COMP 520 - OncoTime Compiler Project Report

Victoria Feere
Jonathan Eidelman
Guilherme Raposo Pereira Mansur

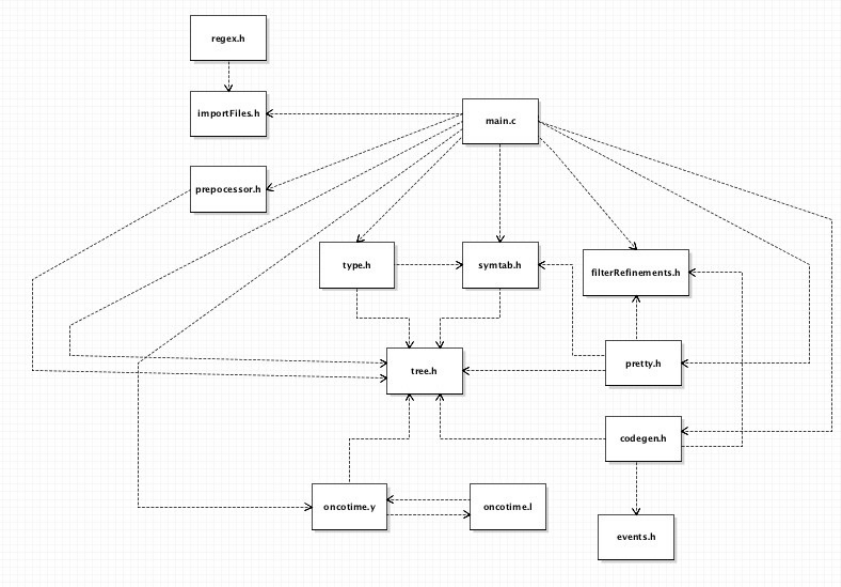
April 10, 2015

Contents

Introduction

For our OncoTime compiler's scanner and parser we decided to use flex/bison primarily because we were all familiar with it, it's efficiency and the amount of documentation out there supporting it. Additionally, the compiler's C code would be easy to debug and is lower-level with many native libraries available to it so it is fast and flexible. As the input program is parsed we build the AST for the program, create a symbol table and perform a type check. In addition to these standard compilation phases we have added a group file preprocessor to handle the import of group files and a filter refinement phase where a pass is made over the initial filters and only the most recent filter is taken as the final value(s). Our compiler generates python code, further information as to our reasoning is discussed in the section on code generation. An overview of the structue of the compiler is presented in the simplified class diagram below.

Figure 1: Class Diagram of the Oncotime Compiler



Scanner & Parser

Group File Preprocessor

There were a few overarching questions we had to ask ourselves before plunging into the OncoTime language specifications. For one, how were we to deal with importing group files? To parse or not to parse? Ultimately, we decided to design a preprocessor to handle this task before code generation.

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.

If Statements

One of the primary subjects of our discussions was whether we wanted to add the notion of an "if statement". It was decided that we were going to add this and that the syntax would adhere to the follow grammatical rule: "if expression then computations else computations endif". The imperative "endif" helps to mark the end of the scope of an if statement.

For Loops

As it stood, the addition of if-statements came at quite a low cost seeing as how well defined and simple their accepting DFA. This, however, was not the case with for loops. In class, there were examples both appearing with and without brackets. Those without opening and closing brackets had one-line below them often performing some simple computation. For this case, we decided it would be best to restrict this case to handle merely print and barchart statements. This, acting as a shorthand for those quick computations when you simply want to iterate through patients, doctors or diagnosis, for example.

Filterable Attributes

One of the most difficult decisions we had to make was regarding what we should allow as attributes in the filter section of an OncoTime program. On one hand it would be possible to allow for the user to define any form of identifier as an attribute and on the other we might want to enforce strict program typing rules in the form of a set of attribute keywords. For the former the trade-off is that there will exist greater flexibility for the programmer but more complexity must be introduced to the compiler's weeding phase. In the case of the latter we simply see a more strict programming language.

The good news about this is that OncoTime is a language developed not primarily for coders but for those who might want to perform quick, concise and practical computations on a well-defined data set. For this reason making a strict language may in fact be of greater benefit to our target end user. Additionally, since it is easy to extend the languages set of allowed attribute keywords it won't come at a very high cost to the compiler writer update in the rare case the database's schema changes in a way requiring language updates. If pre-existing programs written before the new language version were to fail at compile-time given the newest language version it would be improving upon the

fact that with the older language, an error would occur during run-time, errors much more difficult to catch and handle. For these aforementioned reasons our group decided to include a predefined set of keywords for filterable attributes. Although, in future studies, it might be interested to explore the former option.

Script Parameters

Script parameters are defined as identifiers and therefore will not throw an error when found in the following sections:

- filterable attribute assignment as the value being assigned to anything able to be assigned to an identifier (excludes int types such as age birthyear)
- compared against an attribute in an if-statement condition not of type int
- declared as a group in the groups section

Otherwise, the parser will throw an error as it will have not found the matching syntactic structure of an identifier following/proceeding a particular token. This is something that could be changed in future revisions, however it does avoid having to compare parameter inputs of type int with occurrences of it in the script which makes for simpler type checking. Unfortunately, this imposes a slight restriction on the set of potential programs.

