6-21-2011

Error sentinel: A Rule-based spreadsheet program for intelligent data entry, error correction, and curation

Kyle Dewey

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation


This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Error Sentinel: A Rule-Based Spreadsheet Program for Intelligent Data Entry, Error Correction, and Curation

Kyle Dewey
Master of Science Degree in Bioinformatics
Department of Biological Sciences - College of Science
Approved June 21, 2011

Approved by:
Dr. Michael V. Osier
Dr. Gary R. Skuse
Dr. Dina L. Newman
Dr. Rajendra K. Raj
Abstract

Within the biological sciences, spreadsheets are commonly used as a data entry and storage medium. While this practice is simple and generally well understood, the unrestrained flexibility of the spreadsheet medium allows errors to accumulate and potentially propagate. Such errors impede accurate analysis, hindering research. The underlying problem is that the error correction facilities of typical spreadsheet programs are lackluster at best, if they exist at all. For this reason, Error Sentinel was developed. Error Sentinel is a spreadsheet program with programmable error correction facilities. These facilities allow users to define exactly what clean data is, along with corrections for erroneous data. Such rules are specified via a custom visual programming language. Once error correction rules are written, users inputting data need not be familiar with the rules or even have programming skills in order to utilize them. Error Sentinel can be used interactively like a typical spreadsheet program, or non-interactively as with more traditional error correction techniques. To test Error Sentinel’s real-world capabilities, it was successfully applied to the correction of the mtHaplogroups data set. This application has shown that Error Sentinel requires far less time and code to perform error correction than with previous methods. Benchmarking has shown that such gains are at only a modest cost in performance. While Error Sentinel appears quite simplistic compared to typical spreadsheet programs, its error correction facilities are robust, and it is fully capable of being applied to arbitrary data sets represented in the spreadsheet medium.
Table of Contents

- **Introduction** ........................................................................................................................................1
  - Spreadsheets and Curation.............................................................................................................1
  - Motivation........................................................................................................................................14
  - mtHaplogroups Curation.............................................................................................................19
- **Materials and Methods** ..................................................................................................................24
- **Results** ...........................................................................................................................................31
  - Ruleset Design Elements.............................................................................................................31
  - Error Sentinel................................................................................................................................32
  - Language Classification...............................................................................................................39
  - Dynamic Typing..........................................................................................................................40
  - Strong Typing.............................................................................................................................44
  - Paradigm..........................................................................................................................................45
  - Exceptions........................................................................................................................................48
  - Turing Completeness.....................................................................................................................49
  - Application to mtHaplogroups.......................................................................................................50
  - Benchmarks.....................................................................................................................................52
  - Profiling...........................................................................................................................................57
- **Discussion** ......................................................................................................................................59
  - A “Functional” Language..............................................................................................................59
  - Typing Weaknesses.......................................................................................................................66
  - Turing Completeness Consequences..........................................................................................69
  - Performance and Efficiency..........................................................................................................70
  - Miscellaneous Design Flaws.........................................................................................................92
  - Miscellaneous Implementation Flaws..........................................................................................93
  - Current State....................................................................................................................................97
  - Applications.....................................................................................................................................98
  - Extensions and Future Work..........................................................................................................98
- **References** ......................................................................................................................................99
- **Appendix** .......................................................................................................................................109
Introduction

Spreadsheets and Curation

Spreadsheets are commonly used within biology and bioinformatics to store and represent data. Spreadsheets provide a simple, flexible way to represent structured data. Indeed, the table structure spreadsheets are based on has been utilized throughout human history and has major cognitive benefits (Nardi et al. 1990). By representing each item within a set as a row, and each column as an attribute of a given item, it is possible to create an organized structure that is simple to comprehend by both people and machines.

However, excessive flexibility can be more of a liability than an asset. For instance, many spreadsheet programs will allow the user to enter in arbitrary information for any cell, regardless of whether or not that information makes sense. Table 1 shows an example of this.

*Table 1: Example showing too much flexibility in a spreadsheet.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Favorite Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>25</td>
<td>Purple</td>
</tr>
<tr>
<td>Joe</td>
<td>-10</td>
<td>Green</td>
</tr>
<tr>
<td>Jill</td>
<td>483.7</td>
<td>Notebook</td>
</tr>
<tr>
<td>Jack</td>
<td>22</td>
<td>Teal</td>
</tr>
</tbody>
</table>

Table 1 contains a number of errors that a general purpose spreadsheet program is unable to catch. A negative age should not be possible for a person, though most programs will not issue any sort of complaint to the user upon entering such an age. At the other end of the spectrum, it seems unlikely that there is a person who is nearly 500 years old, and with a fractional age on top of that. Outside of the age column, “Notebook” is clearly not a color. Worse yet, depending on the application, “Teal” and “Green” may be considered to be the same color, due to their similarities. While a spreadsheet program familiar with colors may be able to
perform the “Notebook” correction, such a program would not be able to simplify “Teal” to “Green”. It is up to the user to find and correct such errors, which is time consuming and ironically error prone in and of itself.

Though general purpose spreadsheet programs are generally flexible, there are cases where the program will find something it believes is erroneous and correct it. Though such corrections are typically proper, it is a substantial problem when they are undesired. For instance, Microsoft Excel will recognize some gene names like “OCT1” as dates, and will convert them to a more date-friendly representation like “1-Oct” (Zeeberg et al. 2004). Of course, this behavior is incorrect. The change is automatic, instantaneous, and worst of all completely silent; the user is not informed of such a change occurring. Within NCBI (http://www.ncbi.nlm.nih.gov/), gene entries exist with names like “1-Oct”, showing that this type of error does happen in practice and causes real problems.

Beyond this specific example, spreadsheet errors are a major problem in multiple disciplines (Panko 1998; Powell et al. 2007; Powell et al. 2008). The rate of errors seems alarmingly high, though the rate of errors in spreadsheets is approximately the same as the rate of errors seen in other thought-intensive tasks (Panko 1998). However, unlike with programming, there is rarely policy in place to check and correct for such errors (Powell et al., 2008). Additionally, there is a lack of information as to what such a policy should contain (Powell et al. 2007).

Clearly, something must be done to amend this situation. Though spreadsheet auditing software does exist for this purpose, such software is hardly perfect (Powell et al. 2007). In Powell et al. 2007, the software presented tended to merely look for things which can either
indicate errors or lead to errors. This heuristics-based behavior is much like the date conversion problem as seen in Zeeberg et al. 2004: it is often correct, though nothing more. Additionally, auditing tools are only intended to find errors, not to correct them.

The problem with both the spreadsheet and spreadsheet auditing programs is that they have little to no understanding of what the input data is and what it is intended to represent. With the example in Table 1, there is no way for the program to innately understand what an age or color is, and so there is no way for it to verify this information. This is arguably preferable to a program that can erroneously attach meaning to data, as shown in Zeeberg et al 2004.

I have experienced the problems regarding automatic correction (or the lack thereof) firsthand. I participate in ongoing research with Dr. Newman and Dr. Osier regarding human mitochondrial variations. Over time, a large number of unique variations has been observed by a multitude of research groups, and these have been gathered into a single uniform data set. This data set is represented as a spreadsheet, where each variation is a row, and each column represents one piece of information about a given variation.

Over time, a number of errors have crept into this data set, requiring extensive curation to fix. Many of these errors are somewhat systematic; given the pattern of a single error it is possible to find other errors that match the pattern. For example, a common typographical error is to have unnecessary spaces after an entry. To the human eye, such an error is often invisible. However, to a computer, this is significant. The entries “X” and “X “ are not equal to each other, though in reality they may be.

There are also certain oddities in the data set. Such are not technically errors, but they complicate the process of parsing by a computer. For instance, dashes (“-”) are commonly used
to indicate that there is no value for a particular column. This is problematic in two ways. For one, the dash itself is a value. As such, paradoxically, by specifying that there is no value, one inadvertently specifies a value.

A deeper problem is that this “no value” value either holds no semantic information whatsoever, or holds misleading semantic information, depending on interpretation. If the dash is treated like a value, then it appears that all entries with dashes are related. This is incorrect. If the dash is not treated like a value, then there is a “gap” in the information. While this is technically correct, there are better approaches specific to this data set.

Clearly, these are problems that must be corrected in order to facilitate parsing by a computer. Usually, a program is written that can find and correct a certain error or oddity. In the case of a dash, the program would find each column that only contained a dash, and replace the dash with a more specific “no value” value. Though this works, there are a number of disadvantages to this approach:

• The resulting program can only find this kind of error. Besides this, the program is useless.

• Most of the program is not devoted to finding the error, but instead to doing housekeeping work like opening and writing to files. Based on actual scripts written, such housekeeping work accounts for between 72% and 91% of the actual code.

• In containing such housekeeping code, the program is not very robust. It is particularly sensitive to changes in how the data is represented. For example, care must be taken regarding the file format used to store the data.

• The section that does the actual correction tends to be very verbose. Though the phrase “replace every dash with the value from the previous column” is straightforward, it takes a fair
amount of code to do exactly this in most general purpose programming languages. Based on actual code, even the simplest errors required three lines on average. Additionally, assignment statements account for almost all the error correction code, wherein the value of a column is changed.

- It takes a sizable amount of time to write such a program, considering how simple the rule for error correction is. This is unfortunately difficult to quantify, as no data was recorded regarding how much time was spent developing the original error correction scripts. Based on time stamps, it took two days to write code that corrected a total of nine minor errors.

- A fair amount of programming skill is necessary to write such code, along with prior knowledge of the general purpose programming language used. Additionally, the original error correction programs were never intended to be user-friendly, and assumed UNIX shell experience for proper usage.

- The error correction phase occurs after the error has been put into the data set. Ideally, it should be flagged as it is put in, and some sort of message should be given to the user who is attempting to add erroneous data. This would prevent the error from being incorporated at all.

To address all these problems, a different approach must be taken. Error Sentinel represents this new approach. Error Sentinel is able to find existing errors and prevent the incorporation of new errors into a spreadsheet. Error Sentinel’s facilities are far more powerful than any existing automatic correction facilities in other common spreadsheet programs, namely Microsoft’s Excel, OpenOffice.org’s Calc, and Apple’s Numbers. Ideally, Error Sentinel can find far more errors, and corrections are always correct.
Being that data sets are vastly different, it is not feasible to make Error Sentinel understand only a given data set. In the terms of a programmer, it is unacceptable to “hard code” in the rules for error correction. Instead, Error Sentinel utilizes a special purpose programming language.

The application of programming languages to spreadsheets is hardly original. In Microsoft Excel, equations and formulas are written in a sort of simple functional language (Jones et al. 2003). Though this language supports neither abstraction (subroutine definition) nor recursion (defining a problem in terms of itself), its expression-oriented nature makes it a member of the functional paradigm (Jones et al. 2003). Extensions to Microsoft Excel which are based on logical programming languages have been written, for the purpose of imparting deductive reasoning capabilities to the program (Cervesato 2007). Outside of Microsoft Excel, spreadsheets have been married with Lisp in the C32 language, which is intended for the design and implementation of GUI components (Myers 1991). Spreadsheets have even been used as a programming paradigm in and of themselves (Burnett et al., 1998; Tukiainen 2001), as with the Forms/3 visual programming language (Burnett et al., 1994; Burnett et al. 2001). There are even further paradigm divisions within the spreadsheet paradigm, complete with information regarding the difference in error rates between paradigms (Tukiainen 2001).

What separates Error Sentinel from these is that it is intended for the sole purpose of error identification and correction. As such, these functions are easy to perform, where for other spreadsheet languages such functions may be difficult to express, if not impossible.

The core of Error Sentinel consists of only two ideas: matching and replacing. Everything else in the language is merely an extension of these concepts. Rules are written that
can match data, and other rules are written that can perform replacements on said data. Within the language, these concepts are tied to entities known as matchers and replacers. Matchers and replacers are only concerned with the actual data; they are not concerned with the internal format of the data, the names of the underlying files that contain the data, etcetera. The end result is error correction rules that are very short but still capture all of the necessary information to process an error.

Though a great deal can be conveyed through this language, it is still possible to devise a rule that is too complex for the language to handle. In such a case, Error Sentinel can be extended, in a similar manner as presented in Myers 1991. It is possible to write the rule in a more expressive general purpose language, and then bind it to the error correction language in such a way that it seems to be part of the error correction language. In fact, the basic functionality of Error Sentinel has been implemented this way. The more that is written in this other language, the more feature rich the error correction language becomes.

The feasibility of using the matching and replacing concept for identifying and correcting errors has been proven by an existing project, namely ErrorChecker.pl. The program was created during research underneath Dr. Osier and Dr. Newman. This program accepts a basic language in the form of a configuration file, much like the one shown below in Table 2:
The rule shown in Table 2 is interpreted by the computer as follows: “Create a matcher by the name ‘exact’, and give it the parameter ‘-‘. To this matcher bind the replacer named ‘exact’, and give it the value of the previous column as a parameter.” To a human familiar with this language, it reads “when a column containing exactly a ‘-‘ is found, no more and no less, replace it with the exact value of the previous column.” As shown, such a rule is understandable by both a human and a computer, which is necessary for any computer programming language.

Note that there are a number of flaws with ErrorChecker.pl. It has been found to be very difficult to express rules that are anything more than trivial. Very quickly, one must write custom code in another language and incorporate it into the error correction language. If this happens too frequently, then there is little purpose in using an error correction language; one ends up writing lots of verbose code in another language, which would have happened if the error correction language had not been used anyway.

Additionally, ErrorChecker.pl is not very flexible. Custom code can be added in, but that is all that it can do. It cannot recombine existing code in many different ways, for two main reasons. For one, matchers and replacers can only accept data as arguments, where the data is either a value from the spreadsheet or a parameter (parameters are shown in Table 2). This limitation is illustrated below in Figure 1. The second, though related, problem is that rules must be specified in a strict, one matcher to one replacer manner. If data passed to the matcher

Table 2: An example rule for ErrorChecker.pl.

<table>
<thead>
<tr>
<th>Matcher</th>
<th>Matcher Parameters</th>
<th>Replacer</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>“-“</td>
<td>exact</td>
<td>previous_column</td>
</tr>
</tbody>
</table>
matches, then the corresponding replacer is executed on the given data. Behavior that is more complicated than that is impossible to specify within the confines of this language.

Figure 1: graphical view of the arguments a matcher or replacer can take in the error correction language used by ErrorChecker.pl. Note that arguments are limited to data, either specified directly as parameters or as values taken from the spreadsheet.

Ideally, the expressive power of a language should be derived not from the primitives it provides, but from the ways in which those primitives can be linked together. If there is weakness in this linking, then one must put in a “word” for every idea that could ever be conceived in the language, which is impossible. In these ways, ErrorChecker.pl is very limited in regard to its intended purpose, though it does illustrate the feasibility of a language for error correction based on matchers and replacers.

A few modifications have been performed on the language of ErrorChecker.pl to make it usable for Error Sentinel. The first modification was to allow matchers and replacers to take other matchers and replacers as parameters. This is shown visually below in Figure 2. This allows for the behaviors of different modules to be combined in arbitrary ways. In fact, by this
modification alone, it is possible to represent a rule of ErrorChecker.pl as a replacer that took one matcher and one replacer as arguments. This can be seen below in Figure 3. The replacer itself (shown in blue) would have very simple logic behind it: if the matcher (provided as a parameter, shown in green) matches on the given data, then execute the given replacer (provided as a parameter, shown in red) on the data. This shows the flexibility of Error Sentinel; one can specify the behavior of ErrorChecker.pl’s language in Error Sentinel, but it is not possible to specify the behavior of Error Sentinel in the language of ErrorChecker.pl.

Figure 2: a matcher or replacer as defined by the proposed language. Note that it is backwards compatible with matchers and replacers from the previous language, in that it can take one or more data as arguments. However, it now allows for other matchers and replacers to be specified as arguments, permitting for arbitrary behavior.
With the ability to use matchers and replacers together comes additional complexity. It is possible to create a deeply nested matcher (or replacer), one that takes another matcher as a parameter, which internally takes yet another matcher as a parameter, and so on. Depending on the situation, this same chain of matchers could appear in multiple places, with little or no modification. Such repetition is time consuming and error prone, requiring the user to specify the same construct of significant length many times.

This problem is encountered in Microsoft Excel, where the same formula may need to be copied multiple times to achieve a goal (Jones et al. 2003). To solve this problem, Error Sentinel allows for the quick and simple creation of functions. Functions allow for the encapsulation of an idea or procedure into a single reusable unit. Functions reduce repetition and the problems related to repetition.

An analogy to a function would be the verbal explanation of a step by step procedure to 100 people individually versus giving a written copy of the procedure to each of the 100 people.

