (-12) t <> move x 12 t -- gatehouse entrance
```

CASTLES / MOSSY ENGLISH

CASTLES / GERMANIC

CASTLES / DESERT



THAT'S ALL FOLKS!

The slides for this talk will be available at:
http://www.timphilipwilliams.com/slides/minecraft.pdf

The original blog post with source code:
http://www.timphilipwilliams.com/posts/2019-07-25-minecraft.html

For anyone that wants to collaborate, the combinators have been donated to this project:
https://github.com/stepcut/minecraft-data