Rationale for language and tools

We are writing our OncoTime compiler in Haskell. The reasons for this decision are:

1. Based on discussions with students who have taken the course previously, and the Internet, a functional language is a good choice for writing a compiler in. This is because of (among other things) pattern matching support and efficient recursion.
2. We all wanted to learn Haskell, and writing a compiler in it for a DSL under Professor Hendren seemed like a fantastic opportunity
3. Haskell has a very dedicated community, which is always looking to help guide newcomers
4. A compiler for OncoTime in Haskell hasn't been written before.

We are using Parsec - 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. Additionally, we knew that OncoTime would not require us to deal with many computations involving things such as precedence. As a result which meant that a lot of the features of Alex and Happy would go unused.

Design Decisions

We have made the following design decisions for our compiler:

1. An overarching design principle behind our implementation is "explicit is better than implicit". This was done because the DSL is for people who are not programmers. Excessively magical syntax would just be confusing for someone not used to writing code.
2. The structure of the program is concrete - it follows the one presented by Prof Hendren in the OncoTime slides (slide 24). We cannot have inter-mixing between sections. For example, it's mandatory to have use statements before group statements.
3. We allow for the strings to not be quoted. So the following is valid:
   ```
   group Id y = {1, 2, <x>, breast, 1200 to 1231, after 1200, before 5}
   ```
4. As can be seen in the above example, we also implemented logic for ranges and "before" and "after" constructs.
5. We decided to mandate that foreaches need to have braces that start on next line. Again, this was done to make things explicit. So we have the following syntax:

   ```
   foreach Patient p
          {
   ```

```
            ...
        }
```

6. Before parsing, we check if the script name is the same as the filename. The script names can only be alphanumeric.
7. We did not bake in the types of identifiers, and have chosen to define a configuration file where users can define which identifiers they wish to use, as well as which types they want to associate with those types. The logic to create a data-structure from that configuration file was included.
8. We began to implement weeding (to decide which identifiers were allowed or not allowed) in our code, and the basic data-structures and logic to do so currently exists, but eventually we had decided to defer the complete implementation and integration of it until the next phase. One reason was that the same configuration file that determines which identifiers are permitted, also stores which types that are valid for the fields of those identifiers. From our perspective, it made much more sense to worry about dealing with this during the type-checking phase, especially since fields may contain variables whose type is unknown during parsing. We determined that this would mean an extra pass, which we thought to be unnecessary.
9. The following sections are mandatory: header, docs, and computation list. For computation, it is mandatory to have the curly braces signifying the computation block. The programmer can choose to keep this empty. If this block is empty, we display a warning.
10. Following the slides, for printing table x's length, we follow the syntax "x.length".
11. For comments, we do not require that the programmer start every line with an asterisk. This is unlike how it is done in the reference compiler. Our rationale behind this is that unlike languages that have similar syntax for block comments (Java, etc), OncoTime does not (yet) have an IDE. The programmer has to manually enter an asterisk on every single line, which we thought was completely unnecessary and rather painful. Our documentation comment is delimited by the opening '/**' and the closing '*/', i.e. the following format is followed:
    ```
    /**
    This is a documentation comment.
    */
    ```
12. We require that script arguments have types, to keep things explicit and simple.
    ```
    script test(Id a, Date b)
    ```

Team Organization

The work division happened organically. Almost all code written was written at the same time, in the same room. We discussed and wrote the grammar informally, and Shivan wrote up a more formal spec. Brendan had the most experience with Parsec as a result of his work on his assignments, so he created a large portion of the parsing logic, before beginning work on the weeding. Shivan contributed to the types, wrote the pretty printer, did fixes on the parser that arose while writing the printer, and set up the build/run scripts and testing

harness. Yusaira worked on the types, wrote a significant portion of the parsing logic, and set up regexes to handle Parsec's lexer's shortcomings. Everyone fixed bugs as they arose.