Other Syntactic Decisions

A few minor decisions with respect to what is valid syntax were made and are listed as follows:

- 0 cannot proceed any token of type integer - for example, 01 - invalid, 1 - valid
- an "hour" token is introduced and is in 24-hour clock format, accepting both 1 and 2 place LHS hour indicators (ie. 0:23, 00:23, 23:00 - all valid)
- filterable attributes are all keywords and must appear in lowercase
- filterable attributes cannot define "*" (anything allowed) as their RHS as this is the implicit definition when no filter is defined
- we allow sequences of single events but not "-" before an event (might have been a typo in the specs presented in class)
- accessing an element in a table with a particular key looks like "x[i]" and you are able to access a table's length using the keyword "length" as in "x.length"
- "timeline" and "barchart" are special keywords and will correspond to a very specific generated code macro

- strings are indicated in quotation marks
- identifiers are all sequences of characters not corresponding to a specific keyword

Parsing & Creating the AST

Any syntax error undetectable by the scanner was picked up by the parser. This included optional sections and subsections, every possible type and every statement's syntactic structure. Of course, for loops, sequence, print, barchart, if and table statements have special structures as defined in the grammar. However, there are a few special cases where the parsing phase will detect a type error as opposed to during the type checking phase. These special cases include group definitions, where groups must be a string or identifier, declaring filterable attribute values which are associated with a particular type and checking that if conditions also adhere to these attribute types. An example of where the parser would fail is on the statement "if gender = 1 then print x endif".

Pretty-Printing

We based the design of our pretty printer on the one given in the JOOS example. Essentially for each tree structure that we had, we created a printer which performs a case analysis on all possible types within that structure. This allows us to easily print the appropriate construction. We tried to keep pretty printers as specific and as well-encapsulated as possible. for example we were able to easily separate the printing of lists into their own functions. One issue that we ran into was that indentation was difficult to maintain for highly embedded computation sections, and therefore we enforced a maximum of 3 tabs in our pretty printer. Because of recursion, if we wanted to keep track of indentation for all cases, we would need a global variable tracking our current level of indentation. We hope add this in a future version. One important choice we made is that we do not store the content of the user's Documentation Comment. The reason for this is it has no syntactical meaning, it is just to ensure that the user writes clear code. Thus, our pretty printer simply prints a comment with the word "COMMENT" as its document comment, so that its output will be a syntactically valid OncoTime program.

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 Checking

Use & Groups 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 as-

signed 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

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 the Milestone 2 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.

Code Generation

Target Language

Given the scope of our DSL and the types of programs useful for radiation oncology specialists it was clear that the incorporation of data visualizations would both encourage generating a use-able interface and create extremely interesting user interactions. For this purpose one language jumped into mind, python. Python not only has a large number of libraries for straight-forward data visualization and GUI development but it’s relatively portable, lightweight and flexible.

Integrating with SQL

In order to connect to the database in SQL we were required to import the sqlalchemy python package as well as mysqlconnector in the python output script so as to connect to the remote database. Additionally, if you are not locally connected you are required to open port 3306 on your local machine as

an ssh tunnel to the DB server. Appropriate import statements and connection variables are printed to the python file at the very beginning of the code generation phase to enable communication with oncodb on hig.cs.mcgill.ca.

In order to minimize the amount of code generated at runtime we predefined SQL query prefixes written in SQL syntax to later append to them any relevant filter information. Any attribute corresponding to a particular filter field the user is able to specify is projected in the SELECT clause. Event, doctor, diagnosis and patient queries are required to implement the languages specified "foreach" loops.

At code generation time the WHERE clause of these queries is where appropriate filter information can be translated into applicable SQL constraints. To do this, at the beginning of code generation a method was implemented for translating the final filter fields to their appropriate SQL comparator substring. For example, "birthYear: 1992 to 2010" would result in the SQL WHERE clause: "WHERE BirthYear >= 1992 AND BirthYear <= 2010". These are applied to the patient, diagnosis and event queries to be used when needed given a corresponding computation. Flags are implemented to manage whether or not all patients, diagnosis and doctor queries have been used previously in the computation code to retrieve and store relevant tuple information. If not, the variable assignment corresponding to the sql query execution is generated in python and the flag is set to 1 (True). As for events, which take parameters, the constraints imposed by which patient or doctor is passed to it are appended to the query upon generating python code for sequence declarations or loop iterations over sequences.

Sequences & Timelines

One of the more challenging aspects of code generation in Oncotime is matching on sequences. This feature allows the user to ask common questions which would be difficult to formulate in SQL such as 'Were there any times when Patient P got a CT scan, then either made an appointment or completed their treatment plan?'. This is trivial to ask in Oncotime using sequences, however it is more involved in Python.

