COMP 520 Project - Milestone #2

Victoria Feere
Jonathan Eidelman
Guilherme Raposo Pereira Mansur

March 2, 2015

Design Documentation

Group File Preprocessor

To import group files, we allow the parser to generate the AST as usual. We then go to the part of the tree which contains the use statements and create a list of all files which must be imported. We use our original parser to parse each of these files as an independent program which contains only a group section. We connect all these group sections together into one big list, and finally connect that list to the group list in the original AST.

Symbol Table Design

The symbol table is implemented similiarly to the symbol table in the JOOS compiler.That is as a cactus stack of symbol tables. The chief difference is that in this symbol table we use two global variables to keep track of the scope and symbol table we are in: currTable, and currScope. To update the variables with the correct properties, we keep another stack of symbol tables and everytime we enter a new scope a symbol table is pushed on to the stack and everytime we exit a scope a symbol table is popped form the stack that means the current scope and table is the one on top of the stack, this makes implementing the compiler flags easy since it's just a matter of printing the top of the symbol table stack. Also by having a currScope and currTable it's easy to check for declaration, redefinition, and undeclared indentifier errors. So the checking for these errors actually occurs in the symbol table the exception being the filter section wich is handled in the filterRefinements.c file.

Scoping in the symbol table is implemented like in the JOOS compiler through the scope symbol function. However we decided to allow nested for loops to declare the same variable with the same or different type. The newer declaration then masks the older declaration and any use of the variable is associated with the closest scope. We chose to allow this since it was easy to implement and it keeps the language flexible. See valid4.onc program for a

demonstration of scoping.

Script arguments are also handled in the symbol table and their type is inferred at the symbol table level. There is one case that is ambiguous however that is when a script argument is passed and the is only used in the computation section in a print statement. In this cases it's impossible to infer a unique type for the argument since a print statement can print an identifier of various types. Therefore the type is left as IDT (the default type of script arguments).

The symbol table also verifies that sequences are from the predefined lists, and stores their arguments in the symbol table. However the type-inferencig and checking for the sequences parameters is done by the typechecker.

Type Checker Design

Use Group Section

Since type inference and checking for undeclared variables is handled by the symbol table, the type checking phase for our compiler is relatively simple. First, we must ensure that all groups are declared as lists containing only strings. To do this, we have added a flag to check if a given ID_LIST is of uniform type, that is if each element in that list has the same type. If this flag is true, then the list has a type. Thus, for type checking, we must check if the list if of uniform type, and reject if it is not. Then we check if it is of type "string" and reject if it not.

Filter Section

There were very few cases that required type checking in the filter section since the grammar restricted specific types of integer and integer range lists be assigned to specific attributes. For the population attribute section we needed to ensure that any identifier type allowed in a list, such as for the diagnosis attribute, was in scope. This entails that it was defined as script parameter arguments. With respect to gender, it is during this type checking phase where we guarantee that the associated value is either M, F, male or female, case insensitively. In addition, postal codes are validated as a standard Canadian postal code. They must be 6 characters long and alternate between char and int, L7P3P3, for example.

As for the period filter section, a few of the attributes would benefit from being checked for their validity. We decided to check only the months and days fields. Days should comply with the days of the week as well as weekdays and weekends. Months should be an integer between 1 and 12. Additionally, the event filter, being a list of identifiers and strings, check if all identifiers are defined as groups. If they are not the compiler will through an "Invalid Group Error" and display the identifier's name that was not found to be in either an imported group file or as defined in the groups section.

Computation Section

Type checking the computation section is straightforward. We simply recursively call the type checker for each "if", and "foreach", while incrementing the scope pointer so that we give the correct symbol table to the recursive call. The only complicated cases are "barchart" and "print". Barcharts can only be made for tables. Print introduces it's own set of cases.

Print Statements

For print statements, we are allowed to have many printable objects one after another and we simply typecheck each of them to ensure that they are printable. We have made several decisions in this area. Firstly, if we have a statement of the form "print x of y", then we insist that x be an attribute that can be applied to a patient (as is predefined by the filter section) and that y be either a doctor or a patient. We do not yet have attribute for doctors, however we likely will later.

If we have a statement of the form "print x[i]", then we insist that x be either a table or a list. We also insist that i be the correct iteration object. There is a specific type for iteration objects for lists and another for tables. We insist that these match. We have decided that only tables have the property length and that only patients and doctors have timelines.

Sequences

Typechecking sequences creates several problems. Firstly, we must verify that all events are actually valid. We do this by checking with the events declared in events.h. However, then we must keep track of how many parameters each event must take, and what types those parameters must be. To achieve this, we include a constant array with the number of paramaters each event must take, and another array of arrays specifying codes defining which type each parameter must be. We then retrieve these to do type checking. During a type check, we first check that the number of parameters for a given event is correct. We then verify the types of these parameters. First, we check if a parameter has been assigned a type. If it hasn't, we assign it the type that it should have according to our event specification. If it already has a type, we check if that type is the same as our event specification says. If not, we reject the program.

