COMP 520 Project - Milestone #1

Victoria Feere
Jonathan Eidelman
Guilherme Raposo Pereira Mansur

February 16, 2015

1 Design Decisions

1.1 Handling Group Files

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, the decision our group made was to push the verification of a particular group being defined in the group file to the type-checking phase. At which time we will decide how to handle undefined or syntactically invalid groups and notify the user as to what inference will be made. This was decided as for this milestone we felt there were already many other more pressing decisions and extensions to the language that had more impact on what was possible for development in future milestones.

1.2 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.

1.3 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.

1.4 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.

1.5 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.

1.6 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

1.7 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”.

1.8 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.

1.9 Compiler & Generated Code Language

For the compiler code 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.

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. It should be interesting to see the outcome of these decisions when continuing implementation for Milestone 2.