To achieve this, we recurse through a sequence tree twice. The first time, we find all of the events being asked for and the patient arguments given for those events, and generate the SQL queries which retrieve this information. For example, if an element of a sequence is 'CT_sim_booked(p)', then we will generate an SQL query which finds all of the times patient P had the event CT_sim_booked. All of these data are stored in temporary variables labelled "S_n", where "n" is simply a count of the temporary variables created so far. The only sequence feature we allow aside from the arrow, is the logical disjunction. We do this by simply adding the contents of each temporary variable created by the conjunction into one new temporary variable.

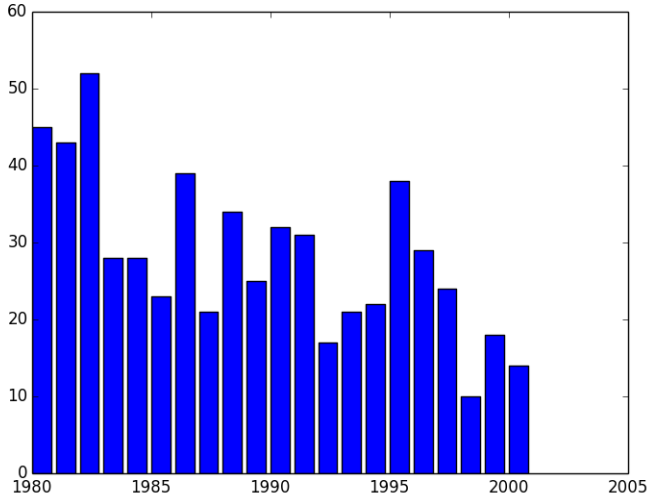
In our second recursive descent through the sequence tree, we generate Python for loops for each temporary variable that we created in our previous descent. Once we've generated these nested for loops, we check if all the events we're examining happened in the order the user has asked for, and if so we add them to a Python list.

Barcharts & Tables

Tables in Oncotime map the number of patients that have a given attribute (age, sex, diagnosis) to that attribute. The patients are selected from the subset of patients specified by the filters. So the generated code for tables is aSQL COUNT statement with a GROUP BY clause on the attribute, refined by the a WHERE clause on the filter fields. Since Tables are declarations they are stored in a varibale which may conflict with the generated python namespce. The main purpose for tables is for data visualization through a barchart.

The barchart statement does this by generating a call to python's matplotlib library's plot function. A barchart generated from a table is show in figure 1 below.

Figure 2: Example "barchart" of subset of patients between ages 15 - 35



Future Work & Improvements

Upon reflecting on our final submission, we feel it would be greatly beneficial to incorporate a doctor filter so as to refine the SQL results when iterating through

doctors. This could include “number of patients”, “start date”, “availability” and “speciality”. This would give users greater flexibility and allow use of the language for more specific scheduling purposes as this seemed to be a relevant point of turbulence at the hospital. Time and resources can be sparse so automated scheduling systems based on employees availability and specialities would be greatly beneficial to administration personnel.

There are many interesting features which could be added. We did not have time to implement the logical conjunction into sequences. It would be interesting to be able to ask questions such as ”Did patient P make an appointment and see the doctor, both in between making their CT scan appointment and completing their CT scan?”. This is not a huge extension from what we did, but would be extremely useful.

We also did not provide any clear separation between the Python namespace and the Oncotime namespace. This could potentially be extremely dangerous. Users could define Oncotime variables which have meaning in Python, or have the same names as temporary variables which we are introducing. There are several possible solutions for this problem. The simplest of these is simply to reassign arbitrary names to user defined variables in the internal parse tree of Oncotime. This would be a simple solution to the problem of conflicting namespaces.

Another interesting feature that could be expanded is the visualization component. Right now only barcharts are generated with no styling directives. It would be useful to extend barchart statements to other types of graph, to allow visualization of data from questions like: ”What’s the proportion of patients between the ages of 18 - 40 with breast cancer?”. This would be a simple modification to the grammar, and the table’s SQL statements. The style directives could be implemented as another filter section i.e a style filter.

Conclusion

Our project demonstrates both some of the real advantages of designing and implementing a compiler for a domain specific language, as well as some of the significant challenges. Our section on code generation shows some of the success of our work. Much of the tedious SQL code which would be required to ask fairly simple questions about our medical database can now be generated easily and automatically by our compiler. Users have sufficient flexibility to ask many useful quetions the data, without the added burden of writing complicated SQL queries.

Furthermore, we have successfully abstracted away some of the complexity of reasoning about time sequences. In traditional query and programming languages, questions about sequences of timesteps and events are difficult to for-

mulate. Oncotime has successfully abstracted this into a simple pattern match over events, expressed intuitively with a series of events and arrows. The target language code required to reason about time is generated automatically, and is no longer the responsibility of the user. Our framework allows users to ask questions in a simple, elegant way, without becoming bogged down in complicated technical details. The result is that scientists can focus on the actual science that they are interested in, and abstract away the practical problems of implementation.