Filter Refinements and Revisiting Pretty Print

In order to store the refined list of filter attributes and specified values for the code generation phase we felt it was a good idea to incorporate a modular phase which takes the AST and creates a new FILTER data type keeping track of only the most recent attribute declaration in the 3 respective filter lists. This was integrated with the pretty printer to print all attributes, including those with no specifications whose default is then represented by "*". When these attributes

become defaulted they are put in comments in the pretty printed file as "*" indicating "all possible values" is not a keyword we chose to include as part of our grammar specifications in Milestone 1. What is really lovely about the new version of the pretty printer is that it prints all groups, including the ones from imported group files just above the filter section.

The filter section complies with the specification to copy the strings from a group's definition directly to it's filter section. If filter field values are redefined, either within the same section or a following section, the most recent for that particular field is stored. If no filter section is indicated then it will not print out the resulting filter section, however it still stores an internal representation where every filter field is empty and therefore allows all values.

Additionally, we decided that only the "diagnosis" field would be able use an identifier as defined in the script arguments. All other attributes are either an integer list and therefore are required to be hard-coded or have predefined identifier types. For example, gender can be either M, F, male or female (case insensitive) and "days" also has predefined type values : monday, tuesday, wednesday, thursday, friday, saturday, sunday, weekend, weekday. Similarily, "start", "end" and "postalcode" needs to be of a specific format. This keeps validation and type checking for these cases up to the compiler and not the generated output program.

As specified, the pretty printer prints implicit type declarations in comment form to the right of simple statements. We do this for all types when they are first declared dependant on how they are declared. This includes things such as list s = ..., groups as in group heart = ... and elements in statements like foreach element i of a, when i is some element in the table a.

For a nice example of the new and improved pretty printer with types and filter refinements please refer to valid1.onc (path : programs/valid/valid1.onc).

Type Check - Invalid Programs

One of the most important parts of this process was to keep track of during which phase specific errors are generated and to test accordingly. During this milestone iteration, we decided to create distinct folders with tests during the type checking phase. This also included anything the group file preprocessor or symbol table catches early on, such as identical identifiers with conflicting types and previously undefined types caught when performing implicit type casting.

The following is an enumerated list of our invalid test scripts corresponding to our type checks by folder name in "programs/invalid/types":

groups_section

1. non-existant-grp-file.onc : A group file that does not exist but is indicated in the program spits an error saying this external file does not exist.
2. invalid-explicit-group-redeclaration.onc : A group name cannot be used twice in a new group declaration.
3. invalid-group-redeclaration.onc : A script argument cannot be implicitly cast as a group.
4. invalid-group-sequence-element.onc : A group cannot have an identifier in it's sequence that is a group type.
5. invalid-undeclared-group.onc : Cannot use an identifier that is previously undeclared in a group value sequence assignment.

filter_section

1. invalid-postalcode-attribute.onc : A postal code which is not a postal code type, it does not adhere to the proper Canadian postal code format.
2. invalid-diagnosis-attribute.onc : A diagnosis attribute which is not defined as a group in either an imported group file or in the group section of the script or as an input parameter
3. invalid-gender-attribute.onc : An identifier in the gender field list that is not one of "male", "female", "m" or "f" (case insensitive).
4. invalid-event-filter-group.onc : An identifier in the event filter list that is not a group name in either the defined groups in the group section or from an imported group file.
5. invalid-day-attribute.onc : A day identifier in the day field that does not correspond to a valid day as defined by the prespecified day types "monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday", "weekday" & "weekend".

compuation_section

1. invalid-explicit-redeclaration-table.onc : A table is declared using the same identifier as a parameter input identifier
2. invalid-sequence-non-event.onc : A sequence list item which is not one of the pre-defined events.
3. invalid-sequence-conflicting-type.onc : We delcare a sequence where a parameter is first inferred to have type Patient, but is then used in an event which expects a paramater of type Doctor. This gives a type error.

4. invalid-sequence-incorrect-arg-num.onc : Here an event takes the incorrect number of arguments, and the type checker rejects it.

5. invalid-index-wrong-type.onc : This program attempts to access a[i] where a is a list, but i is the index of a table. This gives a type error.

6. invalid-barchart-of-list.onc : This program tries to create a barchart of a list. We only allow barcharts of tables

7. invalid-print.onc : A print statement that attempts to print something not a previously declared identifier.

NOTE: The file Contributions.txt keeps track of who-did-what sectioned by each milestone.