*Figure 3: A replacer representing a rule of ErrorChecker.pl in the proposed language. Note that the replacer takes a matcher and another replacer as parameters, and that the logic is intrinsic only to the replacer instead of being built into the language. Everything shown is a language extension, illustrating the newfound flexibility.*
With the explanation route, more time is spent per person, and one is likely to make a mistake in the procedure's description due to fatigue. The end result is that several variations of the procedure will have been described, though perhaps only one is fully correct. However, with the copy route, everyone gets the same procedure, with far less effort. If the procedure is large, this method could save an enormous amount of time.

Through this analogy, it should be clear that functions are a necessity, not a convenience, with Error Sentinel. However, functions are somewhat of an advanced programming feature. For many languages, proper usage of functions requires special training, training that the common user simply does not have (Jones et al. 2003). Ideally, Error Sentinel should be accessible to someone without a solid background in programming, so that the user base is as large as possible. For instance, the language used in Microsoft Excel for defining equations is intended for end users, not programmers (Jones et al. 2003; Nardi et al. 1990).

With accessibility in mind, Error Sentinel is based on visuals instead of on text. Compared to traditional text-based languages, the common user can write code in a visual programming language using far less time and with fewer errors (Blackwell 2001). Additionally, visual programming languages can encourage the user to use abstractions like functions, leading to better programs overall (Blackwell 2001).

The visual aspect of the program will take the form of a dataflow. Dataflows, as the name suggests, show the flow of data through a program (Green et al. 1996). They look and behave much like workflows. Note that dataflows are a separate entity from flowcharts. Flowcharts are specialized for showing the flow of control in a program, whereas dataflows are specialized for showing the flow of data (Navarro-Prieto et al. 2001). Additionally, flowcharts
have proven to be useful only when control flow is complex, or when program text is unavailable (Navarro-Prieto et al. 2001).

An example of a dataflow is shown below in Figure 4. Lines represent the flow of data and boxes represent the modification of said data. This style is borrowed from the Spiegel Visualization Framework’s visual programming language (Bischof et al. 2006). The “Exact” matcher will say “Yes” if the given data matches with what it is checking for, and “No” otherwise. The “Exact” replacer will merely say whatever it was told to say, in this case either “Bar” or “Foo”, depending on the replacer. The “Conditional” replacer combines all previously mentioned matchers and replacers, such that if the matcher says “Yes” one replacer is executed, and if the matcher says “No” the other replacer is executed.

The function in Figure 4 represents an aggregate replacer named “fooBarSwitch”. It changes all instances of “Foo” to “Bar”, and will change all instances of everything else to “Foo”. This example illustrates some of the possible interactions between matchers and replacers. Though it is simple, far more complex interactions are possible. This example also

![Figure 4: aggregate replacer function coded with the workflow visual programming model.](image)
shows how to create a function out of other matchers and replacers. Note that the function needs only input data, and that the user does not have to be concerned with the internal components of “fooBarSwitch” in order to use “fooBarSwitch”.

**Motivation**

The ultimate purpose of this work is to remove and correct errors in the aforementioned mtHaplogroups data set. With an error-free data set in hand, it is possible for real, accurate analysis to be performed using this data. Background information needed to understand the data and what it can be used for follows.

The basic energy “currency” of cells is the molecule adenosine triphosphate (ATP). In eukaryotic cells, the majority of ATP is produced by a specific organelle, namely the mitochondrion (reviewed by Hatefi 1985 as cited by Tuppen et al. 2010). Without a steady supply of sufficient ATP, cells cannot function and quickly die. To illustrate this point, cyanide’s toxicity lies in its capability to effectively halt mitochondrion function, stopping ATP production (Li et al. 2005).

Mitochondria are maternally inherited, and contain a genome that is independent of the nuclear genome (Giles et al. 1980 as cited by Tuppen et al. 2010). The mitochondrial genome is significantly smaller than the nuclear genome, being 16,569 basepairs in length (Andrews et al. 1999 as cited by Tuppen et al. 2010). There are a number of properties that make this genome appealing for study. For one, the genome does not undergo recombination (reviewed by Tuppen et al. 2010). This property, combined with the property of maternal inheritance, makes it easier to trace genetic origins significant distances backward than with the nuclear genome. In part,
this is why the mitochondrial genome is frequently used in the study of ancient human migrations (e.g. Sykes et al. 1995; Risch et al. 1996).

Another key property of the mitochondrial genome is that it mutates more frequently than the nuclear genome (reviewed by Tuppen et al. 2010). As such, a multitude of mutations have accumulated since humans initially evolved (Torroni et al. 1996). The ability to categorize and track such mutations is another property that makes them appealing for the study of human migrations (e.g. Sykes et al. 1995; Risch et al. 1996). Additionally, such mutations can be used as human population markers. These mutations can be put into distinct groups, known as haplogroups (Torroni et al. 1996). Haplogroups can be further subdivided into subhaplogroups. A diagram detailing haplogroups, their relationship to each other, and their relationship to human migrations is shown below in Figure 5.

Figure 5: The various haplogroups and their relationships to each other. The paths are representative of human migrations, and the numbers show how many years in the past the given migration is thought to have occurred. Adapted from Ruiz-Pesini et al., 2007. Taken from http://www.mitomap.org/pub/MITOMAP/MitomapFigures/WorldMigrations.pdf
The mtHaplogroups data set holds a record of all known mitochondrial mutations, henceforth referred to as polymorphisms. From an abstract level, the data records which polymorphisms are associated with which haplogroups and subhaplogroups. Additional relevant information is also stored, including what the particular polymorphisms do, if anything. A more detailed look at the mtHaplogroups data structure is presented in the “mtHaplogroups Curation” section. As to why this data is important, there are a vast number of applications for haplogroup information. What follows is a relatively brief survey of the literature that utilizes haplogroups.

Cells often contain more than a single mitochondrion, and individual mitochondria usually contain multiple copies of their genome (reviewed by Tuppen et al. 2010). This makes mitochondrial DNA more abundant than nuclear DNA in cells, making mitochondrial DNA attractive for forensics applications. In the forensic sciences, determining the haplogroup of each suspect can be used to quickly eliminate suspects (e.g. Brandstatter et al. 2006; Medeiros et al. 2008).

As already mentioned, mitochondria are responsible for the majority of ATP production. Additionally, the mitochondrial genome holds genetic information that is vital for performing this task (reviewed Tuppen et al. 2010). With this in mind, it is no surprise that certain mitochondrial polymorphisms have been shown to affect the levels of ATP produced (e.g. Montiel-Sosa et al. 2006; Ruiz-Pesini et al., 2000). Moreover, certain haplogroups have been associated with varying levels of ATP production. The mitochondria of individuals in haplogroups T and U have been shown to produce less ATP than mitochondria of individuals in haplogroups H (e.g. Montiel-Sosa et al. 2006; Ruiz-Pesini et al., 2000). This is presumably an adaptation to colder Northern European climates (Montiel-Sosa et al. 2006); the mitochondria of
haplogroups T and U lose more energy to heat than mitochondria of haplogroup H (Montiel-Sosa et al. 2006).

Beyond raw ATP production, there have been a large number of studies showing associations between various disorders, mitochondrial polymorphisms, and haplogroups. Leber’s hereditary optic neuropathy (LHON) was one of the first disorders to be linked to a polymorphism in the mitochondrial genome (Wallace et al. 1988 as cited by Tuppen et al. 2010). Affected individuals painlessly go blind sometime between the ages of 20 and 50 (reviewed by Man et al. 2002). There are three different mitochondrial polymorphisms that have been shown to cause the disorder (Hudson et al. 2007). Interestingly enough, the disorder shows incomplete penetrance (Hudson et al. 2007), meaning that having the diseased allele does not guarantee that an individual will be affected by the disease. It has been shown that the penetrance of LHON depends both on what LHON-causing polymorphisms have been inherited and the subhaplogroup an affected individual belongs to (Hudson et al. 2007). Specifically, members of subhaplogroups J1c and J2b having LHON-causing polymorphisms are at elevated risk (Carelli et al. 2006).

In addition to mitochondrial polymorphisms affecting vision, there are also polymorphisms that affect hearing. It is well known that certain mitochondrial polymorphisms can cause permanent deafness with the correct environmental triggers. For example, certain antibiotics, including gentamicin, can cause deafness in individuals bearing a 1555A>G polymorphism (Lu et al. 2010). In much the same way as LHON, this is not a guarantee. Also in a similar way, members of certain haplogroups bearing certain polymorphisms are more likely to
become deaf if exposed to these environmental triggers. Specifically, members of haplogroup D bearing the polymorphism are at a greater risk (Lu et al. 2010).

Another hearing problem that has its roots at least partially in the mitochondrial genome is that of age-related hearing loss (ARHL). Studies have found that individuals belonging to haplogroups U and K are more prone to developing AHRL than individuals belonging to other haplogroups (Manwaring et al. 2007).

Another disorder that is related to mitochondria is that of Parkinson’s disease. Mitochondria are intrinsically involved in the disorder (e.g. Bueler 2009; Schapira 2008), though the majority of the genes that appear to be directly involved are not in the mitochondrial genome (Bueler 2009). Certain haplogroups have been shown to confer resistance to Parkinson’s disease, specifically haplogroup H (van der Walt et al. 2003). Additionally, certain mitochondrial polymorphisms have been shown to increase the risk of developing Parkinson’s disease (Huerta et al. 2005).

There have also been studies linking haplogroups to infarctions. Women of Japanese descent who are also members of haplogroup A have been shown to be at increased risk for atherothrombotic cerebral infarction, a type of stroke (Nishigaki et al. 2007). Another study has found an increased risk of early-onset heart attack in male smokers belonging to haplogroup H (Palacin et al. 2011).

A variety of studies have been performed showing links between different haplogroups and cancer types. Women in haplogroup K have been shown to be at increased risk for breast cancer, while women in haplogroup U are at decreased risk for breast cancer (Bai et al. 2007). Women in haplogroup D have also been shown to be at increased risk for endometrial cancer (Xu
et al. 2006). Men in haplogroup U have been shown to be at increased risk for both prostate and renal cancer (Booker et al. 2006), though a larger subsequent study showed that there is not a correlation between haplogroup U and prostate cancer (Mueller et al. 2009). Subhaplogroups D4a and D5 also seem to be risk factors for developing esophageal cancer in certain regions of China (Li et al., 2010).

Some associations between Alzheimer’s disease and haplogroups have been made. It has been found that men in haplogroup U are more likely to develop Alzheimer’s than men in haplogroup H (van der Walt et al., 2004). In contrast, the same study found the opposite for women: women in haplogroup U are less likely to develop Alzheimer’s than women in haplogroup H (van der Walt et al., 2004). A separate study from a different group claimed that there was no such association between haplogroup U and Alzheimer’s though the sample size was considerably smaller; however the same study suggested that members of either haplogroup H or V were at greater risk for developing Alzheimer’s (Maruszak et al. 2009).

Associations between autism and mitochondrial disorder have also been noted in the literature (Oliveira et al. 2005), though larger subsequent studies were unable to find a statistically significant connection between the two (Alvarez-Iglesias et al. 2011).

Altogether, it should be apparent that the characterization of mitochondrial polymorphisms and haplogroups has real scientific value. However, this can only be done properly when the accuracy of the data set is assured.

**mtHaplogroups Curation**

Data regarding polymorphisms relevant to mitochondrial haplogroups was previously gathered from existing databases and the literature by Eric Stevens and other members of Dr.
Newman’s lab. The raw data is stored in a single Excel spreadsheet, which is both simple and convenient for data entry. Cleaned data is currently stored in a MySQL (http://www.mysql.com/) database available via the mtHaplogroups Web interface (https://momtong.rit.edu/cgi-bin/haplogroups/haplogroups.cgi). Adding entries to the cleaned data through the interface is far more difficult than simply adding it to the spreadsheet, and such data must go through an approval process. However, the data in this format is easy to browse and search.

At the time of writing, the spreadsheet held 4,348 rows and was arranged in ten columns. Each row holds data relevant to a specific polymorphism. The names of those columns, along with descriptions, are shown below in Table 3.

Table 3: Names and descriptions of the different columns in the mtHaplogroups data set spreadsheet.

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Column Name</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Haplogroup</td>
<td>Which haplogroup an entry belongs to</td>
</tr>
<tr>
<td>2</td>
<td>Subhaplogroup</td>
<td>Which subhaplogroup an entry belongs to</td>
</tr>
<tr>
<td>3</td>
<td>Basepair</td>
<td>The basepair in the reference sequence at which the given polymorphism occurs</td>
</tr>
<tr>
<td>4</td>
<td>Reference Sequence Allele</td>
<td>The allele listed in the reference sequence for the basepair in column 3</td>
</tr>
<tr>
<td>5</td>
<td>Variant Allele</td>
<td>The new allele found at this position</td>
</tr>
<tr>
<td>6</td>
<td>Gene Locus</td>
<td>The mitochondrial locus where the polymorphism is found</td>
</tr>
<tr>
<td>7</td>
<td>Functional Change</td>
<td>The effect that the variant in column 5 elicits. For example, if a variant causes a different amino acid to be coded, then both the reference amino acid and variant amino acid will be listed.</td>
</tr>
<tr>
<td>8</td>
<td>Source</td>
<td>Literature citations to sources documenting the polymorphism</td>
</tr>
<tr>
<td>Column Number</td>
<td>Column Name</td>
<td>Column Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>9</td>
<td>Monomorphic/Polymorphic</td>
<td>Describes how specific this variant is to the haplogroup listed in column 1. A variant can be specific to one haplogroup, multiple haplogroups, or to no particular haplogroup.</td>
</tr>
<tr>
<td>10</td>
<td>Comments</td>
<td>Free text describing any other information that is relevant to the given polymorphism</td>
</tr>
</tbody>
</table>

As shown, there are partial redundancies and interdependencies in the data. For example, given only the basepair, one can unambiguously determine the reference allele if the reference sequence is on hand. Similarly, the gene locus can usually be determined using only the basepair. A graph showing dependencies and partial dependencies between columns is presented below in Figure 6.
For curation purposes, these partial dependencies are extremely valuable. Such can be used to automatically determine the consistency of data. For example, if the listed basepair is out of range of the listed locus, then the row can be flagged as erroneous. That said, accurately describing such dependencies in code can be difficult. To better illustrate this, the example of determining the reference allele from the basepair will be used.

Without having the reference sequence available, the basepair is largely useless in determining the reference allele. Using only the data, at best one could list what reference alleles have been specified to exist at a single basepair. If there is only a single entry at a given basepair, then such a list is worthless. Even if there are multiple entires at the same basepair, this

Figure 6: The various dependencies between columns in the mtHaplogroups spreadsheet. Edges from source to destination are read as “can be used to determine”. Vertices that have no dependencies are no included, namely Source, Comments, and Monomorphic/Polymorphic. Figure generated using Graphviz (http://www.graphviz.org/).
is not particularly useful. For example, assume that at basepair 100, there are three listed reference alleles: ‘A’, ‘A’, and ‘T’. One could take the route of consensus and choose ‘A’ as the correct reference allele, but there is no guarantee this is correct. Additionally, all the listed alleles could be wrong; the allele ‘C’ could very well be the correct reference allele at basepair 100.

In this case, additional information is required, namely the reference mitochondrial genome sequence. As already mentioned, this reference sequence is currently 16,569 nucleotides long (Andrews et al. 1999) This necessitates that the sequence is stored in a file external to the error correction program. Additionally, this sequence has changed before (Andrews et al. 1999 as cited by Tuppen et al. 2010), which necessitates that it is possible to quickly and easily change this information. For such a seemingly simple check, the maintenance of this additional information adds an unfortunate bit of complication which is largely extraneous to the actual error correction rule.

Ultimately, the solution to this problem was to hold the reference sequence in a relational database along with other information that was relevant to error correction. In some cases, the original relational mtHaplogroups database was simply modified to include additional information. For example, columns were added to a preexisting table of mitochondrial loci holding the starting and ending basepairs of individual loci. This solution allows for simple, efficient querying of relevant information, in addition to keeping the related information close to the dataset itself. Many of the relationships shown in Figure 6 can be verified with a single SQL statement, including the verification of both reference alleles and loci given a basepair.
ErrorChecker.pl was the logical next step beyond holding error correction information
directly in the database. Although it only supported 4 matchers (8 considering negation) and 3
replacers, it did include a matcher and a replacer capable of relational database access. These
were used to perform the aforementioned reference allele and locus checks, in addition to some
verification on variant alleles. Thereafter it was very easy to add error checking routines, as long
as the check could be represented in SQL. Other checks, however, were more difficult to
express, largely due to ErrorChecker.pl’s restrictions on forming rules.

Materials and Methods
The Scala general purpose programming language was used for both the actual
spreadsheet portion and the core of the error checking language. Two programs came out of this:
one interactive, and the other non-interactive. Both versions use the same underlying suite of
features provided by the error checking language in order to accomplish their goals. In more
technical terms, the Model-View-Controller design pattern was used, and the interactive and non-
interactive versions were merely different Views and Controllers working on the same Model.

