COMP 520 Compiler Design
OncoTime Scopes and Type Specification

1 Overview:

The purpose of this document is to sketch out a plausible scope and type structure for the OncoTime language. You may deviate from this specification if you provide a good rationale for your approach.

2 Program Header

Your compiler needs to verify that the script name matches the prefix of the name of the containing file. That is file `foo.onc` should contain a script that begins with:

```
script foo // should be in file foo.onc
...
```

The script defines the top-level scope. Any parameters to the script should be inserted into this scope.

```
script foo(Sex g, Birthyear b1) // g and b1 are in the top-level scope
...
```

3 Groups

The first thing that needs to be done in the groups section is to inline the groups defined by the `use` clauses. This should be an inlining similar to the way `includes` work in C.

For example, if we had:

```
use foo.grp, goo.grp
```

then there should exist files `foo.grp` and `goo.grp` in the same directory as the program being compiled. It is a semantic error if a file does not exist.

The semantics of the `use` clause should be as if the `.grp` files were inlined into the main script file at the location of the `use` clause. You may wish to just use your scanner/parser on the `.grp` file, and insert the resulting AST in the correct location into the main AST. The `.grp` file can contain only `group` declarations and no other OncoTime constructs.

Assuming that all the `use` clauses have been inlined, then there are more checks to be done on the group declarations.

First, we should decide if entering the group declarations introduces a new scope. If we do start a new scope, then it would remain open until the end of the program, at which time both the main scope and the new scope would be exited. If we make a new scope, then a group declaration could declare something with the same name as a parameter.

```
script foo(Birthyear b)
/* comment */
group b = {1958, 1962}
```

If, instead, we put the group declarations into the main scope, then this example would result in a redefined variable `b`, which should be a warning or an error.

It is your choice, but justify which choice you made in your design decisions.

Each `group` symbol associate the group identifier with the set of values represented by the group. Thus, if a group declaration refers to another group, then the values should be inlined. Since groups are sets, repeated elements are combined into one. You may find it useful to store the group elements sorted, so that it is easier to operate on the groups later on.

You also need to ensure that the all the values on the rhs match the type declaration on the lhs.

For example:

```
group Birthyears b1 = {1958,1962}
group Birthyears b2 = {<b1>, 1954, 1962}
```

would result in two `group` symbols, `b1` would be associated with the list `[1958, 1962]` and `b2` would be associated with the list `[1958, 1962, 1954]`. Note that since groups are just sets, it is ok that `1962` was specified twice for `b2`. However, in the internal representation it only occurs once.

We also need to decide upon what rules should be allowed for the variables (`<id>`) that occur on the rhs of a group declaration. Certainly such variable could be a parameter or the name of another group. You need to decide if the other group must already be defined, or if it could be defined later.

For example, is the following allowed or not? A multi-pass approach to the symbol table would allow it.

```
group Birthyears b2 = {<b1>, 1954, 1962}
group Birthyears b1 = {1958,1962}
```

What about?

```
group Birthyears b1 = {1958,<b2>}
group Birthyears b2 = {<b1>, 1954, 1962}
```

You need to implement the semantic check for your decision, and justify your decision in the design notes.

Another design decision is to determine if a variable in the rhs of a group declaration could refer to a parameter. It would seem to make sense, as in:

```
script foo(Birthyear x) // x is a string passed in when foo is called
/* comment */
group Birthyears y = { 1954,<x> }
```

The implication of this decision is that if you allow a parameter, then you do not know the full group definitions until run-time, and you will have to ensure you generate code to deal with this case.

After doing type-checking/semantic-analysis of the `use` and `group` sections of the program, your symbol table should contain all the groups, with each group name associated with the list of elements of the group.

4 Filters

For the purposes of the course project we can assume that the filters allow only the items given in the OncoTime slides, although you may use a more modular/extensible design if you prefer. That is, you might want to define what is allowed in an external specification file, and then do the checking according to that file.

The purpose of the type-checking/semantic analysis is to:

- Verify the correct use of fields and values.
- Store the values for every field (even the implicit ones) in a format useful for subsequent code generation.

4.1 Fields

As outlined on the slides:

population filter should have five fields: id, gender, birthyear, diagnosis and postalcode.

period filter should have six fields: years, months, days, hours, start, end.

events is a list of events, where each event must be from the allowed events list. (detailed list to come, for now just assume an event could be one of the events listed on the page called *What are events?* on the OncoTime slides. Basically, you will need to be able to check that each event is in a pre-defined list of strings)

doctor filter should have two fields: id and oncologist (true if the doctor is an oncologist, false otherwise).

All fields should be represented in your internal representation, even if the user did not specify all the fields.

For example, if the user specified:

```
population is
  Birthyear: 1958
```

You would insert the missing fields, with the '*' value. That is, you would store the equivalent of:

```
population is
  id: *
  gender: *
  birthyear: 1958
  diagnosis: *
  postalcode: *
```

You should simplify your representation as much as possible. For example, you might want to resolve things like weekend to the actual days. For example,

```
period is
 days: weekend
```

would be represented as:

```
period is
 years: *
 months: *
 days: Sat, Sun
 hours: *
 start: *
 end: *
```

4.2 Groups in field declarations

Any place that a group name occurs in the filter declaration, this should be replaced by the list of group items in your internal representation.

For example:

```
group Events preplanning = {consult_referral_received, CT_sim_booked}
group Events planning = {CT_sim_completed, ready_for_treatment}

myEvents are Events: <prePreplanning>, <planning>, ready_for_physics_QA
```

In this case the list of values associated with `myEvents` should be the union of the items in `Preplanning`, `Planning` and "ready_for_physics_QA".

