PharmaSUG 2025 - Paper OS-145

Code Switching: Parallels Between Human Languages and Multilingual Programming

Danielle Stephenson and Laura Mino, Atorus Research

ABSTRACT

As multilingualism gains importance in the pharma industry, we've observed that the processes for learning programming languages closely mirror those for learning human languages. What insights can we draw from the study of natural languages to enhance our approach to multilingual programming? What similarities can we find between a SAS® programmer learning R and a native Mandarin speaker learning English? Join us, two polyglots who enjoy both people-talk and code-talk, as we explore natural and programming languages to see how techniques for understanding one can be leveraged to understand the other.

INTRODUCTION

Between, between, drink a chair – for the water zero is coming, beloved! We wish to speak to you today about a matter which will stir your livers, and help you to live better lives. This delightful excerpt is a direct translation from one cross-cultural worker's first public speech in his second language, and it highlights just how complex language can be.

Language is a powerful tool for communication and creation. It is also a powerful tool for miscommunication and frustration, especially as we explore beyond our native tongue. Anyone who has stumbled their way through the hilarity and exasperation of learning a new language has a deep understanding of how carrying assumptions from our first language into our second or third can cause serious misunderstandings.

In the pharma industry, programmers who work in more than one programming language are referred to as "multilingual." Interestingly, the field of linguistics uses that same term to describe those who speak more than one natural, human language. Programming languages (used by humans to communicate with computers) share many features of human languages (used by humans to communicate with other humans), and the much-studied process of learning human languages can yield many lessons that apply directly to the process of learning programming languages. The field of linguistics as it applies to human languages has a wealth of insights that we can pull from as we guide our teams into the realm of multilingual function.

WHAT IS LANGUAGE?

One of the definitions that Mirriam-Webster lists for the word "language" is *"a systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures, or marks having understood meanings."* (Mirriam-Webster, n.d.) It is easy to see why humans use this same term to describe the ways in which we communicate with computers. Cambridge dictionary highlights this in their definition of "natural language": *"language that has been developed in the usual way as a method of communicating between people, rather than language that has been created, for example, for computers*." (Cambridge, n.d.)

Both the beauty and the frustration of human languages lie in what makes them unique and different from one another. English author and journalist Geoffrey Willans once said, "You can never understand one language until you understand at least two." No two languages describe the world in the same way. Unique perspectives, disparate distinctions, and new concepts can be found in each language we add to our lives. One language spoken in the South Pacific, for example, has a specific word for *the sound which mangrove mud makes as it squishes between the toes*. Meanwhile, that same language has no specific word which only means *thank you*. Similarly, the Cherokee language has no word or phrase which means

goodbye; speakers instead use ᏙᎾᏓᎪᎶᏆ (*donadagohvi*, “until we meet again”) or ᏩᏙ (*wado,* “thank you”) for their farewells.

Human languages, with all their differences, hold a vast array of insights into the values, worldviews, and priorities of different cultures. Navigating these nuances profoundly transforms a person's worldview.

Human and programming languages share many similarities both in purpose and in process. Much like human languages, programming languages have different strengths and weaknesses -- different concepts that they are particularly strong at communicating and others that they are not ideal for. Gaining fluency in multiple programming languages gives us a deeper understanding of the nuances of each language we use. This process also adds tools to our toolbelt so that we can navigate each project with the confidence that comes from knowing that we are using the best language for the job.

LANGUAGE LEARNING TECHNIQUES

When we understand that human languages and programming languages share many features, we gain a wide variety of tools and ideas from the discipline of linguistics. Humans have already invested centuries of time and effort into discovering the best ways to learn other human languages – we can utilize those tools and ideas and apply them to the learning of programming languages as well.

IMMERSION

As babies, we learn our first language by immersion. We are surrounded by our native language, and as we gain the desire to communicate, we must do so in that language. Later in life, immersion as a language learning technique mimics this first experience and is a proven method to drive rapid and effective learning. With no easier communication options available, learners are forced to use their target language for every necessary task. This improves learning and retention far better than simply completing coursework in a language or taking a class while living in an environment dominated by their native language.

However, humans cannot exactly immerse themselves in the R programming language the same way an American could immerse themself in Spanish by traveling to Mexico for a month. How can we unlock the benefits of immersion for our programming language learners?