For the interactive version of the program, Java’s Swing library was used extensively. Of
special note is Swing’s JTable, which provides most of the necessary functionality of a basic
spreadsheet out of the box.

The error checking language, though primarily visual, is internally represented in XML
(http://www.w3schools.com/xml/default.asp). Preexisting parsers for XML exist. This is a
lesson hard-learned from ErrorChecker.pl, where much of the code is devoted to parsing the error
correction language. Additionally, many of the problems of ErrorChecker.pl are due to ideas not
being able to be represented in its rigid syntax. This makes a much more expressive format like
XML preferable.
To parse and write the XML, Scala’s built-in XML libraries were used extensively. In Scala, SAX (http://sax.sourceforge.net/) is used internally to perform the initial parsing of input files. After this point, nodes are stored as a hierarchy of objects, in a form similar to the document object model (DOM) (http://www.w3schools.com/dom/default.asp). Scala’s XML libraries support both reading in and writing out of XML. This is essential for the visual programming component, which needed the ability to both read and write XML. The only foreseen issue with the choice of Scala’s XML library is that it is inefficient for large files, as it must store the entire contents of files in memory. This should not be a problem considering that programs written in Error Sentinel are not expected to be prohibitively lengthy.

For the visual programming language, it is necessary to have a fair amount of software that is dedicated to making dataflows. Originally, two preexisting solutions were noted that could do this. The first was WireIt (http://javascript.neyric.com/wireit/), a library specifically designed for making the type of workflows often seen in visual programming languages. As the library was written in JavaScript, not Java, the Mozilla Project’s Rhino (http://www.mozilla.org/rhino/) would also have to be used. Rhino allows JavaScript to run on the JVM, just like Java.

Upon further investigation, it was found that this solution was inappropriate. For one, having to use Rhino increased the complexity immensely. Additionally, WireIt still required extensive modifications in order to provide all the functionality required. To top it all off, the latest update to WireIt occurred well over a year ago as of this writing, and the project appears dead.

The other proposed solution was Prefuse (http://prefuse.org/). Although it was written in Java, this seemed to be the only advantage it had over WireIt. It required even more
modification to be useful. Additionally, the last update to it occurred in 2008, so the project is almost certainly dead.

For these reasons, a custom library for making visual programming languages was developed. The core components of the library are not specific to Sentinel, although there is a significant amount of code written on top of these components that is Sentinel-specific. Having to write this library was an unexpected hurdle, as it was nontrivial and it took a substantial amount of precious time away from the main task.

Extensibility can be a double-edged sword when it comes to implementing the basic functionality of the language, as it can become unclear as to what language to use (the core language or otherwise) to implement something. This phenomenon is seen in other extensible languages, most notably LISP, in which many basic core utilities are represented using a few primitives provided by another language. To get an idea of what primitives must be coded outside of the error correction language in Scala, the GNU Project’s implementation of Common LISP (GCL, from http://www.gnu.org/software/gcl/) was referenced. The implementation features a variety of functions coded both in LISP and in C, where the C functions represent the most basic primitives necessary for a fully functional language. The C functions show which primitives are necessary in order to implement other things.

The majority of the C functions encountered were implemented as closely as possible in Sentinel. Functions that were specific to LISP behaviors, such as looping and array creation, were not implemented. Most of the intentionally unimplemented functions represent concepts that are felt to be too complicated for the general user with no programming experience.
Beyond the functions adapted from GCL, several additional functions were added to the core of the Error Sentinel language. Functions for regular expression matching were added, namely Regex and NotRegex. The aforementioned functions utilize preexisting regular expression routines in Java and Scala, found in java.util.regex and scala.util.matching, respectively. These functions determine if a given datum does or does not match a given regular expression.

Functions for relational database access were also added. These functions can be used to both determine if an item exists in a database and to retrieve items from a database. These functions are built on top of the abstraction layer provided by Java’s Database Connectivity (JDBC), and so the database functions are theoretically applicable to any relational database for which a JDBC driver has been written.

For the specific application of Error Sentinel to the mtHaplogroups data set, some data was stored in a relational database. The actual data stored in the database is elaborated on in the results section. The database used was SQLite3 (http://www.sqlite.org), so chosen because of its simplicity and relatively good performance. The JDBC driver used to connect to the database was SQLiteJDBC (http://www.zentus.com/sqlitejdbc/).

A variety of benchmarks were performed in an attempt to gauge the real-world performance of both ErrorChecker.pl and Error Sentinel. For all of these benchmarks, five sequential runs of the same program were performed on the same input. Multiple runs were used in order to prime any caches that are out of direct control. Of these five runs, the most minimal runtime was plotted. The minimum was used as opposed to the average because this value most accurately represents how long the program took (Kaminsky 2010). Being that all the code is
deterministic, theoretically each run should result in the exact same set of instructions being executed in the exact same order. These instructions should all take the exact same amount of time. As such, any time variations are due not to variability in the program, but rather other programs interfering with the program being benchmarked (Kaminsky 2010).

The first of these benchmarks is the most realistic. It pits Error Sentinel against ErrorChecker.pl, using the entire mtHaplogroups data set. For the actual error correction code, the last ruleset written in ErrorChecker.pl was used. This ruleset was previously applied to the data, and the current data shown at https://momtong.rit.edu/cgi-bin/haplogroups/haplogroups.cgi was cleaned with it. It is too long to describe it completely, though it has been included as an appendix. Without too much detail, it corrects certain typographical errors that were found to exist in the dataset. It also converts citation information in a free text format to numerical IDs, and it validates reference alleles, variant alleles, and loci with SQL database queries.

Being that Error Sentinel rules are directly incompatible with ErrorChecker.pl rules, a small program was written to automatically perform the conversion, namely RuleConverter (found in sentinel.utils.legacy). This program achieves a one-to-one transformation of recognized ErrorChecker.pl rules to Error Sentinel rules. The remaining rules that were not recognized by RuleConverter were implemented by hand in the same one-to-one fashion. Every attempt was made to make the rulesets comparable. The resulting ruleset for both ErrorChecker.pl and Error Sentinel contained 14 rules defining what good data looked like and 85 rules describing corrections for erroneous data. Testing has shown that both rulesets achieve the same output for the same input.
A modified version of this ruleset was also benchmarked. This modified ruleset was identical except for checks that used SQL statements and relational databases, which were removed. This was done because execution of such statements obviously takes a significant amount of time, and this time is spent outside of the actual error correction frameworks. Instead it is spent in the relational database drivers and the database itself, which is not the target of the benchmark. This modified ruleset had the same 85 error correction rules, but 3 of the 14 good data rules were removed.

In addition to this real world benchmark, another benchmark was performed using only a single good data and error correction rule. The good data rule specifies that a subhaplogroup cannot be a dash. The error correction rule will replace dashes in the subhaplogroup column with the haplogroup column’s value. This is based on a subset of rules taken from the more realistic benchmarking setup. This was chosen because it is simple to implement in other languages and it requires values across more than a single column, namely the haplogroup.

This dash replacement setup was implemented in both ErrorChecker.pl and Error Sentinel. In addition, single use programs were written in Scala, Java, and Perl that can perform this error correction. These programs were used to try to measure how much of a performance penalty is taken by using an error correction framework. These were also used to try to determine how much overhead is simply due to the language chosen, since ErrorChecker.pl is written in Perl and Error Sentinel is written in Scala.

A summation of all the benchmarking setups is shown below in Table 4. Henceforth, these setups will usually be referred to by their names, for reasons of brevity.
Table 4: The various benchmarking setups used for ErrorChecker.pl and Error Sentinel. Note that although rulesets are not directly compatible between the two programs, great care has been taken to achieve equivalent 1:1 translations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Set</th>
<th>Ruleset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire Full with Database</td>
<td>Entire mtHaplogroups data</td>
<td>The “real world” ruleset, featuring 14 good data rules and 85 error correction rules.</td>
</tr>
<tr>
<td>Entire Full without Database</td>
<td>Entire mtHaplogroups data</td>
<td>The “real world” ruleset, without database-based checks. This features 11 good data rules and 85 error correction rules.</td>
</tr>
<tr>
<td>Entire Dash</td>
<td>Entire mtHaplogroups data</td>
<td>The dash replacement ruleset, which replaces dashes in the subhaplogroup column with the value in the haplogroup column. This features a single good data rule and a single error correction rule.</td>
</tr>
<tr>
<td>Single Dash</td>
<td>Single line of the mtHaplogroups data. The line has a dash in the subhaplogroup column.</td>
<td>The same dash replacement ruleset used in “Entire Dash”, without modification.</td>
</tr>
</tbody>
</table>

Both ErrorChecker.pl and the non-interactive version of Error Sentinel were profiled in an attempt to reveal any outstanding inefficiencies. For all profiling information, the “Entire Full without Database” setup was used. For ErrorChecker.pl, this profiling was performed using the Devel::NYTProf module (http://search.cpan.org/dist/Devel-NYTProf/lib/Devel/NYTProf.pm), retrieved from the Comprehensive Perl Archive Network (CPAN). All the defaults were used for this profiling. For Error Sentinel, profiling data was generated using the Java Virtual Machine’s built in Heap/CPU Profiling Tool (JVM HPROF). The CPU profiling method used was “times”.
“Times” records the absolute number of method calls that were made, along with the amount of time spent on each call. The stack depth used for traces was 30.

Results

Ruleset Design Elements

Although the syntax and underlying implementation of rulesets is vastly different from that of ErrorChecker.pl, there are still a number of commonalities. Matchers and replacers are still the components that ultimately do work. As with ErrorChecker.pl, rulesets are broken down into two separate sections, namely “good data” rules and “error correction” rules.

Good data rules describe what clean data looks like, and each rule consists of a single matcher. There can be any number of good data rules. In order to be marked clean, input data must match all good data rules. If no good data rules are listed, then the data is assumed to be clean.

Error correction rules describe corrections that can be performed on erroneous data to make such data clean. Each of these rules consists of a combination of one matcher and one replacer. Unlike the good data rules, the order of error correction rules is important. When input data fails one of the good data rules, it is passed through each of the error correction rules, in an order defined by the user. If the input erroneous data matches the error correction rule’s matcher, then the rule’s corresponding replacer is enacted. The resulting output of the enacted replacer is treated as a correction. If data marked erroneous by the good data rules matches none of the correction rules, then the data is flagged as being uncorrectable, at least by automatic means.
Error Sentinel

There are two versions of Error Sentinel that have been developed. The first is extremely similar to ErrorChecker.pl, and is intended for largely the same purposes. This version is text-based, and does not allow for interactive data entry. Using a provided ruleset, it scans input data. This data is partitioned into one of two output spreadsheets, based upon the results of the ruleset. One of these sheets is for cleaned data, and the other sheet is for uncorrectable data. All cells marked clean by the good data rules are placed into the cleaned spreadsheet. Additionally, if there is a valid error correction rule for an otherwise unclean cell, then the correction rule is applied and the resulting correction written to the clean spreadsheet. If a cell has been marked as erroneous by the good data rules and there are no matching correction rules, then the cell is written to the spreadsheet for uncorrectable data. Additionally, any messages that are relevant to why the data was marked as unclean and uncorrectable are written to a third output spreadsheet.

The second version of Error Sentinel developed is intended to look and behave much like a typical spreadsheet program. It allows for interactive data entry, and the validity of input data can be assessed on the fly based on a user-defined ruleset. An overall look at this interactive version with the mtHaplogroups data set open can be seen below in Figure 7.
In this interactive version, data marked as clean is shown in green. This can be seen with the selected cell in Figure 7. Data marked as unclean but with correction information available is shown in yellow. The actual correction is shown in the “Sentinel Information Panel”, and also can be seen when selecting the menu option to perform a correction. An example showing a selected cell with an available correction is shown below in Figure 8.

Figure 7: Overall look at the interactive version of Error Sentinel. The mtHaplogroups data set is shown.
In the example shown in Figure 8, “J2A” is not a valid subhaplogroup, but “J2a” is. There is a correction rule that can make this fix using the “GetDifferentCaseSubhaplogroup” replacer, which is described later in the results. Note that the correction shown is intended to be a guide; the user can select the ruleset-defined correction or specify an arbitrary correction. Such behavior avoids the auto-correction problems seen in Zeeberg et al. (2004).

As for data marked unclean by the good data rules but without a matching error correction rule, it is shown in red. This is shown below in Figure 9.
In the example shown in Figure 9, the column “bp Position” is supposed to contain only integers. The value “8281-8289” clearly cannot be converted into a single integer, and so it has been marked as erroneous. Additionally, there are no rules for making this sort of correction, and so it has been shown in red. Note that although Error Sentinel has marked this data as erroneous, it does not attempt to modify or remove the data in any way; it is still the user’s responsibility to correct the error, and until then the error coexists with other data.

In addition to these basic features, the interactive version of Error Sentinel supports two additional features of interest. The first of these allows for the colors of all cells to be shown at once, as if each cell were somehow selected both individually but simultaneously. This feature is accessible via “View->Show Cell Errors”. An example of this feature is shown below in Figure 10.

![Image of Error Sentinel Panel with the option to show cell errors]
Despite its usefulness, the “Show Cell Errors” command hides some important information. Occasionally, direct execution of the error correction ruleset may not reveal all problems in the data. For example, consider the relationship between the haplogroup and the subhaplogroup shown in Figure 6. Depending on the particular ruleset, it may be possible for one of these two to be marked differently than the other. The haplogroup could very well be marked as clean but the subhaplogroup as unclean, or vice versa. Something similar to this problem can be seen in Figure 10 for some of the later entries in the spreadsheet. The haplogroup has been flagged as being correctable (yellow), but the subhaplogroup has been flagged as unclean without a correction (red). Given the relationship between these two values in Figure 6, this is likely indicative of a larger problem. Indeed, this is the case for this example.

Figure 10: The result of using the “Show Cell Errors” command with the mtHaplogroups data set.
The haplogroup “R0”, which is the value of all the later yellow haplogroups in Figure 10, is no longer considered valid in the data. Instead, it is referred to as “Pre HV1”. This change in the haplogroup also demands a change in the subhaplogroup to preserve validity of the entry as a whole. In short, a cell considered clean through direct execution of the error correction ruleset could still be unclean if it is related to an unclean value.

Because of this problem, there is also a modified version of the “Show Cell Errors” command, namely the “Show Cell Errors with Parameters” command. This is accessible via “View->Show Cell Errors with Parameters”. With this command, a cell dons the “worst” color between itself and all related cells. For example, if the cell is marked clean, but it is associated with an unclean cell, then the cell is also marked as unclean. The result of this command on the same data in Figure 10 is shown below in Figure 11.
As Figure 11 shows, additional cells have been marked as erroneous, including some which were previously marked clean. The later haplogroup entries have all been marked as uncorrectable, as their related subhaplogroups were uncorrectable. Additionally, the relationship between the basepair and the reference allele shown in Figure 6 has been illustrated. Although the basepairs for two of the entries were previously marked as clean, they are related to erroneous reference sequence alleles, and so they have also been marked as erroneous and uncorrectable.

Since the “Show Cell Errors with Parameters” may mark valid data as invalid, as with the aforementioned basepairs, it is not intended to be used without discretion. It is intended to illustrate relationships between data, and potential consistency issues. A reasonable way of
thinking about the “Show Cell Errors with Parameters” is that it shows that data which has “at most” the given problems, based on the ruleset. By contrast, “Show Cell Errors” shows data with “at least” the given problems, based on the ruleset. The reality is somewhere in between, and requires human intervention to solve. For this matter, Error Sentinel is not intended to replace the user component, but rather assist the user in making corrections.

**Language Classification**

There are a large variety of different criteria which can be used to classify programming languages. Additionally, given that the Error Sentinel language is a visual programming language, there are even more criteria that can be considered than usual. To illustrate the sheer volume of potential criteria, Burnett and Baker (1994) propose a system just for classifying papers written about visual programming languages (VPLs). Kiper et al. (1997) identifies 18 key criteria for evaluating VPLs, and uses them to evaluate the VPLs LabVIEW and ProGraph. However, all of the proposed criteria are subjective in nature. For example, one of the criteria is “Ease for programmers”. The author’s actual wording for this criterion is “Somewhat less easy” for both LabVIEW and ProGraph.

Being that an objective classification using the proposed criteria in Kiper et al. (1997) would require a sufficiently large case study, such will not be used. Additionally, given the large volume of potential criteria just for programming languages in general, no attempt will be made at an exhaustive list. Instead, evaluation criteria which are most useful for the application of Error Sentinel have been chosen. Such will be described henceforth.
**Dynamic Typing**

In mathematics and computer science, the phrase “data type” refers to a set of values and the valid operations that can be performed on those values (Sebesta 2007). Frequently, many values have some concept of a type, even outside of programming languages. For instance, consider the following four numbers: 1, -2, 3.5, and π. 1 is a natural number, as natural numbers include all positive integers and 0. -2 is not a natural number as it is negative, though is it still an integer. 3.5 is a merely a rational number because of the decimal, and π is an irrational number.