4.3 Parameters in filters

It probably makes sense that one should be able to use parameters in the filters. For example,

```
script foo(Sex gen)
/* comment */
population is
   gender: <gen>
```

would allow the user to call the script as `foo(female)` or `foo(male)`.

5 Computations

The filter part of an OncoTime program defines which patients, doctors, diagnoses and events will be operated on in the computation part of the script. Thus, we can imagine that the computation part creates three tables: patients, doctors, diagnoses. Each table will have an entry for each patient, doctor or diagnosis that makes it through the filter, and will have the field information as defined earlier.

5.1 Foreach

There are three related kinds of foreach constructions, `foreach patient`, `foreach diagnosis` and `foreach doctor`. These can be nested such that each kind of `foreach` is used at most once.

Thus, something like the following is allowed:

```
foreach Diagnosis d
  foreach Patient p
    foreach Doctor md
        ...
```

But, the following would not be allowed:

```
foreach Diagnosis d
  foreach Patient p
    foreach Doctor d
        ...
```

A question to determine is if it should be allowed to reuse the same identifier in an inner `foreach`.

For example, should the following be allowed?

```
foreach Diagnosis d
    foreach Doctor d
        ...
```

You can decide to allow it or not, but justify your decision. Whether it is allowed or not, each `foreach` introduces a new scope, and the identifier introduced in the head of the `foreach` is only visible for the body of the `foreach`. Thus you will have an enterscope on entering a `foreach` and an exitscope on exiting a `foreach`.

In each case, the `foreach` also declares a type for the identifier: Patient, Doctor, or Diagnosis. Each of those types have some predefined allowed fields which are:

patient ID, Gender, Birthyear, Diagnosis, Postalcode

doctor ID, Oncologist

diagnosis Name

It should be invalid to refer to a field which is not defined on that type. For example, the reference to the Gender field of a doctor is invalid.

```
foreach Doctor d
  print Gender of d
```

5.2 Tables

Tables are created by the table declaration, and in this simple version of OncoTime can only be created with the `count` statement.

For example, the following program creates a table x. This means that x should be inserted into the symbol table, and its type should be table.

```
table x = count patients by Diagnosis
```

Similar to the `foreach` we have patients, doctors and diagnoses as the actors, and the fields that can be counted are only those that were defined in the previous section.

For example, suppose we had a database that contained only three females, two of which had breast cancer and one had lung cancer. Then consider the following OncoTime program:

```
script Count()
/* comment */
population is
  gender: female
{ table x = count patients by Diagnosis
}
```

This would create a table with as follows:

0	breast	2
1	lung	1

Tables can be plotted, can have their length taken, and can have their items extracted. For example, we could continue our previous example:

```
script count()
/* comment */
population is
  gender: female
{ table x = count patients by Diagnosis
  print x.length //  can only apply length to tables (and lists?)
  foreach element i of x // i is inferred to be an index, x must be a table
    print x[i] // prints the pairs, [breast, 2] and [lung, 1]
               // x must have type table, and i must be an index
  barchart x // displays the table as a bar chart, x must be a table
}
```

A reasonable restriction would be that tables can only be created in the outermost scope (and not inside `foreach` constructs. We could allow redefinitions of tables, as long as both definitions are tables. For example, the following would be allowed.

```
script count
/* comment */
population is
  gender: female
{ table x = count patients by Diagnosis
  barchart x // displays the table as a bar chart, x must be a table
  table x = count patients by Birthyear
  barchart x
}
```

However, tables must be defined before they can be used, and one cannot use the same identifier to represent a table and a list (does this rule make sense?). Thus the following is not allowed:

```
script count
/* comment */
population is
  gender: female
{ barchart x // x not defined
  table x = count patients by Birthyear

  table y = count doctors by Oncologist
  list y = ... // cannot reuse y as a list, is this a reasonable restriction?
}
```

5.3 Sequences

In addition to the three tables (patients, doctors, diagnoses), the filter will also determine which events will be extracted from the database. We can imagine that we have a long series of events, each event has at least two fields: Time and patientID. However, some events will have additional fields.

Let us use some simple events for this description:

```
CT_sim_booked(time, patientID)
CT_sim_completed(time, patientID)
ready_for_physics_QA(time, patientID)
appointment_scheduled(time, patientID, doctorID)
```

When finding sequences the time field is used to decide the relative ordering, and the remaining fields can be used for matching. So, we could specify things like:

```
...
{
  list S = sequences like CT_sim_booked -> CT_sim_completed
  foreach member s in S
    print s
}
```

For these cases the sequences declaration creates a list of all matching sequences of events. The events listed must be from the valid list of events. If arguments are given for the match, the number of arguments must match the number of fields (not counting the time field, which is implicit). The first use of the arguments in the match also introduces a new identifier (that may be reused later in the the same pattern). The uses of these identifiers should be consistent.

Here is an example of an error:

```
{ // the following uses an undefined event, so error
  list S = sequences like CT_sim_booked -> bad_name -> CT_sim_completed

}
```

6 Deliverables for Milestone #2

1. For the types, you should print out:
   - all the groups, with the all the groups expanded and sorted.
   - the complete filters, with default values filled in.
   - the type for each implicit declaration (i.e. every place where a variable gets a type based on its use).
2. For the symbol table, you should print the symbol table on exit of each scope.