Milestone 2

Design Decisions:

Weeding + Type-checking

We do weeding + type-checking in two separate phases. We have one weeding + type-checking phase for header, use files, groups and filters, and one for computations. We also have two separate symbol tables for these two phases. For the former, we weed use statements by type-checking the files they refer to, and then converting them to group declarations which we then append to the list of group files. During this stage, we add variables to the symbol table, and throw an error if we encounter any redeclarations. We also typecheck the variables in group declarations, as well as the parameters to the program in the Header. The symbol table is implemented using a HashMap. We store a hash of the variable (we get the hashcode from the string representation), and store the type and value of it.

Config

Central to our compiler is our Config file (config.conf, which must be included in the same directory as the compiler itself). This file is essential to the type checking and weeding phase, since it defines the types and values that are permitted for each filter and filter-field. The motivation for the Config file is that it provides a way for users to customize the compiler to suit their needs, seeing as not all hospital databases are the same.

The Configuration file syntax works as follows:
FILTER:(“FIELD1”:[Int], “FIELD2”:[String] , “FIELD3”:[val1, val2, val3, val4] ...)
It’s important to note that the enclosing () around the fields can be replaced with {} to indicate that this filter can be looped over with a for each loop.

Our configuration field permits 3 basic types for the filters and fields. Ints, as seen in Field 1, Strings as seen in Field 2, and Value-Lists as seen in Field 3.

It is also important to note that all names of fields, filters, and variables are not case-sensitive. We decided that it makes our compiler more resistant to insignificant human errors (Brendan wishes the same could be said for Haskell and it’s complaints against his PArsers), and easier to reason about since types refer to one thing.

Weeding the Header:

This simply involves adding the parameters and their types to the symbol table. Following our design philosophy of “explicit > implicit”, we determined that it would be better to require that all parameters in the header require a type declaration. We figured that this would make it easier for users to actually understand what they are doing with the parameters they include , if they know exactly what the parameters are supposed to be. We also verify that each filter parameter is of a field that actually exists in the config file.

Weeding the Group Files

Group files, whether declared as a use file, or defined in the program are weeded in the same way. Their field types are checked against the config file, and if they succeed, then are added to the symbol table. Group statement redeclarations are not allowed. We also do not allow recursive redeclarations. Additionally, the values of each of the groups are checked against the field type they have. For example, if we have a group “Id myID = {1,2,3}”, this would type check correctly if 1) Id is a valid field of some filter in the config file, and 2) Id is defined as an Int in the Config file.

We did not deem it necessary to actually determine the particular filter of each group declaration or header parameter at this time, since we allow the user to potentially use the same group declaration in multiple places.

Weeding the Filters

Each filter name is checked against the config file to make sure it exists. If it does, each of its subfields is in turn verified to exist within the config file for that filter. If a subfield is verified to exist, each of the values defined in the .onc file that filter are type checked against the type defined in the config file. Ints and Ranges are type checked against Ints, Strings against strings, and other values against lists of values in the list of allowed values for that particular field.

If all that succeeds for a single field, the next is examined. If an erroneous error is found, we throw an error and exit immediately. We are of the opinion that if a user makes a mistake, they should be notified immediately so they can take corrective behaviour. We believe doing so allows the user to better understand their code.

Weeding the computations

For the second stage, where we do weeding + type-checking for computations, we have separate rules for each of the different types. The types we have are: List, Table, Filters(or constructors or actors), Indexes,Sequences.
Design decisions:

- List definition and table definitions must happen in the top scope. This is because we can only have a new scope inside a foreach and defining a list of events or a table repeatedly inside a foreach and doing so is probably a mistake.MileStone2TopLevelTable.onc
- As per Oncotime specification, barcharts can only be made of tables. Barcharts can only be created in the top level scope, because doing the same barchart in a repeatedly loop is probably a mistake. (MileStone2InvalidEventForList.onc)
- As per Oncotime specification, Printing a length can only be made of tables, timelines can only be of patient filters. MileStone2TimelineOfNonPatient.onc
- Every event in a list must be from the list of events defined in the config file
- Parametrized events are not implemented since we don't know how to do that.
- Foreaches over a list or a table or a sequence must be declared in the top scope. This is because we can only have a new scope inside a foreach and looping over a list of events or a table repeatedly inside another foreach and doing so is probably a mistake.MileStone2ToplevelForeach.onc MileStone2WronglyIndexing.onc
- Loopables (or Filters or constructors or actors) are things that we can make tables about or loop over like or prints properties of:
  `foreach Patient p { // foreach <loopable_name> var`
  These are defined in the config file as things that are defined with curly braces {} instead of square ones. We defined it this way so that it is easily extendible in the future.
- Currently there are only 3 types of loopables: Patient(s), Doctor(s) and Diagnosis. Their names are case insensitive in the program. We can only have tables of, or print properties of the following properties in a loopable:
  patient: ID, Gender, Birthyear, Diagnosis, Postalcode
  doctor: ID, Oncologist
  diagnosis: Name
  (MileStone2InvalidFieldofConstructor.onc, MileStone2NestedForeach.onc)
- We cannot nest loop over the same loopable twice in the same nesting. Moreover, the outermost loop with a loopable must be in the top scope because otherwise someone is looping over a loopable while looping inside

an event or table. Since multiple nesting is not allowed, as per our current config, only 3 levels of nesting are possible.

Scoping

We have scoping for the Computations block. We have a stack of Scopes, which we push and pop off as and when we enter and leave a foreach. Variables that are defined after a line are not defined. We can't have redeclarations inside a scope, but an outside variable can be redeclared in a nested scope. Also the following is allowed:

```
table p = ...

foreach Patient p { //allowed
...
```

This is because a foreach is its own scope, and Patient p already is in a new scope. (also check MileStone2ComputationRedeclarations.onc andMileStone2ComputationUndefinedVariable.onc)

Pretty Printing

For pretty printing, we chose to in-line group files as opposed to printing out the use statements again. When the program is run with `-pptype` flag:

1. For groups, it expands them, and prints them as comments. Note that the groups in the use files are also expanded. We do not sort the groups - we could not think of any meaningful way of sorting them (based on what?), and indeed, why we would even want to.
2. For filters, we fetch default values for the particular filter's fields from the config file and fill up the undeclared fields with a wildcard ('*').
3. For computations, we print out the type of every variable as and when it is used.

Miscellaneous

We throw an error on group file redeclaration, and mention the group variable that is being redeclared. We chose to go with an error instead of warning because "explicit is better than implicit".

We take care of recursive includes for groups. For example:

```
group Id moreIds = {1000, 15 to 30, <moreIds>}
```

Is disallowed. We also disallow mutually recursive groups:

```
group Id moreIds = {1000, 15 to 30, <evenMoreIds>}
group Id evenMoreIds = {1000, 15 to 30, <moreIds>}
```

We check these things in the first weeding phase (for the header, use file list, groups, and filters).

For filters, we make sure that we can only have the right number of types of fields (as defined in the config). The fields can occur in any order. We do not allow redeclaration of filters. For fields of the filters though, we allow multiple declarations. The idea was to allow in codegen the possibility of unioning the multiply-declared fields, as spoken about in class by Prof Hendren.

Team Work

Brendan worked on the config and annotating the variables, as well as type-checking/weeding for the structure of the filters and fields to check for conformance with the config. Shivan worked on the symbol table and type checking of the first weeding + type-checking phase. Yusaira worked on the symbol table and type-checking of the second weeding + type-checking phase (computations). Everyone worked on miscellaneous bugs. The hardest part of the entire project was making sure everything worked together. In particular, making the symbol tables and annotations work together was annoying.

Enumerations of Type-checks

FieldTypeCheck.onc - Even though a group may type check correctly (i.e. there is a particular field in the config file of the same type as it), if it is used with the wrong filter it will fail.

IntTypes.onc Ints can only be declared in groups that are defined as Int in the Config File

InvalidFilters.onc - If a filter is not defined in the config file, it will fail to type check.

InvalidList.onc - A value in a filter that is not defined in the list of allowed values for that filter's subfield will fail to typecheck.

InvalidParams.onc - The types of the parameters are not found to be fields in the config file

StringTypes.onc -Strings can be declared in groups that are defined as String in the Config File

WrongParamVar.onc - A parameter used in the group of a different type will fail to typecheck

TypeMismatch.onc - A field of type Int (as defined in the config file) fails when given a string value