In and of itself, this kind of classification is not particularly useful. However, when the applicable operations are included along with the values, the utility becomes apparent. For example, consider the greatest common divisor (GCD). The GCD is applicable only to non-zero integers. This is because it is relevant only to rational numbers which, by definition, have a non-zero integer as a denominator. Additionally, the GCD returns a positive integer, again due to its definition. It is not possible to get the GCD of two numbers when either one of the numbers is irrational, as this would be nonsensical. In much the same way, it would be nonsensical to return anything but a positive integer.

This is where typing comes into play. Formally, typing shows that finding the GCD with π as a parameter is not permitted. Applied to programming languages, any attempt to find the GCD when a parameter is π would result in a type error: the expected types differed from the types used. Such errors are preferable to quietly using nonsensical data. For example, assembly languages do not differentiate between integers, real numbers, or even code, which can result in severe and difficult to find bugs.
In general, there are two ways to perform typing: statically and dynamically (Sebesta 2007). In statically typed languages, type errors are found before the program is actually executed. In dynamically typed languages, type errors are found as the program is executed. There are a number of advantages and disadvantages to both strategies. For example, statically checked code typically runs faster than dynamically checked code, as no type-checking overhead occurs while the program is running. However, it is sometimes difficult to express ideas in statically typed languages, as these potentially very abstract ideas must be represented with concrete types (Sebesta 2007).

With this in mind, Error Sentinel is overall a dynamically typed language. It was felt that static typing was too conforming for someone inexperienced with programming. Worse yet, it adds another layer of complication, increasing the time needed to learn the language. There is, however, a severe drawback to this dynamically typed approach. Code written in dynamically typed languages is generally less robust than code written in statically typed languages (Sebesta 2007). Since errors are found as they occur, it is possible to have hidden bugs in code that just need the right input types to be triggered.

Considering that Sentinel is a language specific for error correction, it was realized that type errors could be useful for this purpose. For instance, if Sentinel reads in user data that it expects to be an integer, but it is instead presented with a string, then this is indicative of an error in the data itself. This check is done automatically and “for free” from the standpoint of a user; the user does not have to explicitly perform these checks, as Sentinel implicitly must perform them to execute user code.
An important point about data in a spreadsheet is that all data is seen as a bunch of strings. As such, it must be possible to convert these strings from one type to another. The types that Sentinel recognizes, along with the possible interconversions, are shown below in Table 5.

*Table 5: The types Sentinel recognizes, along with type conversion information.*

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Always Converts To</th>
<th>Possibly Converts To</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>A string of characters</td>
<td>Replacer</td>
<td>Character, Real, Integer</td>
</tr>
<tr>
<td>Character</td>
<td>A single character</td>
<td>Replacer, String</td>
<td>Real, Integer</td>
</tr>
<tr>
<td>Real</td>
<td>A real number (floating-point value)</td>
<td>Replacer, String, Integer</td>
<td>Character</td>
</tr>
<tr>
<td>Integer</td>
<td>An integer</td>
<td>Replacer, String, Real</td>
<td>Character</td>
</tr>
<tr>
<td>Replacer</td>
<td>Returns something based on user-defined rules</td>
<td>String</td>
<td>Integer, Real, Character</td>
</tr>
<tr>
<td>Matcher</td>
<td>Determines if input information matches a user-defined pattern</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The type system of Sentinel is designed to be easy to learn, but unnecessary for basic usage. During function creation, the VPL infers the types of inputs based on the input types of the functions used within a function. This inference can be seen below in Figure 12. Although the type inference mechanism prevents incorrect types from being passed to inner functions, it cannot predict additional semantic value that a user wants for a function.
As an example of additional semantic value, consider another example. Say a user wants to multiply two numbers. One is a constant and the other is passed as input to a function. The constant is a Real, so the user must use Real*, which multiplies real numbers. However, the user may only want an integer as a parameter. In this case, Sentinel’s type inference would choose
the weaker Real as the parameter type, which is not what the user desires. As shown in Figure 12C, the user can still change this type; the problem is that the default is not semantically correct in this case.

It is expected that advanced users will take advantage of the typing system, adding additional semantic value as described above. In this way, it may be possible to do basic error correction without actual code.

Strong Typing

There is another important typing property of Error Sentinel that is important to mention. Typing can be referred to as “strong” or “weak”, and the difference can be substantial. Unfortunately, these two terms are poorly defined in the literature, if at all (Sebesta 2007). For example, Madsen et al. (1990) defines weak typing as “...the type of an expression carries little or no information.” The implied definition of strong typing from this definition of weak typing is that an expression carries all typing information along with it. In contrast to this implied definition, Berry et al. (1979) defines a strongly typed language as one in which “...each value has a unique type.” Although these definitions share some overlap, they are different. For example, the number “1” is both an integer and a real number. By the definition in Madsen et al. (1990), this is perfectly fine. However, by the definition in Berry et al. (1979), this cannot be so; ultimately, “1” must either be an integer or a real number. (Note that Berry et al. (1979) is concerned with type equivalence, and would likely consider integers to be equivalent to real numbers for certain operations, and vice-versa.)

Although it appears that the definition by Madsen et al. (1990) is more realistic by this argument, there are other problems with it. For example, Madsen et al. (1990) claims the
following: “Strong typing also implies that type checking can be performed during compilation, but utilization of weak typing aspects has to be checked during run-time.” Considering this quote, Madsen et al. (1990) considers strong typing to be largely equivalent to static typing, and weak typing to be equivalent to dynamic typing. If this reflected reality, then there would be no use for additional terms to describe the exact same thing.

With this in mind, the definition used henceforth is that used by Tucker et al., namely that a strongly typed system can detect all type errors before invalid types are used. This definition is consistent with both Sebesta (2007) and Berry et al. (1979). It is also consistent with Madsen et al. (1990) if one considers “implies” to mean no guarantee.

Using this definition, Error Sentinel is a strongly typed language. Any attempt to use a value requires that types are compatible, as shown in Table 5. If an attempt is made to use an incompatible type that cannot be converted to a compatible type, a type error occurs. Such an error aborts execution, halting any attempt to use nonsensical data. Given that type errors can indicate errors in the underlying spreadsheet, the decision to be strongly typed is obvious.

Paradigm

Another basic way to classify languages is based on fitting it to a programming paradigm. Although many languages encompass elements of more than one paradigm, there is typically a certain paradigm that is favored for a given language. It is generally accepted that there are four programming paradigms: imperative, object-oriented, functional, and logical (Tucker et al. 2007). A very brief summary of these paradigms follows.
Imperative languages accomplish their work through a carefully ordered series of statements. Each statement causes certain side effects to occur, and a result comes about by building up side effects (Backus 1978).

Object-oriented languages are centered, not surprisingly, around objects. Objects can hold data, typically referred to as state, and can perform certain actions relating to this data. In object-oriented programming, work is accomplished via message passing between objects (Tucker et al. 2007).

In functional programming, functions act as independent units that can work on input parameters (Hughes 1989). Functions can be passed as parameters to other functions, and can be returned from functions (Hughes 1989). In “pure” functional programming, there are no side effects (Pippenger 1997), in stark contrast to side effect-ridden imperative programming. Functional programs tend to be shorter and less time-consuming to write than their imperative counterparts (Hudak et al. 1994).

In logical programming, the programmer is concerned only with logical correctness, according to the rules of prepositional logic (Sebesta 2007). The logical paradigm is perhaps the farthest removed from the rest of the paradigms and tends to be employed only in special cases.

With all these paradigms in mind, it is possible to classify Sentinel. Sentinel is most clearly a functional language. Matchers and replacers act as functions, and all functions must either be a matcher or a replacer (but not both). Although Sentinel programs look like dataflows and usually work like dataflows, this type of thinking is an oversimplification. Consider the function shown below in Figure 13. Looking the bottom right node displaying 35, it is not
actually possible to determine what the “correct” way of looking at the input is. Using LISP-style syntax, one could look at the input in two different ways:

1.) \((\ast\ 5\ (+\ 3\ 4\ ))\)
2.) \((\text{lambda}()\ \(\ast\ 5\ (+\ 3\ 4\ ))\)

In other words, the input could be either the result of the input function or the input function itself, depending on the input type of the target function. In this context, replacers act as functions. Type conversions from values to replacers simply wrap the value in a replacer that returns the value. In converting replacers to values, the function is called to get the value, and that value is returned.

![Diagram](image)

*Figure 13: An arbitrary function used to illustrate the different ways to look at inputs to functions. The input to the bottom right node could be read as either a value or a function, depending on the input type.*

Sentinel is a functional language in a different aspect, as well. All of Sentinel’s built-in functions, with the exception of the random number generating “Random” function, are purely functional. With these pure functions, calling the same function with the same parameters will get the same value twice, without exception. If there were side effects, this would not
necessarily be true. The core was intentionally designed in this manner, as it is easier to reason about purely functional programs (Backus 1978).

Exceptions

Both matchers and replacers are capable of throwing exceptions. Such exceptions are merely piggybacked on Scala’s existing exception handling features, so their implementation was “for free”. Exceptions were deemed a necessity early in the planning phase of Error Sentinel, largely due to its stringent typing rules. Without exceptions, the validity of type conversions would have to be done explicitly, which is counter to being a simple to use language. Not to mention the possibility of neglecting to perform such a check, which could lead to nonsensical values being used and ultimately errors propagated.

Exceptions can also be thrown explicitly using the functions MatchException and ReplaceException. This was also deemed a necessity early on. For example, consider an attempt to determine the greatest common divisor of 0. Since 0 is an integer, this seems to be a perfectly valid request given Error Sentinel’s typing system. However, the result is mathematically undefined, and so any actual return value is worthless. Without the ability to throw exceptions or the like, automatically triggering an error and an appropriate response is not possible.

Note that explicitly thrown exceptions must have messages attached as to why errors occurred. Additionally, all exceptions thrown in Error Sentinel’s code have messages attached regarding why errors occurred. These messages are intended to be of help to end users, and the intention is that they can be used to show exactly why erroneous data is not valid. Such messages can then be used both to assist curation and to refine error correction code.
Turing Completeness

An important property of programming languages is that of Turing completeness. A complete explanation of what it means to be Turing complete is beyond the scope of this discussion. A short explanation is that a Turing complete language is able to compute any number in a finite number of steps (Turing 1936). It is further conjectured that this can be applied to more than just numbers (Tucker et al. 2007). That is, any conceivable algorithm can be implemented in a Turing complete language (Tucker et al. 2007).

Using the ability to compute numbers as a guide, a Turing complete language must at least provide mechanisms to do comparisons and to repeat specific instructions (Turing 1936). Error Sentinel does have mechanisms to do comparisons, but it does not have the capability to iterate on command. In the non-interactive version of Error Sentinel, each cell is iterated over, one at a time. This proceeds from left to right, one entire row at a time. This is predefined and cannot be changed. In the interactive version, the user’s activity determines what cells are evaluated. Short of requesting the user perform a specific action and assuming such will be done, there is no way to automatically evaluate cells. For these reasons, Error Sentinel is not Turing complete.

This determination may seem strange, especially considering that a related spreadsheet VPL, namely Forms/3, is Turing complete (DuPuis et al. 1997). However, note that Forms/3 has functions that support iteration over cells of a spreadsheet on demand, along with recursion support (DuPuis et al. 1997). Error Sentinel lacks these necessary functions and capabilities, and so it is not Turing complete.
Application to mtHaplogroups

Error Sentinel was successfully applied to the correction of errors in the mtHaplogroups data set. Additional functions were created specifically for mtHaplogroups. All of these functions were composed from existing Error Sentinel functions; these functions represent the first functions written in “pure” Error Sentinel, without external functions. A description of the functions created is presented below in Table 6.

Table 6: The different functions specifically created for curation of the mtHaplogroups dataset. All functions were written in the Error Sentinel language, as opposed to an external language that is capable of extending Error Sentinel.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Parameter(s)</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsHaplogroup</td>
<td>haplogroup</td>
<td>Matcher that determines if the given datum looks like a haplogroup. This is verified with an SQL statement that queries the database for haplogroup names.</td>
</tr>
<tr>
<td>IsSubhaplogroup</td>
<td>subhaplogroup</td>
<td>Matcher that determines if the given datum looks like a subhaplogroup. This is verified with an SQL statement that queries the database for subhaplogroup names.</td>
</tr>
<tr>
<td>IsReferenceAllele</td>
<td>basepair, reference allele</td>
<td>Matcher that determines if the given reference allele matches the reference allele listed in the reference sequence for the given basepair. This is performed with an SQL statement.</td>
</tr>
<tr>
<td>IsVariantAllele</td>
<td>variant allele</td>
<td>Matcher that determines if the given datum looks like a variant allele. This is verified with an SQL statement that queries the database for known variant alleles.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Function Parameter(s)</td>
<td>Function Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IsLocus</td>
<td>basepair, locus</td>
<td>Matcher that determines if the given locus name is known and that it contains the given basepair. This is performed via an SQL statement.</td>
</tr>
<tr>
<td>MakeHaplogroup</td>
<td>subhaplogroup</td>
<td>Replacer that makes the given subhaplogroup “look” like a haplogroup. This merely appends an asterisk to it.</td>
</tr>
<tr>
<td>IsDifferentCaseSubhaplogroup</td>
<td>subhaplogroup</td>
<td>Matcher that determines if there is a subhaplogroup in the database that matches the given subhaplogroup, but of a different case. This is performed with an SQL statement.</td>
</tr>
<tr>
<td>GetDifferentCaseSubhaplogroup</td>
<td>subhaplogroup</td>
<td>Replacer that works much like IsDifferentCaseSubhaplogroup, but it will return the subhaplogroup of the appropriate case. If there is not a subhaplogroup of a different case, this throws an exception.</td>
</tr>
</tbody>
</table>

The above functions only verify a subset of the relationships shown in Figure 6. However, they are still able to verify quite a bit. Additionally, some of the relationships shown in Figure 6 are not very strong. For example, given only the haplogroup, the subhaplogroup may not be narrowed down sufficiently. Haplogroup “Pre HV1” has only a single associated subhaplogroup with it in the mtHaplogroups dataset, but haplogroup “H” has 117 associated subhaplogroups. Granted, this still cuts the possible values space by more than 2/3, but this is still largely insufficient. Additionally, for all the rounds of curation that were performed on mtHaplogroups, not a single error was found due to an incorrect subhaplogroup for a haplogroup; such is a gross error and is immediately recognizable.
Benchmarks

The benchmarking results of ErrorChecker.pl and Error Sentinel using both the “Entire Full with Database” and “Entire Full without Database” setups are shown below in Figure 14.

**Figure 14: Runtimes of ErrorChecker.pl and Error Sentinel on the Entire mtHaplogroups Data**

The benchmarking results of the “Entire Dash” setup and the “Single Dash” setup are shown below in Figures 15 and 16.
Figure 15: Runtimes of various programs on the replacement of a dash in the subhaplogroup column with the name of the haplogroup. The entire mtHaplogroups dataset was used.
After these benchmarks were performed, a second look was taken at ErrorChecker.pl. Unlike the non-interactive Error Sentinel, ErrorChecker.pl enforces that corrections are themselves valid according to the good data matchers. The side effect of this behavior is that any erroneous cell with a correction rule must be validated twice, hurting performance. Being that Error Sentinel does not have such behavior, the comparison made in Figure 14 is not equal. Because of this, ErrorChecker.pl was modified to not perform post-correction verification, in an attempt to make it behave more like Error Sentinel. Benchmarks were performed on this
modified version using the same rule sets and data. The results of these benchmarks are shown below in Figures 17 through 19.

**Figure 17: Runtimes of the “Entire Full with Database” and the “Entire Full without Database” setups for ErrorChecker.pl and Error Sentinel. ErrorChecker.pl has been modified to behave more like Error Sentinel, unlike the data reported in Figure 14.**
Figure 18: Runtimes of processing the whole mtHaplogroups data set with a ruleset that is applicable to only dashes in the subhaplogroup column. A version of ErrorChecker.pl that has been modified to behave more like Error Sentinel was used. The details of the data and ruleset are identical to those of Figure 15.
Profiling

The full results of profiling are extremely lengthy. As such, only the 10 most CPU-intensive portions have been listed for both programs. The results for ErrorChecker.pl and Error Sentinel are presented below in Tables 7 and 8, respectively. Note that it is not possible to accurately interpret these results without both the code and an understanding of the code. Such an interpretation is presented in the discussion section. All recorded runtime percentages refer to only the time spent in the given function, and do not include time spent in any functions called by the given function.