One way to apply this learning technique is to allow programmers who are learning a new language to spend as much of their time as possible working in that language for the first several months. Rather than asking programmers to attend classes learning R while mostly working in SAS®, companies can greatly speed up learning times and increase learning competencies by budgeting the time for their newly multilingual programmers to work entirely (or nearly entirely) in their target language for a large chunk of time. The experience of needing to figure out every day-to-day task in the target language mimics immersion and drives faster and more effective fluency for learners.

ENGAGING MULTIPLE NEUROPATHWAYS

Though the field of pedagogy (the study of how to best transmit knowledge) is relatively new, and the field of education even newer and less explored, there are a few key strategies that are commonly agreed upon to aid learning. When learning any new concept, language-related or otherwise, our brains seem to respond better when multiple parts of them are activated. Engaging multiple senses and giving several different kinds of activities to students improves their retention, fluency, and understanding. This is part of why immersion is beneficial to learning a language – every part of the brain which might normally be used throughout the day is still activated, but now linked to the new language.

Possibly the least effective way anything can be taught is by having the students sit and listen to someone talk about it for an hour a day. To add variety and engage multiple neuropathways, common dogma in pedagogy is to break up a class into segments of a shorter length. Following this framework, a class might begin with 20-30 minutes of instruction, followed by a period of hands-on exploration and experimentation, then conclude with a 15-minute wrap-up. In a typical language-learning class, for example, this would look like the teacher introducing the concept of verbs and introducing a small vocabulary list of new verbs. Then, students are given worksheets with example sentences missing verbs

and are instructed to choose an appropriate verb to complete the sentence, working either individually or in small groups. The last 15 minutes of the class are spent with the teacher again lecturing at the board, showing solutions to sentences from the worksheet or answering questions that the students encountered during the experimentation phase.

We can easily apply the same framework to programming. A self-paced learner spends an initial period of time learning a new concept: reading a chapter of a book explaining it, reviewing prewritten code from an expert, or browsing relevant vignettes. Then the learner moves into the experimentation phase by attempting to write their own code to address the concept. The learner can "wrap up" their own learning by testing the code.

If the code does not execute exactly as desired on the first try, then the learner can begin a new iteration of learning the concept: review the book chapter on the concept, search community forums for conversations with peers who experienced similar issues, debug the code, etc. They modify their code in another experimentation phase, and the process repeats until the programmer reaches their goal of understanding.

UNDERSTANDING BEYOND DIRECT TRANSLATION

As a learner starts to acquire a new language, it is natural to begin with direct translation: x = y. The word for *please* in Fijian is *kerekere.* However, as we progress further, this basic understanding has serious limits. Equating words directly does not allow for the truth that languages have different perspectives, different emphases, and different understandings of the world. In order to gain fluency, it is essential that we grasp our new language in a way that goes far beyond direct translation.

The word *kerekere* in Fijian actually means something much more akin to *I am invoking a debt which you owe me.* Although it is used in the same place and for a similar purpose as the English word "please," it carries quite a different underlying meaning. In order to simply add politeness to a request, in Fijian the adverb *mada* (pronounced "manda") could be used. Fully understanding and correctly using the words *kerekere* or *mada* comes with a deeper understanding of the values, worldview, and assumptions of the Fijian culture, and requires that native English speakers move away from directly translating the English concept of *please*.

In much the same way, we often begin learning a new programming language by equivocating things as a direct translation. This command equals this other command. This idea equals this other idea. While this is a necessary initial approach, as we strive to achieve fluency, we must work to look deeper and understand the fundamental differences between language assumptions and underlying processes so that we may appreciate the subtle differences beyond direct translations.

For example, a new R programmer coming from a background in SAS® might be told that a dataset is called a data frame in R. "Dataset" and "data frame" are similar names, and on the surface, they do appear to operate the same way. For beginner R programmers, treating them as if they are identical is not likely to cause any problems – when programming with a flow similar to what might be used in SAS®, an R data frame can be used in much the same way as a SAS® dataset.

However, the truth is that SAS® datasets and R data frames function quite differently in the background. Continuing to assume equality rather than understanding the nuances of how each programming language handles data can lead to lost functionality at best and real errors at worst, as the programmer progresses into R fluency.

