Heroscape 2.0

A case study in using Scala for complex HTML document generation

Ben Hutchison
Senior Developer, REA Group
Coordinator, Melbourne Scala User Group

It all started last Christmas...

- ...when I bought my kids a miniatures wargame called Heroscape
- Fortunately
  - They loved it
  - While my younger son Felix uses them like dolls for imaginative play,
  - my older son Otto wanted to play it using the "proper rules"
- Unfortunately
  - Heroscape's game mechanics make for a **boring game:**
    - Meaningless choices
    - All games end up the same
  - Otto insists on playing it with me anyway
- Game design is a hobby of mine
  - So I set out the design my own (hopefully better) Heroscape MkII

Heroscape MkII

- The plan
  - Retain the **Figures** and **Terrain**
  - Replace the **Unit Cards** that specify game mechanics with my own custom printed cards
  - Use a modified version of the turn-based game mechanics I developed previously for mobile-phone RPG Arcadia
- Thus, a challenge:
  - How to (re)generate nice looking units cards displaying photo and stats for 30+ different unique units/characters
- Technologies considered
  - Code-driven OpenOffice document templates
  - PDF Generation via IText library
  - HTML

HTML Document Generation

- Entering familiar territory, except that
  - Havent done it from Scala before
  - Since cards are printed from screen, no need to ensure cross-browser compatibility
    - Anything supported by Firefox 4.x is on the menu: HTML 5, CSS 3, Mozillla extensions
- Model: Collection of UnitType objects, specifying
  - Name, Photo, Life Points, Attack and Defense stats, other abilities
- View: UnitType => HtmlDom
  - Scalate templates?
  - Custom-built solution: TagTree library

Scalate Templating Engine?

- Model Objects + Templates => String
  - Multiple template dialects, including
    - JSP-like (SSP)
    - Haml-like (SCAML)
  - developed by "Groovy creator" James Strachan (Scala's "Kim Philby"?)
  - Framework agnostic, use standalone or integrated
  - Well documented
  - Reliable
  - Concise templates
  - Imperative execution of template
    - "Start at the top and work down the page"

Why not Scalate

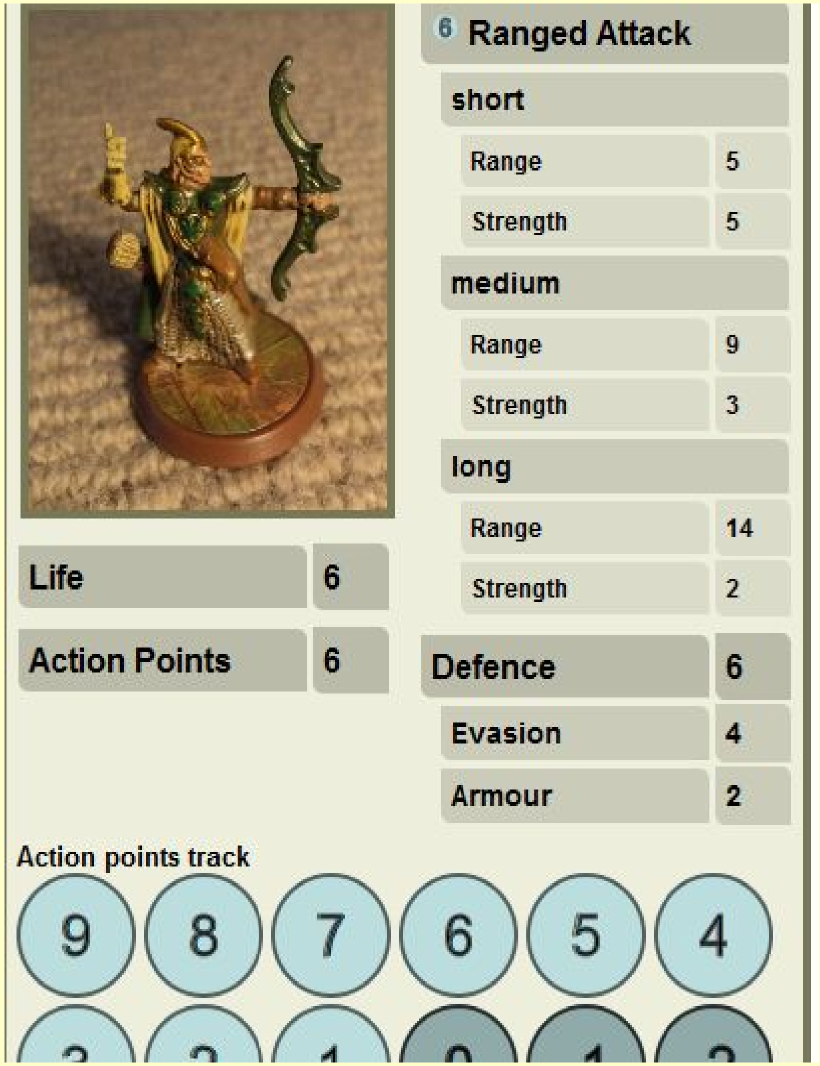
- Used Scalate for first prototype, but wasn't comfortable
  - Template DSL is Yet Another Language to learn & debug
  - Unclear semantics when mixing logic into template
  - Slow execution because templates must be precompiled
- I wanted a more functional model...
  1. Specify how small pieces of the model transform into small pieces of HTML
  2. Compose small pieces into big pieces
- Scala looked appealing
  - Easier to define "small pieces" as methods than it is to create another template file for each piece
  - Execution behaviour of Scala well defined and familiar
    - No extra language to learn