Figure 19: Runtimes of processing the whole mtHaplogroups data set with a ruleset that is applicable to only dashes in the subhaplogroup column. A version of ErrorChecker.pl that has been modified to behave more like Error Sentinel was used. The details of the data and ruleset are identical to those of Figure 16.
Table 7: Profiling information for the ten most CPU-intensive functions for the modified version of ErrorChecker.pl. Note that “__ANON__” is the notation used by Devel::NYTProf for an anonymous (unnamed) function.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Number of Times Called</th>
<th>Percent of Execution Time Spent in Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowMatches</td>
<td>170478</td>
<td>20.1</td>
</tr>
<tr>
<td>concatArgumentValues</td>
<td>174846</td>
<td>10.8</td>
</tr>
<tr>
<td>doRule</td>
<td>132068</td>
<td>9.9</td>
</tr>
<tr>
<td>matchValue</td>
<td>140233</td>
<td>7.7</td>
</tr>
<tr>
<td>prepArguments</td>
<td>174846</td>
<td>7.6</td>
</tr>
<tr>
<td>getDesiredColumns</td>
<td>170478</td>
<td>5.7</td>
</tr>
<tr>
<td><strong>ANON</strong> [ArgumentTaker.pm:62]</td>
<td>174846</td>
<td>4.6</td>
</tr>
<tr>
<td>_doReplacements</td>
<td>5368</td>
<td>4</td>
</tr>
<tr>
<td>rowIsGood</td>
<td>25404</td>
<td>2.7</td>
</tr>
<tr>
<td>matchValue</td>
<td>30245</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Table 8: Profiling information for the ten most CPU-intensive functions for Error Sentinel. Note that “$anon$” is the notation used by HPROF for anonymous classes, and “$anonfun$” is the notation used for anonymous functions.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Number of Times Called</th>
<th>Percent of Execution Time Spent in Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Object.wait</td>
<td>72</td>
<td>33.15</td>
</tr>
<tr>
<td>java.lang.ref.ReferenceQueue.remove</td>
<td>867</td>
<td>32.82</td>
</tr>
<tr>
<td>scala.StringBuilder.append</td>
<td>1043568</td>
<td>0.49</td>
</tr>
<tr>
<td>scala.collection.mutable.FlatHashTable$class.addEntry</td>
<td>460994</td>
<td>0.33</td>
</tr>
<tr>
<td>scala.collection.mutable.FlatHashTable$class.addEntry</td>
<td>300081</td>
<td>0.28</td>
</tr>
<tr>
<td>Function Name</td>
<td>Number of Times Called</td>
<td>Percent of Execution Time Spent in Function</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>sentinel.io.input.csv.ParseCSV$anonfun$parseUnquotedToken$1.apply</td>
<td>304384</td>
<td>0.25</td>
</tr>
<tr>
<td>scala.RandomAccessSeq$anon$13.next</td>
<td>492143</td>
<td>0.22</td>
</tr>
<tr>
<td>scala.runtime.RichString.&lt;init&gt;</td>
<td>360967</td>
<td>0.21</td>
</tr>
<tr>
<td>scala.collection.mutable.ListBuffer.$plus$eq</td>
<td>361816</td>
<td>0.2</td>
</tr>
<tr>
<td>scala.collection.mutable.FlatHashMap$e$class.containsEntry</td>
<td>300081</td>
<td>0.2</td>
</tr>
</tbody>
</table>

**Discussion**

A “Functional” Language

It was stated in the results that Sentinel is a functional language. There are, however, a few caveats to this. The first is in its usage of functions as parameters or return values (referred to as first class objects from now on). While Sentinel’s type system does work as described, nothing in the core actually uses functions this way. There are multiple reasons for this. For one, Sentinel’s core is essentially GCL’s core, sans any functions related to arrays, list processing, or anything specific to LISP. Without these LISP-specific routines, there is nothing in GCL’s core that uses functions as first class objects. As such, there was nothing implemented that uses these.

There is another reason why nothing in the core uses functions as first class objects, hinted at in the results. Sentinel code looks and feels like dataflows, which are fairly easy to understand. They do not work exactly like dataflows, however, when functions are used as first class objects. This adds cognitive complexity, which is counterintuitive to Sentinel being an “easy” programming language.
There is yet a third reason why functions are not used as first class objects in the core, related more to functional programming in general. To treat functions as first class objects is a different way of thinking. Even among programmers not everyone “gets” the idea of functions as first class objects. For this reason, in the interest of avoiding complexity, the core does not utilize functions as first class objects.

In addition to all these reasons, there is a severe weakness in functions as first class objects in Sentinel. Any function used in this manner cannot require parameters to work. In other words, using LISP-style syntax, (lambda () (+ 2 3)) is a possible function, but not (lambda (x) (+ 2 x)), as the second function requires the parameter x. As Sentinel is purely functional, functions that do not take parameters must return constants while performing no other side effects, making such parameter-less functions practically worthless.

There are multiple reasons for this limitation. The first is in the VPL. When the user attempts to create a function, the VPL first looks for possible parameters to the created function. Unconnected inputs are treated as parameters, without exception. In the case of anonymous functions, such unconnected inputs are not parameters to the outer function, but are instead parameters to an internal anonymous function. An example of this is shown in Figure 20. For this function, it deems that the parameters to Int< and GCD are required. However, there are two ways to read the first input to the GCD function, using LISP-style syntax:

1. (if (< {missing}) less more)
2. (lambda (x) (if (< x) less more))

In the first case, which the VPL assumes, there is a missing parameter, so any function made from this must include this parameter. In the second case, the missing parameter is
actually a parameter to an internal anonymous function. If GCD treats its first parameter as case #2 and it passes the anonymous function a parameter, then this is perfectly acceptable. The outer function does not take the parameter in case #2.

The way to solve this would be to make the VPL require more user input. It is believed that case #1 would be the common case, and so it would be safe to assume this. However, there must be some mechanism for the user to bypass this, telling Sentinel that case #2 is correct. This would mean a much more complicated VPL, as there is no obvious way to determine where exactly the anonymous function is. For instance, in Figure 20, either the first parameter to If could be treated as an anonymous function, or alternatively the first parameter to GCD. It is even possible to treat both as anonymous functions. More information about this can be seen in “Typing Weaknesses”.

Outside of the VPL, there are multiple problems that complicate parameterized anonymous function implementation. Internally, function parameters are all named, so chosen as
it forces the user to apply semantic value to parameters. As a result of this behavior, the order in which parameters are passed is not actually relevant for execution. Although there is a concept of a parameter order in the VPL, this is purely for aesthetic reasons. The difference between passing parameters with names and by order is shown below with LISP-style syntax in Figure 21. The value of passing named parameters instead of ordered parameters is shown in Figure 22.

(order-matters first-param
  second-param
  third-param)
(names-matter :second second-param
  :first first-param
  :third third-param)

Figure 21: Two LISP function calls illustrating the differences between using order and names to pass parameters. In 'order-matters', parameters are passed in a specific order, and the programmer must know what the correct order is. In 'names-matter', the order in which parameters is specified is irrelevant. The programmer must know the names of parameters instead.

(make-square-order 4 5 10 12)
(make-square-named :start-x-corner 4
  :start-y-corner 5
  :width 10
  :height 12)

Figure 22: Two LISP function calls showing the semantic value of using parameter names instead of order to pass parameters. In both cases, one makes a square of a given width and height, and places it on a Cartesian coordinate plane using the top left corner as a reference. In ‘make-square-order’, it is not obvious based on this call alone as to which parameters have which meanings. In ‘make-square-named’, this information is far more clear to the programmer, at the cost of extra typing.

If arguments were allowed in anonymous functions, it would lead to potential ambiguities. Names could not be used with arguments, as these could be different for arbitrary functions. Forcing anonymous functions to have consistent names also does not really solve the
problem. In this case, calling code would have to differentiate between anonymous functions and named functions, which is undesirable.

There is a third option to solve this problem: reflection. Although Sentinel is not an object-oriented language, it does support certain forms of reflection, which is typically considered an object-oriented trait (Tucker et al. 2007). With reflection, it would be possible to determine names at runtime, so parameters could be bound to arbitrary names. In such a design, parameter order would become the mechanism through which parameters are passed. Although this defeats the purpose of having named parameters, it would seem a possible solution.

However, even this would not permit anonymous functions with parameters for technical reasons. Such behavior would restrict the usage of arrays as parameters for anonymous functions. For example, Int+, which adds a series of integers, uses an array for its input. Arrays are implemented by specifying the same parameter name multiple times with different parameters, as shown in Figure 23. When named parameters are used, it is unambiguous how to treat these parameters. However, when order is used, certain ambiguities can arise, as shown in Figure 24.

```
(array-call :not-an-array 50
            :alpha-array first
            :beta-array second
            :still-not-array 10
            :alpha-array third)
```

*Figure 23: A LISP-style function call illustrating the usage of arrays. ‘array-call’ takes two non-array parameters, ‘not-an-array’ and ‘still-not-array’. It also takes two array parameters, ‘alpha-array’ and ‘beta-array’. For this call, ‘alpha-array’ holds ‘first’ and ‘third’, in that order. ‘beta-array’ holds only ‘second’.*
(dual-array 5 10 15 20)

*Figure 24: A LISP-style function call showing the problem of transforming named parameters to ordered parameters in the presence of arrays. `dual-array` takes two integer arrays as parameters. With the given arguments, it is impossible to determine which arguments belong with which arrays.*

In summation, there are three reasons why anonymous functions cannot take parameters in Sentinel:

1. The VPL is not intelligent enough to recognize parameterized anonymous functions.
2. All parameters are named and unordered, but anonymous functions lend themselves to ordered parameters.
3. In the general case, arrays cannot be unambiguously interpreted based only on order.

There is a second caveat to Sentinel being a functional language, this time in reference to it being a “pure” language. The core is pure, and does not utilize side effects in any way. As such, any functions written using only the core will be pure by extension. However, Sentinel functions can also be written in another language, one that does allow side effects. It is strongly encouraged that any such functions will be free of side effects. Side effects are usually difficult enough to manage, but in Sentinel they would be a nightmare. Sentinel is designed with the assumption that user code is pure. This can be seen in the workflow-like design, where inputs and outputs are chained to get a result. Workflows can break down when arbitrary side effects occur outside of the workflow. What may have worked fine a moment ago could fail now on the same input, which is nonintuitive with the workflow model.

From a more concrete perspective, debugging would be extraordinarily hard if Sentinel functions had side effects. All functions in Sentinel are static; there is no object-oriented methodology that can encapsulate side effects. This could lead to some serious problems.
Consider a very simple function, an incrementer. The incrementer starts at 1. Each time it is called, it returns the number it is holding and increments the number. This sounds simple enough.

Consider a practical usage of the incrementer: adding line numbers to rows. If Sentinel is run non-interactively, the incrementer will be called once per row, adding a number. However, one must be careful to structure the code so that the incrementer’s code is called last. If not, then line numbers will be added too quickly. If any rows are marked as erroneous after the incrementer was called, then erroneous rows will take whatever numbers there were. Consider an example with five rows, of which 2 and 4 are erroneous. If the incrementer is called before the rows are discarded, the final clean rows will have line numbers 1, 3, and 5, which is not correct.

Of course, with side effects, order of code execution is vital, so one may argue that this is simply a necessary evil of side effects. This may very well be true. However, consider if the user is instead running an interactive Sentinel session. If the user opens and tests a function that uses the incrementer in the VPL, then the line numbers will be thrown off. For however many tests the user ran with the incrementer, the line count will be off by that number. This will result in a gap of line numbers that have been issued, i.e. 1, 2, 3, 7, 8, 9, etc. Simply telling the user to “be careful” when debugging is not sufficient. Any number of things can force reevaluation of the function being tested. In fact, any manipulation of nodes that are in any way connected to code holding the incrementer can cause reevaluation, inappropriately triggering the incrementer. As such, “being careful” can translate to “never debug”, which is unacceptable.
Typing Weaknesses

As stated before, Sentinel does have a few solid types which can be used for rudimentary error-checking purposes. However, there are a number of weaknesses with the typing system. For one, there is no way to create a user-defined type in Sentinel. All of the types are hardcoded in at the lowest level in the interpreter, and this cannot be changed without extensive modification of the interpreter. This is unfortunate, especially given that error correction problems can be easily looked at typing problems. For example, one could look at a mitochondrial locus as a certain type, and merely verify that values are of that type. While this may not be obvious for someone without programming experience, it is assumed that programmers familiar with statically typed languages would be able to quickly see this.

In addition to being overwhelmingly prohibitive to extend, the typing system is underutilized. Although there can be enough typing information provided to rival a statically-typed language, there are no optimizations performed that take advantage of this. The only “optimization” performed is that of checking if expected types match found types. If they match, then no attempt is made at a type conversion. If not, then an attempt is made to convert the given value to a value of the expected type.

Within the VPL, typing information can be used to assist the user in making connections. To some degree, this is already done. Consider the following example, illustrated in Figure 25. “String” is of type String, but Int!= takes only Integers. “String” cannot be converted to a String, and upon trying to connect “String” to Int!=, the user is presented with an typing error message. Everything is disconnected from Int!= in this case, as this is the function that triggered the error.
Although it appears that type information is being used statically in this example, this is not true. In reality, the VPL executes code as it is written to verify types are correct. Dynamic typing is still the only kind of typing used. There is no “compile time”, nor is there really a true “runtime”; as far as Sentinel is concerned, it is always runtime, and it always attempts to run executable code.

There are drawbacks to this kind of behavior. For one, it is woefully inefficient. Consider a function that runs an infinite loop. In a statically typed language, this could never complete, though it is possible to validate its typing information. In Sentinel, this would run forever upon trying to validate typing information. This may be true of a typical dynamically typed language anyway, but there is a twist in Sentinel. In Sentinel, it is not actually possible to write something that would cause an infinite loop, as the interpreter would enter an infinite loop upon completion! The function may be valid code, but it could never be saved to a file. At time of writing, it is not possible to disable this behavior, although it would be trivial to implement.

Another problem with this behavior is that it cannot catch everything. Consider the example shown in Figure 26. Int!= takes Integers, but Int< is a Matcher. Matchers cannot be converted to Integers, so this will always fail. However, it is possible to make this function in

Figure 25: Example of a function in the VPL that can utilize typing information. “String” cannot be connected to Int!=, as Int!= takes only Integers and “String” cannot be converted to an Integer.
the VPL. The reason is that the VPL cannot execute this as it stands, since Int< is missing a required parameter. If it cannot execute it, then there will be no attempt to make a type conversion, and so the typing error is not caught at this stage.

![Int<](image)

*Figure 26: Example of a typing error that the VPL cannot catch. Unless parameters are passed to Int<, this code cannot be executed. Until the code is executed, typing errors cannot possibly be found.*

Given that there is static typing information provided, it may seem strange that it is not utilized until runtime. There are two reasons for this behavior. The first of these is noted in Table 5. Some conversions are context-sensitive. Such conversions may or may not be possible, depending on the value in question. For example, the String “123” can be converted to an Integer, but the String “String” cannot be converted to an Integer. In these cases, static typing information can only be used to show that a conversion is possible. The code must actually be run to see if the conversion succeeds or not.

Even so, the correctness of some conversions can be determined statically. However, this static typing information is not utilized. There is a specific reason for this behavior, rooted at the heart of the interpreter. Table 1 in the results section is not entirely correct. It should read that this is the default behavior, which is permitted to be different between functions. Functions written in a non-Sentinel language can actually override the default conversion routines, allowing for conversions between arbitrary types. Such conversions are permitted to have
arbitrary behavior, even to the point where nothing about typing can be determined statically. This was an intentional design decision to allow for flexibility. In retrospect, this may be too flexible. Overuse of this mechanism could lead to inconsistent, confusing behavior. None of the functions in the core utilize this feature.

**Turing Completeness Consequences**

It has already been mentioned in the results section that Error Sentinel is not a Turing complete language. According to Kiper et al. (1997), this is a severe weakness, as the authors assert that Turing completeness is a necessary property of VPLs. However, the validity of this assertion is called into question. Error Sentinel is not intended to be a general purpose programming language, but rather a language specifically for error correction and curation. According to Nardi (1993; as cited by Grinter 1994), specialized languages should only as complicated as necessary to model their given domain. Given the successful application of Error Sentinel to mtHaplogroups, the Error Sentinel language is clearly powerful enough to achieve its goal as a curation language.

