Final Report

Team Dragon Army - Group 18

OncoTime Compiler

Yusaira Khan : 26052607

Shivan Kaul : 260512593

Brendan Gordon : 260529254

Introduction

Technicians and oncologists at the Centre Universitaire de Santé McGill (CUSM) found themselves with a major problem. A majority of the analysis, verification and planning of patient treatments done at the hospital's oncology department were being laboriously coded by hand. This meant specialists expended valuable time and effort on the minutia of making database queries, instead of more impactful work like planning treatments. OncoTime is a domain specific language (DSL) currently being developed by Professor Laurie Hendren which is intended to fix that problem. The goal of the language is to allow doctors and medical specialists to be able to specify in natural language the kind of patient data they want to see and analyse, and let the compiler figure out the database query handling and pretty printing. This is where Team Dragon Army comes in.

Dragon Army (aka Group 18) is a group of 3 students in Professor Laurie Hendren's COMP 520: Compiler Design course at McGill, composed of Yusaira Khan, Shivan Kaul and Brendan Gordon. The three of them developed the **D**ragonArmy **O**ncotime **C**ompiler (**DOC** for short), a compiler for a variant of OncoTime emphasizing clarity, explicitness and portability. This last feature is important - given the many different

kinds of databases that the OncoTime language could be used on, it's crucial to allow some degree of schema agnosticism. Team Dragon Army set a personal goal to design the compiler in such a way that would allow it to be adaptable to any oncology department's needs.

The rest of this report will outline the structure of our compiler, the design decisions we made with regard to both the language and the compiler, our contributions, and features we would like to implement, given more time. Note that the material presented here borrows heavily from reports for the milestones over the semester.

Compiler Structure

DOC is written entirely in Haskell, making heavy use of the Parsec Parser Combinator Library. On the surface, our choice of language and toolkit may seem unorthodox given that class materials emphasized the use of an imperative language in conjunction with a lexical analyzer and parser generator. However, our choice of language and library made sense for a variety of reasons.

Haskell

- We wanted to do our project in a functional programming language. We had heard from students who had taken the course previously that this would be an intensive project, and that a lot of the features of functional languages (i.e. referential transparency, ease of reasoning) would aid in both the design and development of our compiler and reasoning about it. Also, a functional language has superlative pattern matching support and efficient recursion - features that are helpful while writing a compiler.
- Haskell has a helpful community which is always looking to help newcomers.
- A compiler for OncoTime in Haskell hasn't been written before.

Parsec

- We are using Parsec [1] - a monadic parser combinator library for Haskell. We made this choice because we had worked with Parsec and other options out

there (Alex, Happy) while working on MiniLang, and Parsec seemed to offer a lot more control and better documentation than other tools out there.

- The documentation for Alex and Happy were considerably lacking, and most examples were written for prior versions. Parsec, on the other hand, is a tool with wide adoption both in and outside the field of compiler design. It also offers a lot of flexibility, which we figured would be convenient when it comes to the design and implementation of a new language such as OncoTime.

