CSE 503: Assignment 1

Instructed by *Michael Ernst*

Ji Luo (Student No. 1940053; UW NetID `luoji`)

3 April 2019

It took me about 70 minutes to do this assignment.

Utility Reusing

UNIX is famous for the (perhaps miscredited) philosophy of creating utilities that achieve simple tasks and combining them through piping. Different utilities are chained together to achieve a goal. These utilities communicate via a binary stream, the pipe. Though being binary, most programs consume and produce text. While this methodology has proven effective, it is also prone to errors and reusability problems.

1) Suppose we are working on a table. Since programs communicate in the textual representation, each program must first parse the input, process the data, and finally serialize them into the output. This parse-serialize happens at eery boundary of utilities, which is unnecessary and deteriorates the performance. Even worse, parsing and serilaizing are error-prone. Specifically, serialization usually involves escaping, which, when done incorrectly, could confuse the downstream programs and even cause severe problems (e.g., unintendedly performing a devastating operation). This is the error-prone part.

2) Suppose a DevOp needs to process some data and perform some tasks based on them. In a chain of processing, one or more usually are done as "dirty work", i.e., raw string manipulation. For each specific data processing task, it could be the quickest method to come up with. The resultant is often a shell script one-liner that sophisticatedly combines the utilities at hand but quite limited to the task and even the data of this batch, because the string manipulation part implicitly assumes the input has some simple form. This problem is especially often if the text manipulation is done by regular expressions and the like. A lot of data are only expressible as context-free languages, yet the program works because the input space bears additional assumption. This is the reusability part.

To avoid injection introduced due to SQL statement concatenation, the solution is to use a dedicated program that are dedicated for escaping and statement synthesis, or even better, do not transmit SQL statement in its textual representation to the database engine, rather just hand an abstract syntax tree to it. The idea is similar: keep the data in its native, binary form inside the whole procedure, and only do conversion between binary and textual forms when the data need to cross a computer-human boundary. This is essentially using strongly-typed programming languages. However, most programming languages are not designed for DevOps or day-to-day tasks, and it is much more cumbersome to code in a separate editor, debug and run a "program" than to write a few lines of shell scripts.

A middle-ground is taken by typed, object-oriented shells, e.g., PowerShell and elvish. Putting the data into their native representation allows faster and easier manipulation as well as allowing us to produce different output string (e.g., different columns in a table, different statistics of data) from the data without another parse-serilaise round-trip. But they are not popular, which already indicates they're not well-received. I would say the problem is it's interoperability with traditional process-based utilities, and the difficulty to write reusable scripts (it's as easy to write cryptic PowerShell scripts only useful for specific

cases as it can be for Bash). I believe the solution is to create a interactively usable scripting language at a sweetspot between enforcing reusability and allowing quick hands-on usage patterns.

Monstrous C++ template compiler errors

The title explains itself. For one thing, a lot of template libraries use templates to several levels. When an error manifests itself, the instantiated type name is very long, making it very hard to understand what is going on. Because C++ templates are only checked when instantiated, and several error-elimination rules automatically kick in (SFINAE), the reported error can be seemingly unrelated because it could be the case that the intended instantiation is tossed away, and another instantiation is initially chosen at the first instantiation point, yet an error might be triggered only at a distant point of source code. This situation is much improved thanks to `static_assert` and `enable_if_t` constructions. But the problem isn't completely solved.

I do not program too much in C++ and the few monstrous errors I encountered were solved by looking at the source code and a few words from the error message that jumped onto my iris at random. The difficulty to track down the root cause (the first offending usage of that template) is the major task.

For one thing, it might be desirable to perform some kind of analyses on the template definitions, inferring the semantics of these definitions, e.g., which partial instantiation is chosen in which case. Abstraction might be similar to `concept` that would be in C++2x (a kind of constraints for template parameters). If we instantiate the templates over an abstraction over the concepts, we could find out which instantiations are valid. C++ template is really a program in disguise (C++ template is Turing complete), so there is chance that one can translate program analysis methods into techniques to understand C++ templates.

On the other hand, the representation of an error message could be improved to be more human-friendly. The former is more software engineering, while the latter is more human-computer interaction.