Although Error Sentinel is not currently Turing complete, it is entirely possible to extend it to be so. With very little effort, special matchers and replacers could be written in a language external to Error Sentinel. If these imparted support for arbitrary iteration over spreadsheets, Error Sentinel would quickly become Turing complete. However, such functions have been intentionally unwritten. Such would operate external to Error Sentinel’s typical control flow, and could thus become points of confusion. At this moment in time, such functions seem completely unnecessary for curation. As such, in accordance with Nardi (1993; as cited by Grinter 1994), such functions are better left unwritten.
**Performance and Efficiency**

The benchmarking results shown in Figure 17 are both revealing and somewhat surprising. Error Sentinel outperformed ErrorChecker.pl both with and without the database-related checks by a significant margin. This is surprising since ErrorChecker.pl is far simpler than Error Sentinel.

The other benchmarks are not consistent with this finding. With the simple ruleset consisting of only a single good data and error correction rule in Figure 18, Error Sentinel is by far the slowest error correction program. Figure 19 is even worse by comparison.

With these benchmarking results in hand, along with the results from profiling in Tables 7 and 8, it is possible to narrow down where inefficiencies are in both programs. ErrorChecker.pl will first be looked at. The “rowMatches” function takes the most time overall. This function is used to determine if a given matcher matches on given input data. Considering the nature of this function, its position as the most time consuming function is unsurprising.

The next function listed, namely “concatArgumentValues”, is far more interesting. This function is used by all replacers used in the ruleset to output corrected data. It simply concatenates all the parameters that were given to the replacer. For the particular ruleset used, all replacers have only a single parameter. As such, this ends up returning only a single value in all cases, without any actual concatenation. Given the seeming simplicity of this operation, it may seem odd that it is so time consuming overall. In this case, Devel::NYTProf provides additional useful information, profiling down to the line. Two lines in particular seem to take a particularly long amount of time, each accounting for roughly 3% of the total runtime.
The first of these lines performs the initial setup for a simple “for” loop. This loop always ends up executing exactly once. The relevant code for this setup is presented below:

```
for( my $x = $start; $x <= $end; $x++ ) {...}
```

It cannot be stressed enough that it is not the body of the loop that takes 3% of the total runtime (shown as ...), but rather the shown portion itself. The second time consuming line checks whether or not the return value of the function is an empty string. This is presented below:

```
return ( ( $retval eq "" ) ? undef : $retval );
```

These two lines are required for correctness, and so this is a dead end as far as code optimization is concerned.

“concatArgumentValues” calls one other routine as it executes, though this routine does not contribute to “concatArgumentValues”’s sizable runtime. (As noted in the results, reported values do not account for any subroutine calls made by the given subroutine.) This called routine is the anonymous function listed as “__ANON__[ArgumentTaker.pm:62]”. This anonymous function in turn calls another function, namely “getArgumentValue”. “getArgumentValue” accounts for 2.4% of the runtime, and is ranked 11th, just beyond the cutoff of 10 for being listed in Table 7. Note that the anonymous function only wraps another function around “getArgumentValue”; the anonymous function performs no explicit work other than calling “getArgumentValue” and returning its result. Being that the actual call to “getArgumentValue” is not included, it seems that a massive inefficiency has been spotted. The mere creation and execution of this anonymous function consumes 4.6% of the total runtime, and the function does no real “work”.

This problem was explored further. It was thought that this was perhaps an inefficiency in Perl’s anonymous function generation facilities. As such, a named function was created that performed the same call to “getArgumentValue” as performed by the anonymous function. In the same context, the named function was referenced with a function pointer. Benchmarking was then performed on this modified ErrorChecker.pl, taking the minimum of five runs as with the rest of the benchmarks. Unfortunately, the runtime did not improve. In fact, the runtime actually increased by 0.04 seconds, but this change was considered so insignificant as to not be relevant. Profiling was performed again, as well. The named function that was used ended up being ranked in the same position as was the anonymous function, though calls to it accounted to 4.5% of the runtime instead of 4.6%. This difference is so small as to likely be due to chance.

This bit of code was again modified. For both the named function and the anonymous function, indirect referencing of a function was still used to call the function. In other words, instead of calling the function “myFunction” like so:

myFunction( $parameter );

...it was called like the following:

$myVar = \&myFunction;
$myVar->( $parameter );

The second method of calling “myFunction” comes with a performance penalty, as it requires that “$myVar” be dereferenced. As such, this bit of code was modified to call the function directly. Benchmarking was performed again on this modified code. The resulting minimal runtime of five runs was 6.73 seconds, which seems significantly different from the original 6.9 seconds.
For these reasons, it appears that dereferencing function pointers in Perl is not very efficient. Considering the relatively small amount of code that needed to be changed, the performance increase is substantial. Unfortunately, this modification would break the correctness of the code if database matchers had been used, for reasons beyond this discussion. As such, such a modification cannot remain.

This dereferencing inefficiency alone could account for the poor performance of ErrorChecker.pl in Figures 14 and 17. Perl’s object-oriented programming facilities are used extensively in ErrorChecker.pl. Such objects hold a number of methods (the special name for a function belonging to an object), which are accessed by dereferencing function pointers. Avoiding dereferencing function pointers thus means avoiding using objects in Perl altogether, which would require a total overhaul of the current ErrorChecker.pl.

Beyond the dereferencing efficiency problem, there appear to be other inefficiencies in Perl revealed by a look at line-by-line profiling information. An upsetting trend, already hinted at, is that function calls are generally expensive. For these calls, one must first unpack any arguments passed. At the end of the function, one must return a value, assuming there is a value to return. Although these two operations are seemingly simple, they comparatively take an enormous amount of time. For example, the “getDesiredColumns” function spends more time performing these two operations than it does doing actual work. In fact, for the ten functions listed in Table 7, these two operations account for a full 32.4% of the total runtime. This does not even take into account the other 579 functions that were called during the execution of ErrorChecker.pl.
It may seem strange that so much analysis has been performed on ErrorChecker.pl, given that it is not part of this work. However, this is needed to contextualize the comparisons made in Figures 14 and 17. Given the seeming novelty of ErrorChecker.pl and Error Sentinel, it is not known what else could be used for comparison. A single use program was not written that implements the full error correction ruleset for precisely the reason why Error Sentinel was developed: such would be large, complex, and essentially useless after its development. If one estimates that each of the 99 rules requires between 10 and 20 lines of another language to implement, then the resulting program would be approximately between 1,000 and 2,000 lines. For comparison to these figures, the entirety of ErrorChecker.pl is just under 2,200 lines. Quite simply, it is not feasible to develop such a program strictly for benchmarking purposes.

Such a single use program would help immensely in trying to gauge the performance of ErrorChecker.pl and Error Sentinel, especially if it were written in either Scala or Perl. The previous discussion of ErrorChecker.pl unfortunately reveals that the comparison shown in Figures 14 and 17 is not direct, simply because different languages were used to implement the programs. In contrast to Perl’s inefficient function dereferencing and object-oriented facilities, Scala is purely object-oriented. This means that almost everything tangible in the language is either an object or a method being called on an object. For example, even the basic expression “1 + 2” is seen as calling the “+” method on a “1” object with a “2” object as a parameter. Given Figures 14 and 17, this almost certainly is implemented in a more efficient manner in Scala than in Perl. Additionally, since all actions that can be performed in Scala require function calls, function calls themselves must be more efficient than they are in Perl. For these reasons, it seems that the comparisons in Figures 14 and 17 more accurately reflect efficiency differences.
between Perl and Scala, as opposed to efficiency differences between ErrorChecker.pl and Error Sentinel.

Indeed, the rest of the benchmarking results seen in Figures 15, 16, 18, and 19 are consistent with this hypothesis. In ErrorChecker.pl, ruleset elements are implemented as objects. To be specific, matchers, replacers, parameters to matchers and replacers, rules, and entire rulesets are all implemented as objects. Altogether, for the larger ruleset used in Figures 14 and 17, this means that ErrorChecker.pl will make a multitude of objects, which, in turn, result in a multitude of function pointer dereferences.

The following attempts to model the creation and use use of these objects. For each good data rule, at least two objects are made: one for the matcher, and another containing both the matcher and the rule itself. Additionally, one object is created for each matcher parameter. Matchers can have an arbitrary number of parameters, but for the purposes of this model it is assumed that each matcher takes a single parameter. Not only does this simplify the model, it holds true for the real error correction ruleset used. Altogether, for this model, this means there are three objects that are created per good data rule.

In determining if a row matches a given good data rule, this means that these objects must interact in a certain way. In the current design, the parameter objects are dereferenced twice: once in a preparatory step and again for retrieving the actual value. The matcher is itself dereferenced in the process. Altogether, this means that there are at least three function dereferences per row.

For each error correction rule, at least four objects are created: a matcher, a replacer, an object representing the rule for the matcher, and an object representing the error correction rule
as a whole. Additionally, replacers can take an arbitrary number of parameters; again, one is used for the same reasons. So for the purposes of this model, each replacement rule requires 6 objects, including a parameter for both the matcher and the replacer.

The evaluation of error correction rules results in even more function dereferencing. The object representing the matcher rule as a whole must be dereferenced, which, in turn, performs three additional dereferences. If the rule matches, then the parameters for the replacer must be dereferenced for preparation, followed by the replacer itself. This means an additional two dereferences. The replacer must dereference its argument during processing. In the error correction data set used in Figures 14 and 17, there is at least one more dereference to keep track of, namely the aforementioned concatenation of arguments. Altogether, a replacement rule needs 4 dereferences if it does not match, and 8 if it does.

Beyond the individual rules, entire rulesets are implemented as objects. Each column can have an arbitrary number of rulesets associated with it. In this model, it will be assumed that there is only one ruleset per column, which correlates to the actual error correction ruleset used. This one ruleset requires an additional dereference.

With all of this information in mind, consider a column with N good data rules and M replacement rules. This will be represented using $3N + 6M + 1$ objects, including the object for the column as a whole. To make things more concrete, Table 9 shows how many good data rules and error correction rules are used per column for the full error correction ruleset utilized in Figures 14 and 17.
Table 9: The number of good data rules and replacement rules used along with the number of objects created for each column for the real error correction ruleset utilized in Figures 14 and 17.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Number of Good Data Rules (N)</th>
<th>Number of Error Correction Rules (M)</th>
<th>Total Number of Objects Made (3N + 6M + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haplogroup</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
<tr>
<td>Subhaplogroup</td>
<td>1</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Basepair</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Reference Allele</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Variant Allele</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Locus</td>
<td>2</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>Mutation</td>
<td>4</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Citations</td>
<td>1</td>
<td>73</td>
<td>442</td>
</tr>
<tr>
<td>Monomorphic/Polymorphic</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
<tr>
<td>Comments</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
</tbody>
</table>

As shown in Table 9, altogether 532 objects are created for the real error correction ruleset used in Figures 14 and 17. However, the problem lies not in the number of objects created, but rather in how they interact with each other. This can also be modeled somewhat generally. Good data rules will be considered first. In the best case, a row is declared invalid by the first listed good data rule. In this case, 4 function dereferences are required, including one for the good data rule as a whole. In the worst case, a row passes all the good data rules except for the last one. In this case, all good data rules were dereferenced, resulting in 4N dereferences.
When a row enters the error correction rules, the best case is when it is corrected by the first rule. This means 9 function dereferences, including one for the replacement rule as a whole. In the worst case, it matches only the last listed error correction rule. In this case, $4(M - 1) + 9$ function dereferences are needed, since $M - 1$ matchers did not match, and the last matcher did, resulting in a replacement. Note that this simplifies to $4(M + 1) + 1$.

In addition to those dereferences, one function dereference is required per ruleset per column. Altogether, this means the best case number of dereferences is 4: three for the good data rule and one for the column rule. The overall worst case is $4N + 4(M + 1) + 2$, since all good data matchers were executed and all error correction rules were evaluated. This includes the dereference needed to access the column ruleset as a whole.

At this point, there is enough information to determine exactly how many function dereferences are needed per row per column for ErrorChecker.pl. This information is presented below in Table 10. Note that since the best case behavior is always four dereferences, they have not been included in the table. The exceptions to this rule are the Haplogroup, Monomorphic/Polymorphic, and Comments columns, which have no associated rulesets, and so they require no function pointer dereferences.
Table 10: The worst case number of function dereferences needed per row per column on the mtHaplogroups data set for ErrorChecker.pl utilizing the full error correction ruleset without database checks used in Figures 14 and 17.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Number of Good Data Rules (N)</th>
<th>Number of Error Correction Rules (M)</th>
<th>Number of Worst Case Function Dereferences Per Row (4N + 4(M + 1) + 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haplogroup</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
<tr>
<td>Subhaplogroup</td>
<td>1</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Basepair</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Reference Allele</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Variant Allele</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Locus</td>
<td>2</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>Mutation</td>
<td>4</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>Citations</td>
<td>1</td>
<td>73</td>
<td>302</td>
</tr>
<tr>
<td>Monomorphic/Polymorphic</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
<tr>
<td>Comments</td>
<td>0</td>
<td>0</td>
<td>0 (no associated ruleset)</td>
</tr>
</tbody>
</table>

Altogether, this model predicts that in the absolute worst case, ErrorChecker.pl requires 414 function dereferences per row of the mtHaplogroups data set. Given that the full data set contains 4,348 rows, this means there is a worst case need to dereference an astounding 1,800,072 function pointers.

As amazing as it sounds, this is a theoretically conservative estimate. The actual design of ErrorChecker.pl has been simplified in this discussion. With the profiling data in hand, it is
actually possible to determine the exact number of function pointer dereferences that were needed. The actual number of function pointer dereferences needed for the entire mtHaplogroups data set using the real error correction ruleset without database-related checks is 1,645,646. Given the aforementioned inefficiency of function pointer dereferencing, this seems to be a serious problem.

There is enough data at this point to estimate how long ErrorChecker.pl spent dereferencing function pointers. Based on the benchmarking results after the function pointer was replaced with a direct function call, 2.46% of the runtime was previously spent dereferencing function pointers ( \( \frac{6.9 - 6.73}{6.9} = 0.0246 \) ). This function was called 174,846 times. With this in mind, a proportion can be set up, with the number of function pointer dereferences over the total percentage of runtime. Stated another way: \( \frac{174,846}{2.46} = \frac{1,645,646}{X} \), where X is the total amount of time spent dereferencing function pointers. This results in an estimated value of 23.2% of total runtime spent dereferencing function pointers. This value seems to be extremely high.

At this point, the seemingly strange results presented in Figures 14-19 should be more clear. For the small ruleset, where ErrorChecker.pl far surpassed Error Sentinel in performance, far fewer function pointer dereferences were needed. Using the above model as a guide, only 4 function pointer dereferences are needed for clean rows, and 14 function pointer dereferences are needed for erroneous rows. Only 197 rows need the replacement in this smaller data set. In total, this means that only \( 197 \times 14 + (4348 - 197) \times 4 = 7,362 \) function pointer dereferences are needed for the entire sheet, at least according to the model.
To put it in perspective as to how small this number really is, assume worst case behavior for all rows with the data presented in Table 10. With this assumption, by the time the 18th row has been processed using the full error correction ruleset without database checks, more function pointer dereferences have occurred than in processing the entire spreadsheet with the dash replacing ruleset. Looking at actual profiling data of ErrorChecker.pl with the dash replacing ruleset, 41,942 function pointers were dereferenced in the code. Though this is significantly more than the number predicted by the model, this number is only about 2.5% the amount needed when the full ruleset without database checks was used. Indeed, under these circumstances, the profiling data reveals that merely reading in and parsing the spreadsheet data becomes the bottleneck, as opposed to the execution of the error correction ruleset.

The pointer dereferencing efficiency problem alone does not explain all the benchmarking results presented in Figures 14 -19. The profiling information for Error Sentinel presented in Table 8 reveals some striking inefficiencies. However, caution must be taken when analyzing this data. Given the large percentage of the total runtime consumed by “java.lang.Object.wait” and “java.lang.ref.ReferenceQueue.remove”, these appear to be good candidates for optimization. However, these two functions appear to be artifacts from the profiler itself. They do not originate in any code that was either directly or indirectly called by Error Sentinel. As such, they will be ignored.

The next function listed in Table 8, namely “scala.StringBuilder.append”, is not called directly by Error Sentinel. This is a Scala library routine called indirectly. The call originated in a routine that is used to parse individual tokens (cells) in comma-separated value files. This routine is called for every unquoted token in comma-separated value files. This applies to the
majority of the cells in the mtHaplogroups data set, which contains a total of 43,480 cells.

Clearly, this routine is called frequently, and so efficiency is crucial for it.

Based on these results, this routine appears to be the biggest bottleneck in the entire system. Another function in this list is directly related to this routine, namely “

sentinel.io.input.csv.ParseCSV$$anonfun$parseUnquotedToken$1.apply”. Given the large portion of the overall runtime consumed by these two routines, a second look was performed at the routine used to parse unquoted tokens. The routine in question was found to be extremely inefficient. For a token of length N, it required that N + 1 objects be created in order to parse it. Worse yet, it ran in O(N^2) time as opposed to the theoretical minimum of O(N).