Enter TagTree

- Very simple DSL for creating HTML from Scala code
  - Write the view directly in Scala
- Immutable & Side-effect free
  - Requires ++ operator to glue sibling tags together
- Tree structured
  - Internally represented as scalaz.Tree[NodeInfo]
  - Wrapper over that provides HTML-specific behaviours, eg marginPx(px: Int)
  - Pass child nodes up to parent

HTML 5 / CSS 3 Goodies

- Vector Graphics via embedded SVG inside HTML
  - Used to draw the circles on the game cards
- Rounded Corners in CSS
  - Each corner can be controlled individually
- Columns in CSS
  - Renderer distributes page content over columns



6 Ranged Attack

short	
Range	5
Strength	5
medium	
Range	9
Strength	3
long	
Range	14
Strength	2

Life	6
Action Points	6

Defence	6
Evasion	4
Armour	2

Action points track

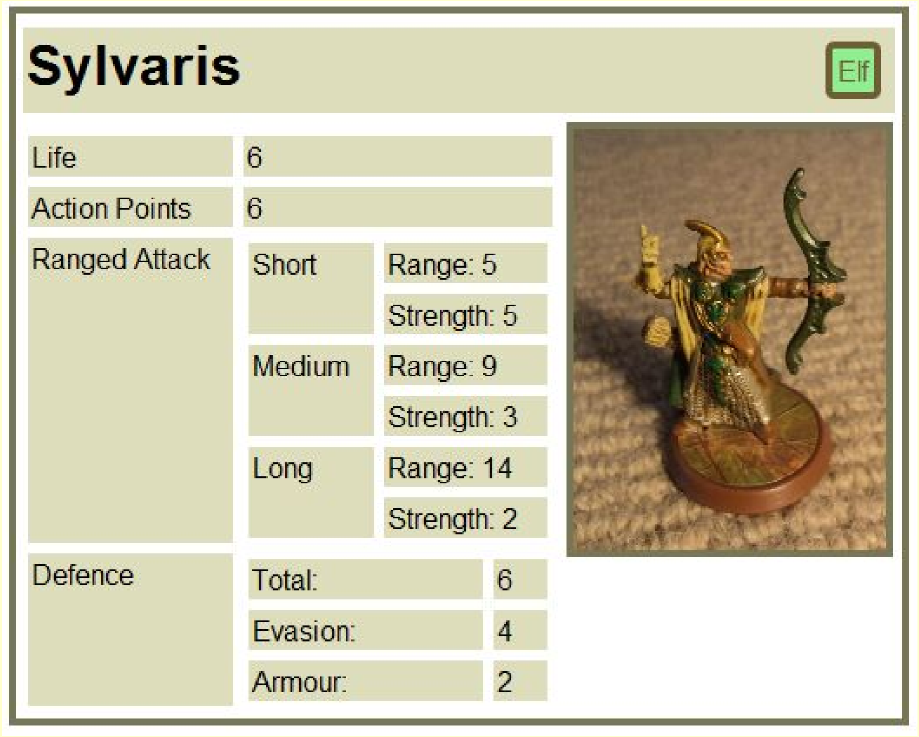
9 8 7 6 5 4

The Card Layout Problem

- Goal: Cards “look nice”
- Different units have different stats and abilities
  - Only some units have Ranged Attacks, which have different strenths at different ranges
  - Wanted units to have arbitrary special abilities, which might have their own stats
- I'm a classic “lazy programmer”
  - Didn't want to hand-customize card layout for different units
  - Searched long and hard for a general solution

2nd Prototype: Nested tables

- Tabular grid with internal subdivision
- Assumptions:
  - Cards consist of a Seq[Option[Stat]]
  - A Stat is one of several known cases. Each case knows how to convert itself into HTML
    - Simple numerical stat, eg life points
    - Complex multi-part stat like ranged attack



