Ian’s notes

Database Schema

This is super-simple:

- The usual `User` and `RMCookie` stuff for login management.
- A `Module` table for module definitions (at the moment, just a title, the module definition as text, and the module owner). Should eventually be expanded to include a `VERSION` column (see below for a bit more about this).
- A `ModuleActivation` table recording surveys scheduled for a given user. This is the table that records the hash linking a user and a module activation.
- A `ModuleData` table that holds survey results. This has columns to record the module activation hash, the module ID, the name of the question (in the module DSL, questions are always given as `name = BlahQuestion parameters` and these are the names that appear in this table) and a textual answer value.

Module DSL

- **IMPORTANT** Need to get the error handling for the module DSL really right: there should be *no* Javascript errors!
- Low-level syntax: the current synyax doesn’t use layout; instead it’s kind of a “wordy” language where the presence of keywords gives enough information to produce an unambiguous parse. I hadn’t quite decided whether this was the best way to do things, but I think it might be more effective for allowing non-programmer users to write modules than having an offside rule approach. “Wordiness” also makes life easier if you have macros.
- Some things to add to expression syntax: `case ... of` selection, `let ... in` or `where` for local names in more complicated expressions.
- Question types: at the moment, these are individual constructors of the `Question` data type, mostly to keep things simple as I started implementing them, but that’s not a very extensible way of doing things. I hadn’t settled on a really good way of representing these things.
- Development plan (such as it is):

V0.1 Pretty much done, except for module metadata and a distingushed "activation" component for modules:

1. Basic syntax: literals (numeric, string, boolean, arrays, records); identifiers; simple expressions
2. Module definitions: framework, metadata, component contents, distinguished "activation" component
3. Function definitions: pure call-by-value only
4. Fixed component definitions: `SurveyPage`, 3-4 `Question` types
5. Data collection infrastructure

V0.2 Not started:

1. Add more question types
2. Parameterised modules
3. ???

- Needs to be multilingual from the start:
  - Define default language at top of module: `language=none` means "text" values are IDs in message files; `language=de`, `language=de_CH` work as you'd expect.
  - Need some sort of textual interpolation for inserting numbers, pluralisation, etc. (shakespeare-i18n?).
  - Message files as part of the module bundle: either going from an ID to a natural language string, or from language A to language B.
  - Should have a utility in the "Module Mall" to build a null translation file from a module by pulling out all the unique text elements, and for checking translation files against module text elements.
- Things that are needed that I've thought about but haven't started:
  - LimeSurvey import (from XML format).
  - Containers for pages with choice and sequencing – basically, this would be another kind of `TopLevel` called `Container` or something that you could name. I've not thought very much about how to make sequencing and choice work in the kind of declarative framework I've been using, but I don't think it would be that hard to do.
  - Modularisation: registration and naming of modules, referencing and linking modules via imports.

  - Module metadata: this could either be done as part of the “module options” or using some other syntax.
  - Other module types: not really any idea how to do this – embedding of items defined in other ways into an `IFRAME` container or something like that? Really need to assess the feasibility of embedding to know whether we need to support a richer model in the DSL.

- Rendering ideas:
  - How about defining survey elements in terms of basic UI elements, so there is a Core module language with more complex elements done via desugaring?
  - Example:
    - Full DSL: `smoker = YesNo "Do you smoke?"`
    - Core DSL:

      ```
      smoker = Div.yesno-question
         Text.choice-label "Do you smoke?"
         Choice "Yes" -> True
                "No"  -> False
      ```
  - The processing flow for rendering a module would then be something like this:

```
[Module] -> (Parse) -> [ModAST] -> (Check) -> (Desugar) -> [CoreAST] -> (Translate) -> (Render) -> [HTML+CSS+JS]
               |                      |                                      |
               v                      v                                      v
         Syntax errors          Unique names                          (given target
                               Name resolution                         language and
                                                                    translation files)
```

Module definitions

Need some way of identifying the “main” top-level component to be activated when a module is loaded (probably just make it so that the survey page called “main” is activated). The point here is that there can be multiple top-level components (survey pages, for example) within a single module, and eventually it should be possible to change the “activated” top-level component as a result of user selections.