As such, both this routine and its counterpart that parsed quoted tokens were modified. The modification resulted in a function that would create exactly 2 objects per token parsed, and run in O(N). This modified version of Error Sentinel has shown approximately a 10% performance improvement with the dash replacement ruleset, and a 5.6% performance improvement with the real ruleset without database checks. However, its performance when applied to only a single line of the mtHaplogroups data set has not improved. Note that this is not the version that was benchmarked in Figures 14 -19.

The next two routines in Table 8, both with the name “scala.collection.mutable.FlatHashTable$class.addEntry”, are Scala library routines indirectly called while loading spreadsheets. Specifically, they are used in the association of good data rules and error correction rules with spreadsheets. Currently, this association process is extremely naive. For every spreadsheet made, another hidden spreadsheet is made that holds good data rules and error correction rules. Each rule is then loaded into this hidden sheet, one at
a time. If there are three rules that are intended to occupy every cell of a spreadsheet, then the corresponding hidden spreadsheet will be iterated over in its entirety three times, once for each rule. This is extremely inefficient both in terms of CPU and memory. A reference to the rule is held by each cell of the hidden spreadsheet, which in the case of the mtHaplogroups data set means the creation of over 100,000 objects just for this simple example!

This is a case where the flexibility provided by Error Sentinel is also a weakness. ErrorChecker.pl only allows rule granularity to be down to the column. In ErrorChecker.pl, a rule can apply to the whole sheet or a whole column. Any other combinations are invalid. In Error Sentinel, a rule can apply to a single cell, a single row of cells, a single column of cells, to a single spreadsheet, or any combination of the above. This extra flexibility means that association of error correction language to cells is more difficult to do efficiently. There is likely an efficient way to solve this problem, though it seems non-obvious at this point in time. As such, no attempts were made to make this process more efficient.

The next line of interest in Table 8 is “scala.RandomAccessSeq$$anon$13.next”. This is a Scala library routine indirectly called by Error Sentinel in the writing out of spreadsheet data. Specifically, it does the opposite of the aforementioned token parsing routines: it creates tokens and writes them out. It is called even more frequently than the token parsing routines. The non-interactive version of Error Sentinel produces three output sheets, one for good data, one for erroneous data, and a third for explanations why data is erroneous. As such, for each input cell, this routine is called three times. Altogether, on the full mtHaplogroups data set, it is called well over 100,000 times.
This routine was found to contain several inefficiencies. Like its parsing counterparts, an excessive number of objects were made, though the number was unrelated to the length of the input. It also runs in $O(N)$, the theoretical best possible, though it iterates through the input unnecessarily more than once. As such, it has a constant factor that could be reduced. The routine was modified to create only as many objects as absolutely necessary, and to limit the number of iterations. The resulting modified version of Error Sentinel was benchmarked again. It has been shown to perform 3% faster on the real ruleset without database checks, and 5.6% faster on the dash replacement ruleset. Again, this is not the version of Error Sentinel that was benchmarked in Figures 14 - 19.

The next line of interest in Table 8 is that of “scala.runtime.RichString.<init>”. This refers to the creation of a class automatically by Scala. Specifically, it occurs during the association of the error correction language to spreadsheets. In the particular context, an implicit type conversion from Java’s “String” to Scala’s “RichString” was forced. The portion was rewritten as to avoid the need for a conversion. Benchmarks of this modified version of Error Sentinel showed only marginal improvement, with a 1.5% performance boost for the full error correction ruleset without database checks, and no improvement for the dash replacing ruleset. Given that this performance inefficiency occurs during the loading of error correction rules, and that the dash replacement ruleset contains only two rules overall, it is unsurprising that there is no improvement for the dash replacement ruleset.

The next two lines in Table 8, namely “scala.collection.mutable.ListBuffer.$plus$eq” and “scala.collection.mutable.FlatHashTable$class.containsEntry”, are both Scala library routines that are indirectly called by Error Sentinel. Once again, these are linked to the
association of the error correction language with spreadsheets. However, ultimately these were linked to a particular data structure that is used during the association. This data structure is somewhere between a list and a set. It can determine if an item is contained within in constant time, and it is possible to add items to it in constant time. It also retains the order of added items, unlike the typical set with constant time operations. Internally, the data structure uses both a list and a set.

The decision was made to rework the data structure in an attempt to improve performance. This ultimately meant choosing the best underlying list and set implementations to use internally. Between Java and Scala, there are several choices for both sets and lists, and the best combination was experimentally determined. The resulting benchmarks with the modified version of Error Sentinel are quite interesting. With the real error correction ruleset without database checks, performance improved by an impressive 7.5%. However, performance with the dash replacement ruleset actually degraded by 2.9%. Given that the larger ruleset is expected to be more realistic, it has been decided that this optimization should be kept in Error Sentinel. Again, note that the benchmarking information shown in Figures 14 -19 is not of this modified version of Error Sentinel.

With all of these modifications, the performance of Error Sentinel was improved by a substantial amount. The final version with including all the aforementioned modifications was benchmarked using all the aforementioned data sets and rulesets. The performance improvements are shown below in Table 11.
Table 11: The benchmarking results of Error Sentinel on various data sets and rulesets before and after performance optimizations were performed.

<table>
<thead>
<tr>
<th>Data Set And Ruleset</th>
<th>Runtime Before Modifications (Seconds)</th>
<th>Runtime After Modifications (Seconds)</th>
<th>Percent Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full mtHaplogroups with real ruleset with database checks</td>
<td>14.4</td>
<td>13.73</td>
<td>4.7</td>
</tr>
<tr>
<td>Full mtHaplogroups with real ruleset without database checks</td>
<td>3.54</td>
<td>2.94</td>
<td>16.9</td>
</tr>
<tr>
<td>Full mtHaplogroups with dash replacement</td>
<td>1.99</td>
<td>1.74</td>
<td>13.6</td>
</tr>
<tr>
<td>Single line of mtHaplogroups with dash replacement</td>
<td>0.9</td>
<td>0.88</td>
<td>2.2</td>
</tr>
</tbody>
</table>

As far as optimizing Error Sentinel goes, there are still a number of things that could be done. In general, Error Sentinel seems to perform poorly with small data sets and rulesets. There are several reasons for the poor performance. Based on the results for hardcoded Scala and Java in Figures 16 and 19, this performance loss begins even before any portion of Error Sentinel runs. Scala and Java code do not run directly on the hardware, but rather on the Java Virtual Machine (JVM). The JVM, in turn, runs on the hardware. It takes time to start the JVM, and the results for Java in Figures 16 and 19 best shows how much time this takes. Additionally, it would appear that there is a certain additional penalty for starting Scala code, beyond starting the JVM. Further exploration of this shows why: the command used to run start Scala code is
actually a shell script that ultimately starts the JVM. This script, though short, runs several external commands before starting the JVM, which could incur this additional performance hit.

The aforementioned performance hit of starting Scala is only partially to blame, however. Once Error Sentinel is started, it must load in the base language components. This must be done on each invocation; a performance sacrifice in exchange for the extensibility it offers. This loading process results in 76 Kb of XML spread across 10 files being read in and parsed, which takes time. Only after this information is parsed in can any other loading occur.

Once the base language components are loaded in, any additional language components that are specific to a project must be loaded in. For this particular benchmark, no additional components are used, so this step is skipped. The spreadsheet data must then be loaded in, corrected, and written out. Error Sentinel, does this in a three stage process, which is not very efficient. In the first stage, the spreadsheet is read in, which is disk intensive but leaves the CPU largely idle. In the second stage, actual error correction is performed, which has the opposite problem: the CPU is fully utilized while the disk is idle. In the third stage, the spreadsheet is written to disk, again utilizing the disk but not the CPU. A more efficient design would perform error corrections on data already loaded in concurrently with the reading in of new data and the writing out of new data. This design could theoretically achieve both maximum CPU and disk usage simultaneously, whereas this is impossible with the current three stage design.

That said, there is only a single row of data to read in for Figure 19, so this inefficiency is likely not a major factor in this case. Additionally, the other programs use an implicit three stage process. Each of the other programs in Figure 19, including ErrorChecker.pl, processes data according to the following pseudocode:
while there are rows to read in:
    inputRow = readRow( input )
    outputRow = processRow( inputRow )
    writeRow( outputRow, output )

Based only on the pseudocode, it should be clear that there is a three stage process for these programs. The difference is that there are three stages for each spreadsheet row, as opposed to three stages for each spreadsheet as a whole. Even if one assumes that there is buffering occurring behind this pseudocode, the basic principle does not change. If a buffer runs out of data, a read is forced. Similarly, if a buffer grows too big, a write will be forced. These reads and writes occur synchronously, which forces execution to stop. Similarly, while execution is occurring, the buffers remain static and the disk is idle. In other words, even with buffering, the disk and CPU are not utilized in parallel.

Given the benchmarking results in Figure 17, perhaps too much concern is being placed on the poor efficiency of Error Sentinel in the smaller benchmarks in Figures 18 and 19. On a real data set with a real, significantly large error correction language, Error Sentinel is able to outperform ErrorChecker.pl. Additionally, none of the profiling results in Table 8 point to inefficiencies in running the error correction language itself. Rather, Table 8 shows that all the current inefficiencies are in the initial loading of the language, along with the loading in and writing out of spreadsheets. This is in contrast to the profiling results of ErrorChecker.pl in Table 7, which are all related to inefficiencies in the actual error correction language.

Beyond the performance issues revealed through benchmarking and profiling, there are some other, more theoretical, performance issues that should be addressed. All these theoretical problems lie in the implementation of the error correction language itself. First and foremost, the language is interpreted as opposed to being compiled. Given the visual programming aspect of
it, this is essentially a requirement. When a user wants to try things out in the visual programming environment, the need to go through a constant code, compile, and test cycle would hinder progress. Such a compilation cycle could make the environment less responsive, which is undesirable to say the least.

Although Error Sentinel is interpreted, a number of measures have been taken to make this as efficient as possible. For example, once functions are initially loaded from files, the files never need to be consulted again for the duration of the program. These program files are parsed in and immediately converted to parse trees, with one parse tree per function. Each of these parse trees “looks” like code in the VPL, and they function in the same manner.

Note that these parse trees are never actually executed. Instead, these parse trees are used to build matchers and replacers on demand. This building process carefully constructs and connects the matchers and replacers to mirror the parse tree’s structure. The result is a composite matcher or replacer, constructed from other matchers and replacers. This is a very CPU efficient design, as illustrated through the example shown below in Figure 27.

Figure 27: Example used to show how Error Sentinel executes error correction rules. “Int+” is used to add an arbitrary number of integers together, and “Int-” is used to subtract an arbitrary number of integers form each other. This example shows the mathematical expression \((3 - 2) + 1\).
Consider the “Int+” replacer, which adds an arbitrary number of integers together. When this replacer is executed, it must first get the integer values of all its parameters. To do this, it calls a method on each input parameter which can get the parameter’s value as an integer. If the parameter is merely a constant, as with “1” in Figure 27, then the constant’s value is returned. However, the parameter could very well be another replacer, as with “Int-” in Figure 27. When “Int-” is told to return its value as an integer, it goes through the same process as “Int+”, ultimately returning 1 in this example. The important part of this is that the underlying parse tree is completely implicit. From the perspective of “Int+”, “Int-” could very well have been a constant. Additionally, the function pointer that must be dereferenced to get the value of “Int-” also serves to traverse this implicit parse tree. In other words, the necessary traversal of the parse tree is done “for free”, as it is combined with the need to determine the integer value of “Int-”.

There is a dramatically inefficient side effect to this design. The amount of memory needed to execute a ruleset is linear with respect to the length of the ruleset. Using the same replacer in N rules results in N nearly identical replacers being made, instead of a single replacer. For a more traditional language, this is analogous to a compiler implementing subroutines by copying the subroutine code to each place the subroutine is needed. With such an implementation, for a subroutine whose body consumes N bytes of memory, M calls to the subroutine would require in total MN bytes of memory. With such an implementation, an infinite number of calls require an infinite amount of memory just to store the code. With an implementation that holds only one copy of a subroutine in memory, the number of calls to the subroutine is irrelevant to how much memory the subroutine code consumes: it will always be N.
Although this is inefficient, it is not anticipated to be a major problem. As already mentioned, other portions of Error Sentinel make tens of thousands of objects, and a ruleset that contains this many rules seems impractically large, at least for use by people.

There is another important design decision influencing efficiency that should be mentioned. Although Error Sentinel supports dynamic typing, the underlying interpreter is able to avoid type conversions and even type checking in many situations. For example, in Figure 27, everything will be statically typed to integers. No type checking will ever be performed in this example, as it is unnecessary. Additionally, type conversions involving constants are performed only once. For example, if the integer constant “1” is passed to “Real+”, which adds real numbers together, the integer is converted to a real number only once. The converted real number is used for all subsequent operations, via a mechanism that makes it impossible to tell that a type conversion ever occurred. Through the same mechanism, Error Sentinel can sometimes determine if types will never be compatible, avoiding runtime execution at all. However, currently this portion is very limited, and applicable only to constant parameters.

In the current design, the majority of type conversions are related to reading in spreadsheet data. Since spreadsheet data can contain anything representable as a string, it is unsurprising that type checking and type conversions must be performed on this data. However, the incorporation of variables that require type conversions will not propagate type conversions elsewhere. For example, again consider Figure 27. The integer constant “3” could very well be replaced with a variable pointing to a cell in a spreadsheet. As a result, a type conversion must occur at runtime, in order to validate that the spreadsheet cell contains an integer. However, all other operations shown in Figure 27 are still statically typed to integers. This includes even the
output of “Int-”. This is possible since exceptions are thrown the moment type conversion fails. This immediately aborts execution, and so there is no danger of using an inappropriate value.

A final important note regarding efficiency is in regard to the functional paradigm. For certain operations, the functional paradigm has shown to be less efficient than the imperative paradigm, even at a purely theoretical level (Ben-Amram et al. 1992; Pippenger 1997). Additionally, the vast majority of the literature regarding efficient data structures assumes that an imperative language is used for implementation (Okasaki 1998). With such an assumption, it is impossible to implement these structures efficiently while retaining functional purity.

**Miscellaneous Design Flaws**

It is not possible to extend Error Sentinel with arbitrary types. For example, consider the locus column in the mtHaplogroups data set. Loci could very well be represented as types. Because they are currently not, there are problems if one desires to use loci as parameters. Whenever an attempt is made to use a locus, for the purposes of correctness, the user must explicitly check that the given parameter is a locus. As mentioned elsewhere, such checking is both cumbersome and error prone. It would be much better to automatically and instantaneously throw an exception when an attempt was made to throw an invalid locus, much like the rest of the types Error Sentinel supports.

For that matter, the current design of Error Sentinel would have to be overhauled for the specification of arbitrary types. Without getting too deep into the implementation, the possible types are hardcoded in a manner that is not at all extensible.
Miscellaneous Implementation Flaws

The class hierarchy for parameters is fairly complicated. For example, the class representing integer constants extends six other classes, not including three additional classes that all user-defined objects in Scala automatically extend. Additionally, this class hierarchy fully exploits some of Scala’s more advanced features, which are likely beyond the grasp of the typical Java programmer. To put it simply, this class hierarchy cannot be directly represented in Java.

There is currently no way to dynamically reference other cells in a spreadsheet. In other words, it is not possible to choose which cell a value should be taken from at runtime, though it is possible to do so at the time of coding. For example, in the mtHaplogroups data set, the value in the basepair column is needed in order to determine if the value in the locus column is valid. Currently, this is specified with an association in an XML file. In the IsLocus matcher itself, there is nothing that attempts to find the locus column.

It open to debate as to whether or not this is a bug or a feature. It would actually be fairly trivial to add a function to do this, at least with the current parameter design. However, such behavior is so flexible as to be potentially confusing. Additionally, the current interactive version of Error Sentinel would not be able to highlight such parameters. Worse yet, the interactive version of Error Sentinel could show misleading results for such functions. If the value of a dynamically specified spreadsheet cell changes, it currently has no way to know which cells rely on this value for error correction. Without this information, related cells cannot be reevaluated in response to a change. Selecting such cells will show stale error correction information.
Currently, the error correction language is capable of dealing with only a pre-specified finite number of input cells. To illustrate this problem, consider an arbitrary rule: the value of a cell C must exist in another column U. Although this sounds very simple, it is not possible to implement this rule in the current version of Error Sentinel. That said, it is possible to write a function in a non-Error Sentinel language that can do this, and extend Error Sentinel with this function. However, it would not be very efficient. For such a rule, it is theoretically possible to read the column only once, building an index. Subsequent invocations of the rule could consult the index, which is far more efficient than directly consulting the column. Currently, the language is too far removed from the spreadsheet to be able to build this index correctly. The index would have to be updated in response to the spreadsheet changing, but there is currently no way to send such a message.