Sylvaris Elf

Life	6	
Action Points	6	
Ranged Attack	Short	Range: 5
		Strength: 5
	Medium	Range: 9
		Strength: 3
	Long	Range: 14
		Strength: 2
Defence	Total:	6
	Evasion:	4
	Armour:	2

An aside: Shading *leaf* table cells

- Prototype 2 yields a table with internal sub-tables
  - Though it would "look nice" to shade only the leaf cells of the table
  - Creates a mosaic effect
- Wanted a CSS selector that finds leaf table cells
  - *All TD elements that do not contain a child TD element*
  - After a long search, concluded it is inexpressible using any form of CSS at 3.0 level
- Recall that TagTree presents DOM as a Tree[NodeData]
  - Can write a tree transform, (Tree[NodeData])=>Tree[NodeData], that adds a class attribute to every leaf TD cell
  - Get some more practice in functional programming along the way
  - Isn't that fun!

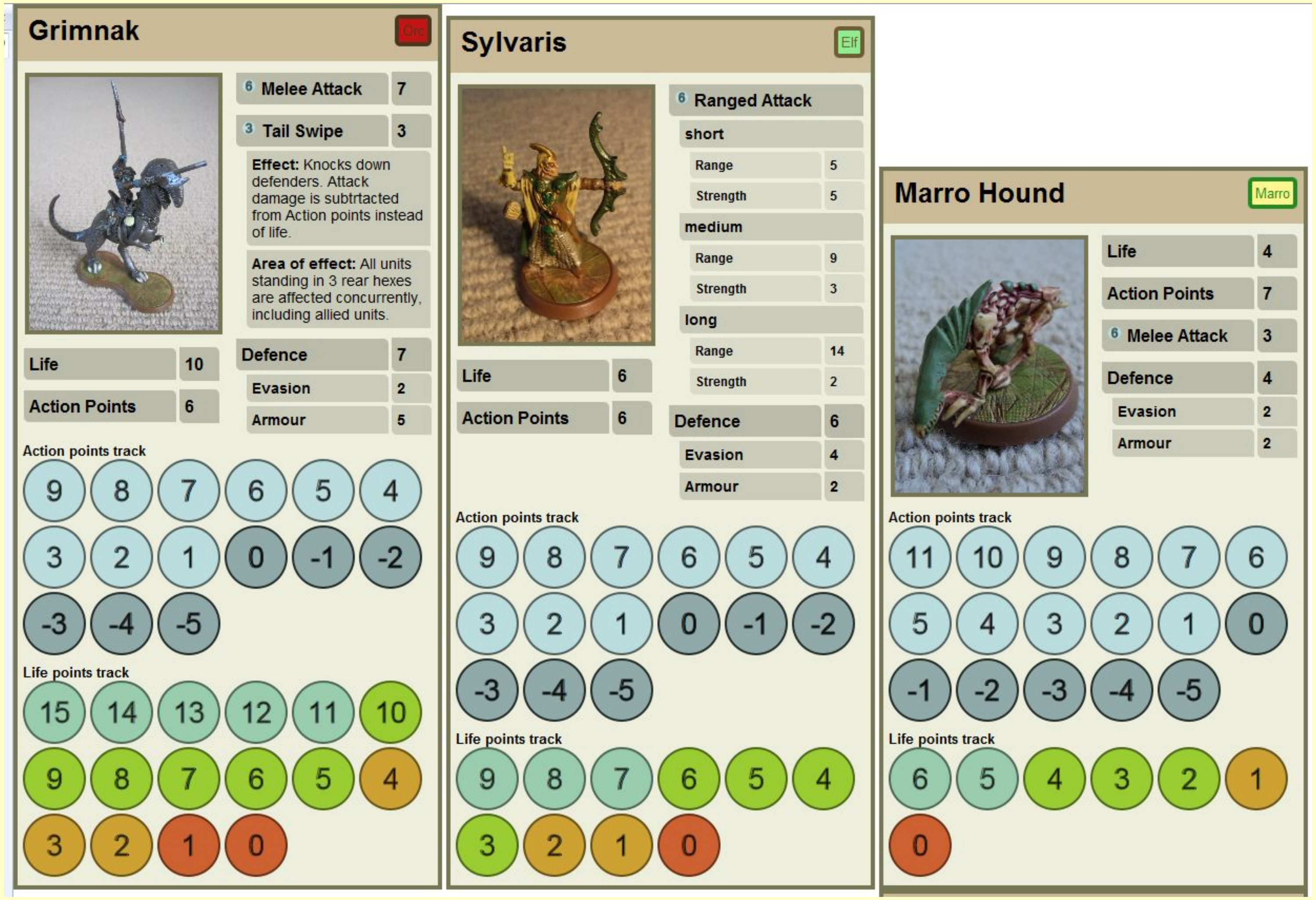
Shading *leaf* table cells 2

- A tree transformation consists of
  - Transform applied to each node of the tree
  - An order in which the transform is applied to the nodes
- To identify a TD tag which has no TD children
  - We must first have visited the children, so traversal order must be *bottom up*
  - We must encode the presence of child TD tags into the transform result as a boolean "isLowest" flag
    - Tree[NodeInfo] => Tree[(NodeInfo, Boolean])]
- Afterwards, we're left with Tree[(NodeInfo, Boolean])]
  - Need to throw away the now unneeded "isLowest" flag
  - Enter **unzip**: Tree[(A, B)] => (Tree[A], Tree[B])

From the Specific to the General…

- Turns out that our bottom-up tree transformation generalizes to **scanr**
  - **def** scanr[B](g: (A, Seq[Tree[B]]) => B): Tree[B]
  - the **r** means scan from the right, or end, of the collection
- scanr, and its top-down counterpart **scanl**, can be viewed as "like **fold**, but accumulates intermediate results"
- with **fold** being a fundamental *traversal* or *aggregation* operation in functional programming
  - def foldLeft[B](z: B)(op: (B, A) => B): B
  - def foldRight[B](z: B)(op: (A, B) => B): B
  - The function **op** visits every element of the collection
  - Unlike **map**, the output value of **fold**, of type B, can have completely different shape/structure

3rd Prototype: Indented Layout



Grimnak Orc

6 Melee Attack	7
3 Tail Swipe	3

Effect: Knocks down defenders. Attack damage is subtrtacted from Action points instead of life.

Area of effect: All units standing in 3 rear hexes are affected concurrently, including allied units.

Life	10
Action Points	6
Defence	7
Evasion	2
Armour	5

Action points track

9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5

