Language Workbench Challenge 2013

Key challenges this year:

- Modularity: language reuse and language referencing.
- Analysis and verification: advanced static checking of domain-specific models.

Questions and comments to Tijs van der Storm storm@cwi.nl.

The assignment: Questionnaire Language (QL)

Forms-based software for data collection has found application in various areas, including scientific surveys, online course-ware and guidance material to support the auditing process. As an overall term for this kind of software applications we use the term "questionnaire". In this LWC'13 assignment the goal is to create a simple DSL, called QL, for describing questionnaires. Such questionnaires are characterized by conditional entry fields and (spreadsheet-like) dependency-directed computation.

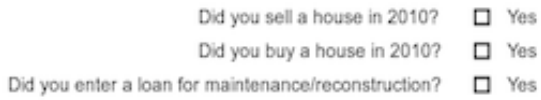
Example

The following example presents a possible textual representation of a simple questionnaire.

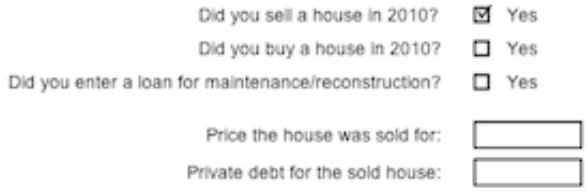
```
form Box1HouseOwning {
   hasSoldHouse: “Did you sell a house in 2010?” boolean
   hasBoughtHouse: “Did you by a house in 2010?” boolean
   hasMaintLoan: “Did you enter a loan for maintenance/reconstruction?”
boolean
   if (hasSoldHouse) {
     sellingPrice: “Price the house was sold for:” money
     privateDebt: “Private debts for the sold house:” money
     valueResidue: “Value residue:” money(sellingPrice - privateDebt)
   }
}
```

This simple form should generate into a GUI which allows the following user interaction:

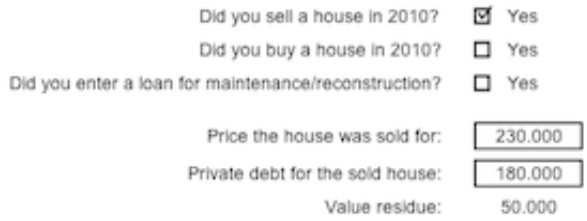
Step 1



Step 2



Step 3



The key things to observe:

- Questions become available in the GUI as soon as their enabling conditions are satisfied.
- Default styling and widgets is implied. For instance, boolean questions produce checkboxes, numeric fields are rendered as text-boxes, computed values are read-only.

Elements of QL

The snippet of fictional QL above is textual only to provide an example. QL should be easy to express as a graphical language as well. The main ingredients of QL can be summarized as follows.

Syntax

QL consists of *questions* grouped in a top-level *form* construct. First, each question *identified* by a name that at the same time represents the result of the question. In other words the name of a question is also the *variable* that holds the answer. Second, a question has a *label* that contains the actual question text presented to the user. (Note that technically this is a presentation issue that could be in a separate language for layout and styling, but to make QL standalone we need it here. See below for more on the layout language.) Third, every question has a *type*. Finally, a question can optionally be associated to an *expression*: this makes the question *computed*.

A questionnaire consists of a number of questions arranged in sequential and conditional structures, and grouping constructs. Sequential composition prescribes the order of presentation. Conditional structures associate an *enabling condition* to a question, in which case the question should only be presented to the user *if and when the condition becomes* true. The expression language used in conditions is the same as the expressions used in

computed questions. Grouping does not have any semantics except to associate a single condition to multiple questions at once.

For expressions we restrict ourselves to booleans (e.g., `&&`, `||` and `!`), comparisons (`<`, `>`, `>=`, `<=`, `!=` and `==`) and basic arithmetic (`+`, `-`, `*` and `/`). The required types are: boolean, string, integer, date and decimal and money/currency. NB: this set could be extended with other data types, such as an enumeration data type (e.g., "good", "bad" and "don't know"), or integer range (e.g. 1..5). The only requirement is that the data type can be automatically mapped to a widget.

Notes:

- If possible, the expression language for conditions and computed values should be reused.

Semantics

The output of a QL description should be a simple GUI program that shows questions as soon as they become enabled. (So the initial view should consist of all questions that do not have a enabling condition). The user should be able to fill in answers, to which the system responds with more questions to be filled in and/or with the display of additional computed results. After all questions have been filled in --- the fixed point has been reached --- the result of the complete questionnaire should be saved somehow (e.g., as XML, YAML, JSON, etc., or in a database).

The semantics of expressions and question variables is standard, except that an unbound variable (= unanswered question) has the special value `undefined`. Any expression referring to an undefined variable is undefined itself. In conditional context, `undefined` means `false`.

Rendering a QL questionnaire as a GUI should make sensible defaults for aspects of presentation. For instance, the widget used for a question can be derived from the type: checkboxes/radiobuttons for booleans, text fields for strings, spinlocks for integers etc.

Optional: layout and styling (QLS)

The base version of QL is rendered using default heuristics for the presentation aspect of a questionnaire. In this optional assignment the goal is to create a separate layout and styling language. This language can be used to group related questions in pages (with navigation links), sections and subsections, select specific font styles and colors for labels, and choose widgets other than the default choice. For instance, a boolean can be represented by a check box, two radio-buttons, a drop-down list etc. A (bounded) integer could be represented either as a text-field or as a slider. Dates could map to a date picker widget. Etc.

The QLS language *references* the QL language: the names of questions are used to identify the points where to apply a particular styling. This should be non-invasive: QLS styling is applied where needed, the rest of a QL program is rendered as if no QLS existed. In other words: the QLS language is optional for the assignment, but also optional for use.

Challenges

- Modularly integrate QLS into QL; the QL language implementation should be independent of QLS.
- Check QLS *against* QL; check that only defined questions are referenced and make sure that the presentation choices made in QLS are compatible with the type of questions in the QL program.

Optional: analysis of questionnaires

Implement one or more of the following analysis tasks:

- Test for cyclic dependencies. For instance, the following snippet should be rejected:

  ```
  if (x) { y: "Y?" boolean }
  if (y) { x: "X?" boolean }
  ```

  The reason is that `y` will only be asked for when `x` is true, but `x` will only get a value when `y` is true. Of course such cyclic dependencies could occur transitively and nested in expressions. Another way of stating this check is: the ordering of questions should be consistent with how the question variables are used in conditions and computed values.

- Type check conditions and variables: the expressions in conditions should be type correct and should ultimately be booleans. The assigned variables should be assigned consistently: each assignment should use the same type.

- Test for determinism. Since the same data item could be assigned in multiple branches of a questionnaire, we have to make sure that at runtime always only a single branch is selected. For instance:

  ```
  if (x) { a: "A?" boolean }
  if (!x) { a: "A?" boolean }
  ```

  Is OK since `x` and `!x` will never both be true at the same time. For more complex conditions this becomes much more involved, requiring satisfiability checkers etc. Note that allowing multiple occurences of the same question (identified by its name) essential turns the questionnaire tree into a graph: they are multiple paths leading to the same question.

  Since a textual representation of such graphs require an identifying naming scheme, and we wouldn't want to have to duplicate questions and labels all over the place, the above example could possibly better be written using a definition construct:

  ```
  def a: "A?" boolean
  ```

```
if (x) { a }
if (!x) { a }
```

Nevertheless, it is still required to ensure that we are not asked the *a* question multiple times. Another way of stating the requirement is: *a question should only be asked once (if ever)*.