A SAS® dataset is two-dimensional in terms of how data are stored. The basic unit in SAS® is a variable, organized into rows (a horizontal dimension) and columns (a vertical dimension). Each variable holds a single piece of information (either a number or a character string) and can have three location attributes within a program (dataset, row, and column).

R data frames, on the other hand, are only one possible piece of R's data storage system -- and they are capable of much more complexity. R stores data arranged into vectors, which allows for nearly unlimited dimensions.

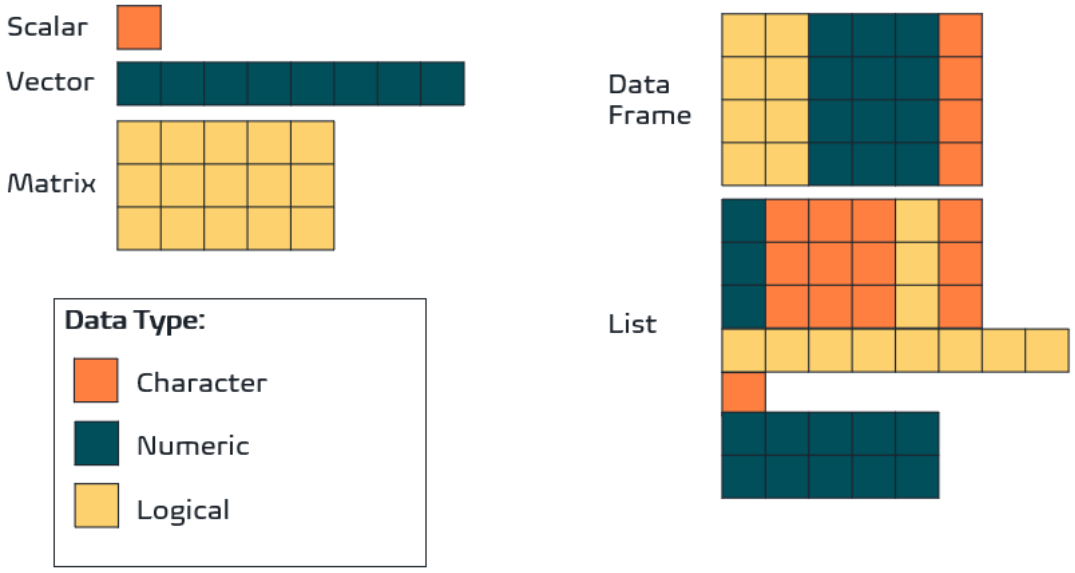

Figure 1. Visual representation of various object types that data may be stored in within R. Note that scalars, vectors, and matrices may contain any type of data so long as only one type is used throughout. For example, this figure displays a matrix of logical data, but matrices are not limited to only logical data. A matrix can contain character, numeric, or logical data; the only restriction is that all data is of the same type.

Data in R can be stored as various object types as shown in Figure 1.

A scalar is a single piece of information, which can be connected to other scalars to form vectors. Vectors, then, are sequences of elements that lack dimensionality – they are neither horizontal nor vertical, only having a length; they are simply a connected sequence of elements, like a chain with multiple connected links. Vectors can then be arranged into matrices or data frames, at which point they gain dimension. Matrices and data frames are structured in rows and columns by combining vectors of equal size, where each vector becomes a column. A matrix contains only a single type of information across all columns, while a data frame only needs to have the same type of information within a single column. Finally, lists can incorporate scalars, vectors, matrices, or data frames of any length, and have no restrictions on data type combinations.[1]

To add to the complexity, while a vector within a data frame can simply be a single column of information, each element within the vector could itself be a vector. So a data frame may hold a two-dimensional set of information, or each cell might itself hold a two-dimensional set of information, a multi-dimensional set of information, or even a set of more data frames. R begins to look much more like a fractal, with information all the way down.