Life points track

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Sylvaris Elf

6 Ranged Attack	
short	
Range	5
Strength	5
medium	
Range	9
Strength	3
long	
Range	14
Strength	2
Life	6
Action Points	6
Defence	6
Evasion	4
Armour	2

Action points track

9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5

Life points track

9 8 7 6 5 4 3 2 1 0

Marro Hound Marro

Life	4
Action Points	7
6 Melee Attack	3
Defence	4
Evasion	2
Armour	2

Action points track

11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5

Life points track

6 5 4 3 2 1 0

3rd Prototype: Indented Layout

- Back to the main story
  - Recall that in the prototype 2: “A Stat is one of several known cases. Each case knows how to convert itself into HTML“
- I wanted *yet more generic* layout
  - Add new stats without worrying about how they will look
  - Table subdivision not nestable beyond 3 levels
- Idea: Coerce all stats to a *standard* but *extensible* structure
  - type StatTree= Tree[(**Seq[TagNode]**, **Option[Seq[TagNode]]**)]
    - Label HTML shown green
    - Optional value HTML shown blue
  - foreach stat define: (stat) => StatTree
  - Define one generic render transform: (StatTree) => HTML

API Enrichment

- Java interfaces are often, by convention, minimal
  - java.util.List: ~ 25 methods
  - scala.collection.Seq: ~200 methods
  - Root cause: lack of traits in Java
- It is routine to *enrich* Java APIs in Scala with convenience methods via implicit conversions
- Example: java.awt.Color enriched by RichColor, adding
  - hue, saturation, brightness
  - withHue, withSaturation, withBrightness
  - brightenPercent, darkenPercent

Cofree: What's common to Lists & Trees

- In discussions on the Scalaz list during the project, I discovered *Cofree*
  - trait Cofree[+F[+_],+A]
  - "A stream of some functor F[A]"
- Abstracts out the commonality between Lists and Trees
  - They have two ends
  - They store a value in each node
  - They have a variable *branching factor* at each node
    - Lists have 1 branch at each node
    - Binary trees have 2 branches at each node
    - HTML trees have N branches at each node

Cofree for customized Trees

- I have tried to reuse the scalaz Tree trait in TagTree, but..
  - One major problem: scalaz stores a node's children as a *Stream* rather than a *Seq* or *Iterable*
    - Prevents use of other data structures like linked-lists or immutable Vectors

```
sealed trait Tree[+A] {
  /** The label at the root of this tree. */
  def rootLabel: A
  /** The child nodes of this tree. */
  def subForest: Stream[Tree[A]]...  }
```

- Cofree[Seq, A] might offer a solution?
- Subtype- vs Parametric- polymorphism?