Server code

- The email/password authentication works more or less the same as in BayesHive, and it makes the same assumption: that it can transmit passwords in plaintext because the site will only ever be accessed via HTTPS.

- The same `Angular.UIRouter` code as in BayesHive is used for serving up Angular `ui-router` pages.

- I've written some little stuff (in `Foundation.hs` and `angular/shared/alerts.julius`) to integrate Yesod's messages (Ãă la `setMessage`) with Bootstrap alerts. Instead of calling `setMessage`, you call `setAlert` (or `setAlertI` for internationalisation) with an alert type (`Error`, `Warn`, `Info`, `OK`) and a message. The client-side code maintains a list of current alerts and there's an Angular directive called `<alert-list>` you use to put them somewhere on the page.

- In the code to actually run surveys (`getSurveyRunR` in `Handler.Survey`), there are a couple of error cases that in an ideal world wouldn't crop up at all. In particular, it shouldn't be possible to create a module activation for a module that has errors. To make this really true, it will be necessary to have module versioning, so that it's possible to edit a module definition while still having module activations referring to a "known good" module definition. In the presence of module imports, it will be necessary to keep track of the correct versions of imported modules to use to go with a given version of a module. This won't be hard to do: have a `VERSION` column in the module definition table that contains the version value for "published" modules, or NULL for any edited version of a module that hasn't yet been made available for use, along with a separate table to record the version dependencies with entries saying "Version X.X of module A depends on version Y.Y of module B". That way, resolving module versions for imports is just a matter of traversing this dependency graph, and you can have different modules depending on different historical versions of other module at the same time, without any interference.

Original notes (a bit braindumpy...)

Basic syntax

Layout

- Use the same "optional layout" approach as Haskell? I.e. layout converts to `{ ... ; ... ; ... }` on lexing?

- How does this work with macros?

Identifiers

- In general, identifiers begin with a letter and contain only letters, digits, `_` (underscore), `-` (hyphen) or `.` (period). Regular expression: `/[a-zA-Z][a-zA-Z0-9_-.]*/` (actually slightly different because of Unicode letters).
- Module identifiers are a series of period-separated identifiers, each of which begins with a letter. Regular expression: `/[A-Za-z][a-zA-Z0-9_-]*(\.[A-Za-z][a-zA-Z0-9_-]*)` (again with the letters...).
- Namespace identifiers follow the same rules as module identifiers.
- Some identifiers are reserved as keywords.
- Keywords are case-insensitive. User-defined identifiers are case-sensitive. (I've not actually implemented this properly yet.)

Literals

- Numeric literals: `123`, `1.4`, `-0.5E-6`, `25%`
- String literals: `"abc"`, `"Hello, ""Bob""!"` (the only escape sequence is `""` for double quote)
- Boolean literals: `true`, `yes`; `false`, `no` (all options equivalent when boolean value required).
- Vector/array literals: `[ 1, 2, 3 ]`, `[ true, false, false ]` (heterogeneous). Equivalent to `[ 0 => 1, 1 => 2, 2 => 3 ]`, i.e. treated the same as maps/records.
- Map/record literals: `{ a = 1.2, b = "abc", XC5 = false }` – this is a bit more restricted than ideal: it should be possible to use identifiers, strings or numeric values as keys.

Expressions

The usual kind of thing:

- Numeric operators: `+`, `-`, `*`, `/`, `^`
- Comparison operators: `==`, `/=`, `<=`, `>=`, etc., plus case-insensitive versions, like `@==`, `@/=`, etc.
- Boolean operators: `and`, `or`, `not`

- Function applications: `f(1, 2)`, `myFunc(x, "abc")`, `floor(x)`, etc.
- Conditional expressions: `if a then 1 else 2`, etc.