Why does this matter to the clinical data programmer? By assuming that an R data frame and a SAS® dataset are exactly identical in how they store data and how they can be manipulated, the programmer may inadvertently lock themself out of being able to do several sophisticated and advanced manipulations on their R data. For example, we can access and manipulate every vector of a certain type across every data frame we are working with at once, without loading each individual data frame. There are plenty of cases with simple data programming where working within the relatively simpler anatomy of a SAS®

[1] R also includes an object type called arrays, which are 3- and 4-dimensional structures for storing data. While arrays are beyond the scope of this discussion, it is good to note their existence as yet another example of R's data storage complexity.

dataset could be the best option: there's less opportunity for unexpected complications, and less computing power required to handle the entire collection of data. However, understanding the R data frame as something which can *operate* like a SAS® dataset, but which is not quite the same thing, allows fluent multilingual programmers to determine the best tool for the job as they group, manipulate, and access data.

LEARNING AND TRANSLATION CHALLENGES

It is clear from these examples that we can gain positive insight into the process of programming language by learning from the field of human linguistics. In the same way, staying aware of the pitfalls of second language communication will allow us to avoid similar traps as we learn new programming languages.

UNTRANSLATABLE CONCEPTS

Humans across the globe share core experiences, feelings, and concepts. All people know what happiness, loneliness, and frustration feel like, regardless of where they live or what language they speak. So it can be jarring and unexpected when we do encounter a concept that simply does not have a translation in our target language. Historically, there was no one single English word that encompassed "the feeling of joy or satisfaction at another person's misfortune," which is why the German word *schadenfreude* became popular among English-speakers to describe that very specific feeling. In fact, "schadenfreude" became so popular that it is now officially included in the Oxford English Dictionary. (Oxford University Press, n.d.) Adopting a loanword to transfer a previously untranslatable concept is one way that linguistics overcomes gaps in translations, but not every instance can be handled this way. It is important to bear in mind that sometimes there are concepts or sounds in our native language that do not have an equivalent in our target language, or that there are ideas that we may have to learn from scratch in our target language because our native language does not have an equivalent.

Unique Concepts

Certain human languages utilize sounds unique to that language, which simply are not common or shared in all spoken languages. For example, the common Mandarin word 谢谢 ("thank you") contains a sound that does not exist in the English language.

Linguistics has a few different workarounds for this problem. One is to use an approximation from the native language. Pinyin is a system in which Mandarin sounds are represented using agreed-upon letters from the English alphabet such as "x," "q," or "j," even though the true sounds to not match up with the way that English uses those letters. The word 谢谢 is written in pinyin as "xièxie." The sound represented here by "x" is not present at all in English. It resembles a "hs" sound – nothing like the "ecks" sound that an English speaker might associate with the letter "x."

As the new Mandarin speaker learns their target language, they will likely wish to move beyond these approximations and install a Chinese keyboard into their word processing software so that they can correctly type the words that a native Mandarin speaker will know exactly how to pronounce: 谢谢.

In a similar way, SAS® has several concepts that R does not, and vice versa. An excellent example is formats.

How might we check a data frame in R to make sure that the DATE9 format has been correctly applied to TRTSDT and TRTEDT? The answer to that question is that, unfortunately, it is not possible. R does not support formats – they are a concept that does not exist in R the same way that they are built into SAS®.

Like using pinyin to represent Mandarin sounds, R users can add metadata to their data frame indicating that specific columns will follow the DATE9 format and then export the data as an XPT file. While this

metadata may not be intelligible to R, it will show up as desired when a native SAS® software reader, such as SAS® Universal Viewer, reads the XPT file.

Untranslatable Distinctions

Unfortunately, sometimes there are untranslatable concepts that do not have such an easy workaround. The Spanish language utilizes the tilde accent to transform a lower-case letter “n” to “ñ.” These two letters have slightly different sounds in Spanish, with the accented “ñ” more chewed in the mouth, but they are audibly quite similar for an English speaker. However, that little squiggly line can make quite the difference in the meaning of a word. In Spanish, the word *año* means “year” as in the question, “Cuantos años tienes?” or “How many years do you have?” to ask a person their age. Forgetting the tilde and associated inflection on that letter transforms *año* (year) into *ano* (anus). Thus, “Cuantos anos tienes?” is a much different, and much less appropriate, question.