Internally, some of the names chosen for classes are poor. For example, sets of functions are referred to as “languages”, when the word “language” can refer to many different things within the context of Error Sentinel.

A major flaw is that function names must be globally unique. Although the visual programming component shows functions as being members of different packages, this is strictly for visual presentation. All these names exist in the same namespace. For small rulesets that only utilize a few dozen custom functions, this is not a problem. However, for a project of an arbitrarily large size, this could be a nightmare. Sharing code would become problematic, as no two people could use the same function names. In such a case, the names of functions would have to be changed before use, which at the very least could result in confusion.
Many parts of the system are represented in XML. In XML, the order of tags is supposed to be irrelevant to program correctness. In cases where order does matter, supplementary information is supposed to be provided which can be used by a parser for correct ordering. However, there are certain XML files where order is vital and there is no such supplementary information. Most notably are files containing user defined functions in Error Sentinel. In cases where arrays are passed, the ordering of arrays is determined by the order of tags. Error Sentinel claims to guarantee that the order of arrays will be preserved, although this flaw in the XML means that such a guarantee should not be true. In this case, the problem is not that Error Sentinel is incorrect per se, but rather that it does not conform to the XML specification. For that matter, no formal schemas were defined for XML files. These files are fairly straightforward, but this does not excuse making a schema.

The implementation of the Error Sentinel language features a large number of built-in routines. In total, there are 90. Although the vast majority of these correlate to basic functions in the core of GCL, including routines for integer arithmetic, this still seems to be very high. According to Green et al. (1996), a multitude of such basic operations is a bad thing for visual programming languages, as such run contrary to the design goal of simplicity. New users must at least glance over these to be aware of what is already provided, which takes real time. This is especially troubling given that the majority of users of spreadsheet languages use fewer than ten built-in routines in practice (Nardi et al. 1990). Removing routines seems hardly the proper approach to fixing this problem, and only with real usage data can it be accurately determined as to what is or is not necessary.
The control flow for the non-interactive version of Error Sentinel is extremely simple. Cells are corrected one at a time, proceeding first across columns, then across rows. For example, in a 2x2 spreadsheet, it processes cells in the following order: (1,1), (1,2), (2,1), and (2,2). This is much like reading text in English.

This simplicity has a major drawback. In cases where cells are interdependent, as with the mtHaplogroups data set, it is not possible to guarantee that corrections will be performed in the proper order. If the correctness of cell X requires that the value of cell Y be known, then cell Y must have any corrections performed on it first. In other words, if other cells depend on a given cell, than the given cell must be corrected before the other cells.

The way to solve this problem is to form a dependency graph, such as the one shown in Figure 6. Columns are then checked in the order yielded by the graph. With figure 6, this would mean checking the basepair, then the reference sequence allele, then the variant allele, and finally the functional change. The locus can be checked after the basepair, but before the functional change. By ordering data this way, it can be guaranteed that any corrections that can be performed on related data have been performed.

This is not a problem with the mtHaplogroups data set, because it just so happens that the columns are already in the same order as a dependency graph would yield. As such, this problem does not affect the mtHaplogroups data set. However, this is due mostly to luck; such an assumption is not safe to make for an arbitrary data set.

The aforementioned solution of forming a dependency graph is not too difficult. Indeed, Error Sentinel already contains code that can generate such graphs, as such capability is needed.
to resolve language dependencies in the proper order. However, the code has not been applied to this problem.

There is another problem regarding interdependencies in an arbitrary data set. In order to resolve dependency graph, the graph must take the shape of a directed acyclic graph (DAG). The dependencies for mtHaplogroups shown in Figure 6 happen to form a DAG. However, this cannot be guaranteed for an arbitrary data set. It is completely possible to write circular rules: the value of X requires Y, and the value of Y requires X. For such a circular rule, the non-interactive version of Error Sentinel would have a major problem. It is possible to have an “infinite loop” in the ruleset; correcting X requires a correction in Y, which requires a correction in X, ad infinitum. The interactive version of Error Sentinel can handle this because the control flow is based solely on user activity. The user must manually go between Y and X each time this happens, and a human can very quickly see that an infinite loop is formed. However, for a computer, this is a far more difficult problem. Indeed, the problem of determining if code does or does not form an infinite loop is known as the halting problem, and the general form of the problem is mathematically impossible to solve (Turing 1936). As such, for circular rulesets, Error Sentinel will likely never be able to guarantee that corrections will be applied properly.

Current State

The language interpreter and non-interactive version of Error Sentinel are considered complete. Both have undergone extensive testing. The interactive version of Error Sentinel is usable, though incomplete. It currently lacks a GUI for associating language components with spreadsheet cells. Such associations are specified in XML, and so for all tests these were manually written in XML. It is intended that such a GUI will be available in the future.
The visual programming component of Error Sentinel is currently extremely rough. There are currently a number of minor bugs that collectively hamper usage. Functions themselves are represented as XML, so it is possible to make functions without the visual programming component. For all the mtHaplogroups curation functions made, XML was written by hand, instead of being generated by the visual programming component. The intention is to fix all these bugs in the future.

Altogether, as of this writing, Error Sentinel consists of just over 24,000 lines of well-commented Scala. Given the large size, there are almost certainly additional hidden bugs. That said, there are certain “hotspots” in the code that have been analyzed repeatedly, and are more likely to be correct than others as a result. For example, the file containing the bulk of the code for parameters has gone through no less than 28 separate revisions. Given the importance of such code, it should be apparent why the number is so high.

Applications

Error Sentinel’s potential applications are not limited to the mtHaplogroups data set. Theoretically, it can be applied to any data that can be represented as a series of one or more spreadsheets. Based on personal communication, there are people in the biotechnology, engineering, and information technology fields who all think they could use Error Sentinel, though such is anecdotal evidence.

Extensions and Future Work

Microsoft’s Excel allows for arbitrary plugins to be written. If Error Sentinel’s interactive version was implemented as such a plugin, it would be far more natural to the typical user. Integration with Microsoft Excel would mean a much richer set of tools would be available.
to the user without much additional work. This is in contrast to the current spartan interface of
the interactive version of Error Sentinel.

As mentioned in the implementation flaws, it would also help immensely to add support
for user-defined types. Considering the goal of a simple language that supports both strong and
dynamic typing, doing this properly would likely require some ingenuity. Not to mention that
the current parameter design would need to be completely overhauled.

At the time of writing, the Error Sentinel language works, but there are a number of
things that could be done better. As far as efficiency goes, a whole slew of optimizations could
be performed. Beyond CPU efficiency, memory efficiency optimizations are completely lacking.
For a “real” programming language, it would be unacceptable to implement subroutines in the
same wasteful way as Error Sentinel.

References

   Salas. “Reassessing the Role of Mitochondrial DNA Mutations in Autism Spectrum

   Howell. “Reanalysis and Revision of the Cambridge Reference Sequence for Human


Appendix

“Entire Full with Database” and “Entire Full without Database” Rule sets

The following tables detail the “real life” error correction ruleset used by ErrorChecker.pl in benchmarking. These illustrate a number of errors that have been found in the data, along with the ways they were corrected. Given that the Error Sentinel version of these rules is in a far bulkier XML format, and that such rules are equivalent, Error Sentinel rules have not been shown.

To understand these rules, understanding of ErrorChecker.pl’s matchers and replacers is required. In ErrorChecker.pl, items in quotes are considered to be constants, whereas unquoted items are treated as column names. Column names act as variables. For example, if ‘basepair’ is specified as a column name, then the value of the basepair column for the current row is used.

In addition to parameters, ErrorChecker.pl matchers and replacers have a “target”. The target is the value in a cell. This same design is used in Error Sentinel, though it is decoupled from the language.

ErrorChecker.pl has a feature lacking in Error Sentinel. Both error correction frameworks allow for multiple good data rules to be specified. Error Sentinel enforces that all good data rules match, or else the given input data is considered invalid. ErrorChecker.pl supports the same behavior, but it can also be specified that only a single good data rule must match. This feature is used in the correction of select columns. It will be made clear as to which behavior has been selected.

The matchers and replacers in ErrorChecker.pl that are relevant to the “Entire full With Database” rule set are detailed below in Table 12.
Table 12: The various matchers and replacers used by ErrorChecker.pl in the “Entire Full with Database” ruleset.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>Matcher</td>
<td>Matches if the value yielded from the concatenation of all given parameters is the same as the target.</td>
</tr>
<tr>
<td>!exact</td>
<td>Matcher</td>
<td>Same as the Exact Matcher, but it negates the result.</td>
</tr>
<tr>
<td>exact</td>
<td>Replacer</td>
<td>Concatenates all parameters together and returns the result.</td>
</tr>
<tr>
<td>regex</td>
<td>Matcher</td>
<td>Concatenates all its parameters together and treats the result as a regular expression. Matches if the target matches the regular expression.</td>
</tr>
<tr>
<td>!regex</td>
<td>Matcher</td>
<td>Same as the Regex matcher, but it negates the result.</td>
</tr>
<tr>
<td>database</td>
<td>Matcher</td>
<td>Takes a database type, database username, database password, database name, and an SQL statement as parameters. Matches if the target is contained in the results.</td>
</tr>
</tbody>
</table>

Note that the “Entire Full without Database” ruleset is the same as the “Entire Full with Database” ruleset, sans the rules that involve database matchers. As such, a separate ruleset for “Entire Full without Database” has not been presented.

The following are the actual rules used. The structure of these tables mimics the specification for actual rules in ErrorChecker.pl. Rules for determining good data are presented before rules for error correction, and do not have associated replacers. Order of rules is significant, as they are applied in the presented order. Rules are put into separate tables according to what columns they apply to. The value in the column is used as the aforementioned target. Columns without presented rulesets have no rules associated with them, and are considered valid without question. See Table 3 for a full enumeration and description of all columns in the mtHaplogroups data set.
There are no error correction rules for the haplogroup in this ruleset. The subhaplogroup error correction rules are presented below in Table 13. It was specified that all good data rules must match.

*Table 13: Error correction rules for the subhaplogroup.*

<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>&quot;^([A-Z][a-z]\d\d\d] PRE HV1\d[HV[^da-z]\d\dJT]\dJT]v1^[A-Za-z0-9]\d\d\d$&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>exact</td>
<td>&quot;-&quot;</td>
<td>exact</td>
<td>haplogroup</td>
</tr>
<tr>
<td>regex</td>
<td>&quot;^K1B1&quot;</td>
<td>exact</td>
<td>&quot;K1b1&quot;</td>
</tr>
<tr>
<td>exact</td>
<td>&quot;K &quot;</td>
<td>exact</td>
<td>&quot;K&quot;</td>
</tr>
</tbody>
</table>

As shown in Table 13, the subhaplogroup is verified with a regular expression. Note that in reality this line takes up a single line. Also note that this oversimplifies the problem of correcting subhaplogroups, and the final ruleset used uses a different entirely method to perform this check. This regular expression is the only good data rule used in the correction of the subhaplogroup.

Actual error correction rules, as opposed to good data rules, follow. The dash replacement rule is the first of these. As shown, it uses the haplogroup as a replacement (note the lack of quotes is significant). The next two rules fix two common typographical errors found in the data set.

The rules for verifying the basepair column are shown below in Table 14.
Table 14: Error correction rules for the basepair.

<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>&quot;^\d+$&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Note that there is only a single good data rule for the basepair, with no error corrections. This correlates both to the simplicity of the basepair, and its position in the graph in Figure 6. Being at the “top”, there is nothing that it must be consistent with, though other data must be consistent with the basepair. The regular expression merely verifies that it has a sequence of digits, as with an integer value.

Table 15 shows the error correction rules for the reference sequence allele. It was specified that all good data rules must match.
Once again, there are no replacement rules in Table 15. The first rule merely verifies that the given allele is an A, T, C, or G, representative of DNA nucleotides. The query used in the next rule gets the reference allele for the given basepair. Note that the second rule alone would be correct. The first rule is intended to avoid database queries where possible, as there are no reference alleles that are not one of A, T, C, or G. Additionally, in the “Entire Full without Database” ruleset, this database matching rule is nonexistent, so the first rule asserts that there is still some error correction performed on the reference sequence allele.

Table 16 shows the error corrections used for the variant allele. It was specified that only a single error correction rule must match.
The error corrections for the variant allele shown in Table 16 are very similar to the rules for the reference allele, shown in Table 15. The first rule is identical, and represents a common case that does not require a database query. The second rule merely retrieves all the known variant alleles possible, checking that the target variant allele is among them.

Table 17 shows the error corrections used for the locus. It was specified that all good data rules must match.

Table 16: Error correction rules for the variant allele.

<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>&quot;^[ATCG]$&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>database</td>
<td>&quot;SQLite&quot;,&quot;&quot;,&quot;,&quot;&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The error corrections for the variant allele shown in Table 16 are very similar to the rules for the reference allele, shown in Table 15. The first rule is identical, and represents a common case that does not require a database query. The second rule merely retrieves all the known variant alleles possible, checking that the target variant allele is among them.

Table 17 shows the error corrections used for the locus. It was specified that all good data rules must match.

Table 17: Error correction rules for the locus.

<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>!exact</td>
<td>&quot;NONCODING NUCLEOTIDES&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>!regex</td>
<td>&quot;MT-ATP\d/MT-ATP \d&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The error correction rules for the locus shown in Table 17 are the most complex so far. The first two rules explicitly invalidate two loci found in the data set. In the case of “NONCODING NUCLEOTIDES”, there is no such locus, and so it is clearly erroneous. In the case of "MT-ATP\d/MT-ATP\d", this is technically correct, though it is problematic for other reasons. The database query verifies both that the given locus exists, and that the specified basepair is held within the locus’ range.

As for the locus error correction rules, the first two correct common typographical errors. The second two replace alternative locus names with more standard names. In this case,
although the locus names are technically correct, they add no further value. To simplify searching, only distinct locus names are permitted, and so these are replaced.

The error correction rules used for the functional change are shown below in Table 18. It was specified that all good data rules must match.

Table 18: Error correction rules for the functional change.

<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>!regex</td>
<td>&quot;[;:/]+&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>!exact</td>
<td>&quot;NONE&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>!exact</td>
<td>&quot;NONCODING&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>!regex</td>
<td>&quot;SYN.+&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>regex</td>
<td>“NONCODING”</td>
<td>exact</td>
<td>“NONCODING NUCLEOTIDES”</td>
</tr>
<tr>
<td>regex</td>
<td>“^SYN”</td>
<td>exact</td>
<td>&quot;SYN&quot;</td>
</tr>
</tbody>
</table>

As shown in Table 18, the functional change is more free form than some of the other datums. Note that in the final error correction set used, more constraints are placed on the functional change.

The first good data rule explicitly invalidates functional changes that contain colons, semicolons, and forward slashes. It has been previously found that all such functional changes are invalid, hence this explicit removal. The next two rules explicitly invalidate two synonyms for “NONCODING NUCLEOTIDES”. As already mentioned in the presentation of the locus error correction rules, synonyms add complication without value, and so such are considered errors. The next good data rule is targeted at certain typographical errors regarding the
functional change “SYN”. Such errors all started correctly, and therefore this rule invalidates anything that starts correctly but is followed by erroneous information.

The first replacement rule replaces one of the aforementioned synonyms with a previously approved synonym, namely “NONCODING NUCLEOTIDES”. The second replacement rule replaces the aforementioned problematic instances of “SYN” with a single correct instance.

The only rules that exist in “Full with Database” that have not yet been presented are those for citations. While there is only a single good data rule defined for citations, there are 74 error correction rules. These rules form the bulk of the error correction rules defined in the “Full with Database” ruleset. These rules do not correct errors per se, but are rather used to establish a mapping between textual citation information and numerical citation IDs used in the mtHaplogroups relational database. As such, these rules are extremely lengthy and repetitive.

Rather than present all the replacement rules, a select number of them have been presented as examples. The rest of the rules follow the same pattern. All the good data rules and selected replacement rules are shown below in Table 19.

Table 19: Error correction rules for citations. All good data rules have been shown, though only two of 74 error correction rules have been shown. The remaining error correction rules follow the same pattern.
<table>
<thead>
<tr>
<th>Matcher Name</th>
<th>Matcher Parameters</th>
<th>Replacer Name</th>
<th>Replacer Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>&quot;ACHILLI ET AL. 2004/RUIZ-PESINI ET AL. 2007&quot;</td>
<td>exact</td>
<td>&quot;103/161&quot;</td>
</tr>
</tbody>
</table>

As shown in Table 19, multiple citations can be linked to a single entry. These are delimited with a forward slash (/). Although it may seem like poor practice to use IDs directly in this fashion, this was found to be necessary. Initial revisions attempted to determine these automatically. This process was found to be enormously complex, was occasionally incorrect, and a few edge cases required direct id mapping anyway. This mapping is actually significantly shorter than the now defunct solution of dynamically determining IDs.