Definitions

- General language features:
  - Functions: call-by-value or call-by-reference? Allow both, but require call-by-reference to be explicitly flagged?
  - Macros: need to think about these – some sort of backquote-like templating scheme?
  - Modules: what can go in a module? what happens when a module is activated? (Distinguished `main` component?)
  - Enumerations: basically just a way to map between identifiers and some numeric or textual representation.
- Application-specific features:
  - Components: `SurveyPage`, `Question`, etc.
  - Assets: images, video, audio.
  - Translations: base language to other languages; templating/functional approach for pluralisation and word order issues.
  - Help/description text: message catalogue approach? Plays nicely with translation.
  - Module metadata: simple key/value block (like Cobolo `DESCRIPTION SECTION`!).

Namespaces

- *Modules* live in *namespaces*, e.g. `base`, `openPsych`, `openPsych.Tests`, `ClinicA`, etc.
- Within each namespace, module names can be considered hierarchically, e.g. `SubstanceAbuse.MoodSurvey`, but this is not necessary.
- Naming: `<namespace>:<module>` gives namespace explicitly, e.g. `ClinicA:Utils.Logo`, `openPsych:Tests.StroopTest`; without namespace, a module name references the first matching module in a namespace search path.
- A namespace is essentially a record containing modules; a module is essentially a record containing definitions of various kinds.

- Defining modules:

  ```
  namespace openPsych           -- Defines the namespace for all
                                -- further definitions in the file
  module Simple.MoodSurvey
    ...
  ```

- Setting namespace search path:

  ```
  use namespace ClinicA as A
  use namespace openPsych
  ```

  or

  ```
  use namespace ClinicA as A, openPsych as oP
  ```

- Namespace `base` available by default.
- Namespaces searched in order given in `use namespace` declarations.
- Aliasing with `use namespace ... as ...`.
- Parameterisation over namespaces:

  ```
  module Simple.MoodSurvey(styleNamespace)
    use namespace styleNamespace
    ...
  ```

  which can then be used as

  ```
  use namespace ClinicA
  activate Simple.MoodSurvey(ClinicA)
  ```

Survey element types

In LimeSurvey, questions must be members of *question groups*. Normally all questions in a group are displayed together and the groups are displayed one after another. It's possible to assign groups to "randomisation sets", within which the group ordering is randomised each time the survey is used. It's also possible to show and hide groups based on responses to earlier questions.

There is an "Expression Manager" that allows you to interpolate expressions based on results of earlier questions into the text of later questions, to control whether or not a question appears based on Boolean conditions, and so on.

LimeSurvey question types:

```
4.1 Arrays
 4.1.1 Array
```

```
   General "sub-questions with discrete choices" setup.  All the
   others are specialisations of this.  So, provide a general
   parameterisable array question type and then implement all the
   others as specialisations of this.
 4.1.2 Array (5 point choice)
 4.1.3 Array (10 point choice)
 4.1.4 Array (Yes/No/Uncertain)
 4.1.5 Array (Increase/Same/Decrease)
 4.1.6 Array by column
 4.1.7 Array dual scale
 4.1.8 Array (Numbers)
 4.1.9 Array (Text)
4.2 Mask questions
  Specialised "don't fit anywhere else" question types.
 4.2.1 Date
 4.2.2 File upload
 4.2.3 Gender
 4.2.4 Language switch
 4.2.5 Numerical input
 4.2.6 Multiple numerical input
 4.2.7 Ranking
 4.2.8 Text display
 4.2.9 Yes/No
 4.2.10 Equation
4.3 Multiple choice questions
  Really "multiple" choice: checkboxes.
 4.3.1 Multiple choice
 4.3.2 Multiple choice with comments
4.4 Single choice questions
  Radio boxes and lists.
 4.4.1 5 point choice
 4.4.2 List (Dropdown)
 4.4.3 List (Radio)
 4.4.4 List with comment
4.5 Text questions
   Basically just different sizes of text entry.
 4.5.1 Short free text
 4.5.2 Long free text
 4.5.3 Huge free text
 4.5.4 Multiple short text
```