In a similar way, R has subtle distinctions that SAS® does not. Data in SAS® falls into one of two categories: character data (such as text strings for variables names) or numeric data (such as values for test results). In addition to character and numeric data, R also has a third data type: logical. Logical data in R is Boolean, with values of either TRUE, FALSE, or NA. These values are not the simple TRUE/FALSE character strings that we find in SAS®, but rather are logical values that can be manipulated in many ways. For example, Boolean values can be created within a {dplyr} summarize function and then quickly summed to get the number of records meeting various criteria in a few simple statements.

```
i_interval%>%
  group_by(USUBJID, APERIOD)%>%
  summarize(TIR  = sum(70 <= AVAL & AVAL <= 180),
            TBTR  = sum(54 <= AVAL & AVAL < 70),
            TVBTR = sum(!is.na(AVAL) & AVAL < 54))
```

Program 1. Summarizing Boolean Values in R

In order to use TRUE/FALSE values to perform this calculation in SAS®, we must take a completely different approach, because these TRUE/FALSE values in SAS® are simply characters and are not something that we can perform arithmetic with.

```
data i_interval;
   set i_interval;
   if 70 <= AVAL and AVAL <=180 then TIR = 'TRUE'; else TIR = 'FALSE';
   if 54 <= AVAL and AVAL < 70 then  TBTR = 'TRUE'; else TBTR = 'FALSE';
   if AVAL ne . and AVAL < 70 then   TVBTR = 'TRUE'; else TVBTR = 'FALSE';
run;

proc freq data=i_interval;
  table usubjid*aperiod*TIR/ out = TIR;
run;

proc freq data=i_interval;
  table usubjid*aperiod*TBTR/ out = TBTR;
run;

proc freq data=i_interval;
  table usubjid*aperiod*TVBTR/ out = TVBTR;
run;
```

Program 2. Summarizing Records in SAS® by TRUE/FALSE Values

With an understanding of the data types available to us in SAS®, we might take a different approach in order to get the values we need in fewer steps.

```
data i_interval;
   set i_interval;
   if 70 <= AVAL and AVAL <=180 then do; GROUP = 'TIR'; output; end;
   if 54 <= AVAL and AVAL < 70 then do;  GROUP = 'TBTR'; output; end;
   if AVAL ne . and AVAL < 54 then do;   GROUP = 'TVBTR'; output; end;
run;

proc freq data=i_interval;
  table usubjid*aperiod*group/ out=f_interval;
run;
```

Program 3. Summarizing Records in SAS® by Group

Understanding the distinctions between different data types in SAS® and R allows us to manipulate data using the techniques which are most efficient in each language, rather than attempting to mimic the logic used in one language to perform a similar task in the second language.

FALSE COGNATES

Some of the most frustrating and embarrassing phenomena in language learning are false cognates. These deceptive words *sound* the same across multiple languages but they *do not have the same meaning at all*. Sometimes this is due to divergent language evolution, where each word originally came from the same root in a parent language but shifted over time to different meanings; alternatively, it could be complete coincidence arising from the fact that there are only so many combinations of sounds producible by the human mouth. Many a new Spanish speaker has declared themselves to be "*embarazada*" only to be even MORE embarrassed to realize that they have announced themselves to be pregnant when they thought that they were admitting to being embarrassed. A new Russian speaker might see a sign for a "Магази́н" (magazine) only to realize that rather than a periodic paper publication, they will find a shop.

Figure 2. A storefront in Bishkek City. On the left side of the store's signboard, the word "магазин" (transliterated into English as "magazine") indicates this building is a store, though the false cognate may be confusing to English-speaking visitors. Image courtesy of Google Earth (Google, n.d.).

As programmers learn more than one programming language, it's important to keep this concept of false cognates in mind. The same term may be used in different programming languages to mean completely different things, which can confuse beginning multilingual programmers.

In SAS® software, we might use a “factor” when coding various statistical models. There is even an entire procedure called PROC FACTOR. The SAS® language mostly uses the term “factor” similarly to the way we might use the word in plain, spoken English, as in the phrase “factors to be aware of.” As a statistical model is coded, we will list the factors for the model to account for – things like visits, subjects, and treatments.

However, in R, the term “factor” means something completely different. In R, a factor is a data structure that allows us to work efficiently with categorical variables that have a fixed and known set of possible values by assigning integer levels to each value.

Assuming that terms which sound the same in two different languages refer to the same concept in both is an easy trap to fall into, and new language learners can greatly benefit from taking a moment to double check that a given word is not a false cognate.