Phases

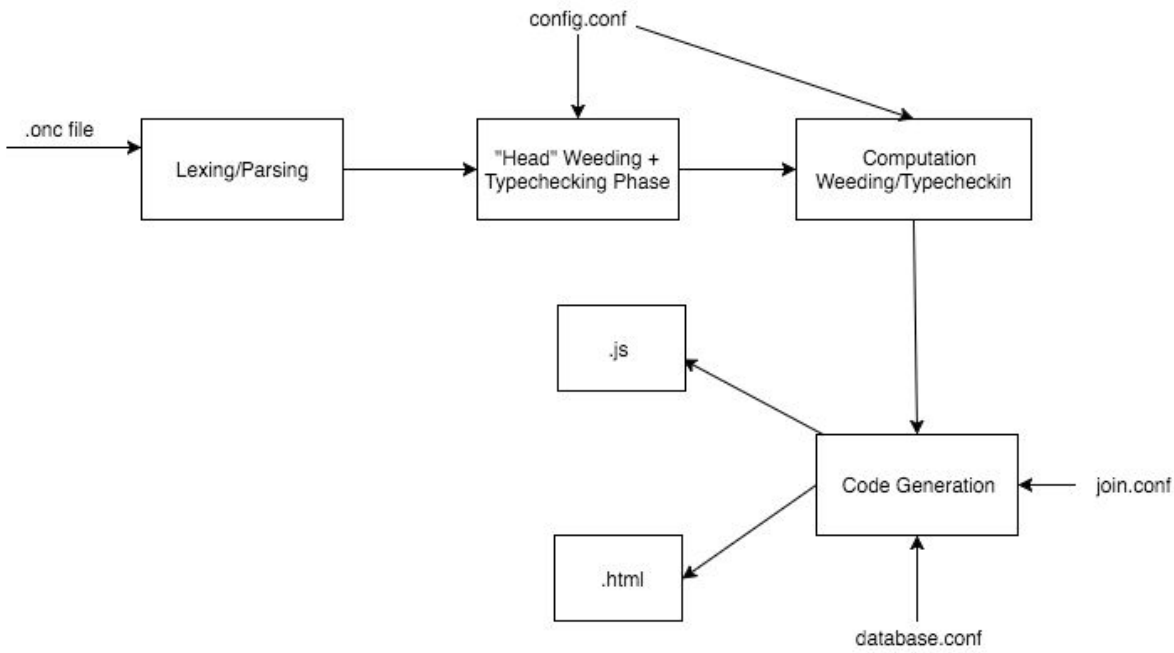

Figure 1: Phases for DOC

As we can see in Figure 1, our compiler follows the usual compiling lifecycle, but with a few interesting additions with respect to the weeding/typechecking phase and codegen phase. Plus, we use multiple config files to keep things as general as possible for all stages of the compilation process.

Lexing/Parsing

Parsec is a Parser Combinator which works best for LL grammars. Parsec lets us defined reserved keywords and operators (which we do in Lexer.hs). Then, in our Parser.hs, we construct and compose Parser Combinators which consume the input based on our 'grammar'. The 'grammar' is defined simply by specifying what kind of input our Parser Combinators consume and types they output. The types are strong - we define *all* of our constructs in Types.hs. This was one of the strengths of our compiler - we spent a significant amount of time and energy thinking about the language early on and defining our Types well, which meant that later on things fit properly, and we didn't have to change our Parsing/Lexing phase too much. As an example, here's some cleaned up Haskell code showing how the top-level Parser works, for the entire program. Note that in our actual code we have some other ugly practicalities such as saving line numbers, annotations, etc. Haskell comments are denoted by two dashes (--).

```
oncoParser :: Parser -- showing the top level oncoParser, with type Parser

oncoParser = -- function definition

    do

        whiteSpace -- eats up whitespace

        hdr <- headerParser

        doc <- documentationParser

        use <- many useListParser -- the `many` keyword consumes, well, many useLists

        grp <- many groupsParser

        filt <- many filtersParser

        comp <- many compsParser

        return $ (Program hdr doc use grp filt comp)
```

We can see how cleanly this maps to the structure of an OncoTime program. The headerParser consumes the head of our input program and emits a Head type;

similarly for the other constructs. At the end, our oncoParser emits the top-level Program type which also serves as our AST.

Config.conf

(For the exact syntax, see our Milestone 2 report submission)

Our config.conf file allows users to specify which filters (e.g. doctor, population), and which fields (e.g. id, oncologist) for those filters are permitted. In addition, it allows them to specify whether it has an integer type, which allows ranging, a string type to allow arbitrary values, or a list of permitted values. We also allow users to specify which types can be looped over. In effect, this config file is a portable grammar of sorts, which allows users to define how they wish to filter their data, and how it can be operated on. Our config.conf file is central to our philosophy of making our compiler portable.

Weeding + Typechecking

Instead of doing weeding followed by typechecking, we split up the phases into 2 by doing weeding + typechecking first for header, use files, groups and filters, and then doing the same for computations. Given how intensive the weeding is for OncoTime, doing it this way made sense. Some interesting highlights -

1. We had two symbol tables, one for each phase. While building up the type information and storing it in the symbol table, we could make use of this in the weeding phase. We went into exhaustive detail on how we do this in our milestone 2 report. The only new thing added since then is that we now weed for foreach sequences that don't make sense (Diagnosis -> Patient -> Doctor).
2. We require all parameters to be typed. Explicit > implicit
3. We added in Annotations (which was non-trivial in a pure functional language) after realizing that we would need this information for the code generation phase since we discard our symbol table after typechecking. This

helps us generate appropriate methods for the ‘Print’ statement and to see which ‘actor’ a particular program refers .

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>}
```

Code Generation

DOC compiles .onc files (with supporting .grp files) down to JavaScript (running on Node.js). The reason for compiling down to JavaScript was the excellent support for data visualization and package management. Barcharts are outputted in pretty graphs in HTML using Plot.ly [2]. The outputted .js files are capable of executing the SQL queries required to fulfill the needs of the information requested in the .onc files.

Highlights:

1. To handle events, we define what the SQL query should look like for each of the 5 events we’re supporting : patient_arrived, ct_sim_booked, ct_sim_completed, treatment_completed and end_of_treatment_note_finished (Task). Note that we do not require the user to type in explicitly the word ‘end’ to end a sequence - the closing ‘]’ should be enough indication that the sequence has now ended.
2. If we see a sequence, we gather the events defined in it, generate the SQL query for all the events, nest a query to deal with Patient filter inside each of them (we could in the future just run the Patient filter query once and then memoize it for a performance boost), concat them, use the filters to generate the WHERE clauses, and execute the multi-SQL query. With the resulting list of lists of results objects, we sort them by Patient IDs, make sure we only have Patients which actually have all events which we’re interested in, and then pretty print them. Note that we did not have time to implement wildcard

matching for sequences (event1 -> {event2, event3} -> event 4) or alternation. Our sequences do not mandate that the events happen in order, just that the events happen for the patient. For any patient, the first time that the event occurred is printed.

3. For table printing, we take an initial pass over the computations to build up a 'print table'. This table contains all the variables we need to print with all the information needed to print them. For example: table x = count patients by Diagnosis would contain the information (x, patients, diagnosis). Now, we simply fold over the print table, generate the SQL query to get the data (using the filters), and print it. We also handle barcharts and lists in this way. Note that we do not support printing tables via iterating over all rows and doing `print x[i]`. We only support printing the entire table. This is because for now we mandate that we can only have print statements inside the foreach element i of table... block. We also do not support timelines.
4. For barcharts, we use Plot.ly and emit JS + HTML code to visualize the data.
5. We did not have time to support taking arguments from the command line (variables declared in the header). Using these in the program may cause undefined behaviour. We also do not print timelines for patients.

Database.conf + join.conf

Our database.conf and join.conf files are similar to our config.conf file, in that it allows users to configure DOC to suit their needs. In this instance, however, they operate entirely behind the scenes.

The database.conf defines a mapping between the filters and fields, and what table and column names they correspond to. For example, "Patients", "Patient", and "Population" all map to the patient table "Patient", and a value like "doctor_loop" specified in the database.conf would correspond to the "DoctorSerNum" column in the database.

Also related to database configuration, is the join.conf file. This file specifies which tables can be joined, and which column they can be joined on. In our particular

instance, we specify that Doctor, Patient, and Diagnosis can be joined on PatientSerNum.

Conclusion

When we originally considered how to approach the problem of designing a compiler for a domain specific language like OncoTime, we hit upon a problem: What if the domain for which the compiler was designed for changed? Not only that, but how would a compiler designed for one specific domain, adapt to other similar ones? After considering this problem for some time, we realized that it made little sense to hard code in a grammar that was specific to the single domain of the Oncology department at the CUSM. It is our opinion that OncoTime provides a basic structure for filtering data, and operating on it - how and what data is retrieved should be domain and schema agnostic. In this way, we saw that OncoTime could be adapted to suit the needs of hospitals outside of Montreal, or domains completely unrelated to Oncology altogether.

We tried our best to make OncoTime as modular and portable as possible. While we didn't fully succeed in that regard, we are still very proud of our work. We fulfilled most of the requirements assigned to us, but also added a wealth of features that we think will make OncoTime a richer language, with a wider range of application.

Future Work

We quite enjoyed working on OncoTime, and at times got quite enthusiastic discussing all the features we wanted to add.

Chief amongst these ideas are the following, which we feel emphasize OncoTime's guiding principle of making analysis and filtering of data easy, and our philosophy of adaptability.

- Jupyter style notebooks. Essentially, we want to be able to generate fully dynamic HTML pages. Then, enterprising users could easily generate script to

take files, turn to html pages, and host on a server. This would allow analysts easier access to data.

- We would like a more elegant and nuanced config system. There are quite a few problems related to our current system, specifically with regard to how it handles events (it doesn't), or how its interface with a database is relatively limited. A better system for specifying joins would also be ideal.
- Some sort of automatic configuration that would generate config files from a given database schema. Right now, we impose an upfront cost on users for configuring the compiler, when it wouldn't be too much harder to generate it automatically.

Who did What

Yusaira

Handled the weeding + typechecking for computations, preprocessing for lexer, major fixes for Parsec, and events SQL generation and sequences JavaScript. The team's Maverick.

Shivan:

Handled the build and test framework, Pretty Printer, Header Weeding + Typechecking, setup for JavaScript code generation + database connections, and minor code generation for foreach loops and sequences and integration. Provided witty quips on occasion.

Brendan:

Worked on the initial setup of the parsers, header weeding, almost everything to do with the creation and integration of configuration files, typechecking of filters and fields, implementing annotations, generating all non-Sequence computations, and keeping everyone's spirits up.

Resources Used

Libraries:

[1] Parsec: https://hackage.haskell.org/package/parsec

[2] Plot.ly: https://plot.ly/

[3] Data.HashMap.Strict: http://hackage.haskell.org/package/unordered-containers-0.2.2.1/docs/Data-HashMap-Strict.html