LANGUAGE ARTIFACTS

One of the most amusing linguistic challenges is artifacts. These can arise when the context that is needed to understand a concept is missing at some point in the language process, resulting in things that seem nonsensical to a new learner. Learning the context surrounding the way a given term or process was created can greatly help comprehension.

The Cherokee language uses the same word for “butterfly” and “elephant”: ᏧᎾᏍᏗ (*kamama*). This seems ridiculous at first glance, but a bit of context can help it make sense. When Cherokees and Europeans first came into contact, there were no elephants in North America, so the only point of reference the Cherokees had for the creatures that Europeans were describing came from pictures, drawings, and descriptions. None of these showed the scale or size of the animal. With size removed from consideration, the shape of an elephant’s ears resembles the shape of a butterfly’s wings. An elephant’s trunk, which it uses for drinking water, sounded very much like a butterfly’s tongue, used for sipping nectar. Thus, Cherokees applied their word for butterfly to the very similar creature which the Europeans were describing.

In the same way that understanding historical context here helps reduce the ridiculousness of the term, a bit of background context may help programmers coming to SAS® from R who are frustrated with the row-wise processing of SAS®.

One of SAS® software’s biggest superpowers lies in the speed with which it processes large amounts of data. Part of the reason that SAS® is able to process large quantities of data faster than other data analytics languages like SQL and R is the fact that SAS® reads and manipulates data one row at a time. This row-wise processing can be frustrating for a programmer coming from SQL or R who is used to being able to manipulate many rows of data at once or call data from one row to another, but it can help to understand the context: this seeming limitation in SAS® actually powers its advantages.

GRAMMAR DIFFERENCES

One of the most challenging aspects for language learners to grasp is the grammatical structure and nuance of their target language. Verb conjugation, sentence structure, and parts of speech -- the underlying processes of language composition -- are often so subconscious that it becomes difficult to identify and understand both the structure of our native language and how it differs from the target language.

Order of Operations

A very simple example of linguistic grammar differences is order of operations. In English we might say “you do not understand.” The sentence is structured such that you, the subject, physically comes before the action of understanding. In addition, the negative “not” comes in the middle of the compound verb “do understand.” However, in Afrikaans we might say “verstaan jy nie.” Here, the order of operations places the action first, then the subject, and then the negative. This is also true of programming languages. When learning a new programming language, it is important to take note of the “grammar,” or the structure of logic.

In SAS® software, we would place the condition of an if/else clause before the object:

```
if aval > 0 then anl01fl = 'Y';
   else anl01fl = '';
```

Program 4. SAS® if/else Clause

However, in the {dplyr} package in R, we would place the object before the action:

```
anl01fl = if_else(aval > 0, 'Y', '')
```

Program 5. R if/else Clause Using {dplyr}

Implicit and Explicit Pieces

While the previous example is a very simple instance, grammar differences can be much more complex with more far-reaching implications than simple word order.

A Spanish speaker might have a difficult time remembering to include the subject in their sentence when speaking English. In Spanish, the subject is implicit in the verb conjugation, while in English, verb conjugation accounts for fewer intricacies and sentence subject must be explicitly stated. "I see the house" becomes "Veo la casa" (literally, "see the house") -- the word *yo* ("I") does not need to be included, because it is implicit in the verb *veo*.

In a very similar way, programming languages have their own implicit and explicit processes and grammar structure. Neglecting or ignoring these differences can lead to issues. The above-mentioned if/else commands in SAS® and R yield more difficulties than simply word order as we move into more complex instances.

```
if index(PCORRES, '<') = 0 then PCSTRESN = input(PCORRES, best.);
   else PCSTRESN = .;
```

Program 6. Complex SAS® if/else Clause

```
PCSTRESN = if_else(str_detect(PCORRES, '<') == FALSE, as.numeric(PCORRES), NA)
```

Program 7. Complex R if/else Clause Using {dplyr}

In this case, it is essential to understand the underlying order of operations (grammar) of each programming language, and what pieces of logic are implicit or explicit in each language.

Because SAS® works row-by-row, when an if/else clause is introduced, it is implicit within SAS® software's structure that all derivations following that clause are only applied to rows which meet the initial criteria. In this example, the derivation `input(PCORRES, best.)` is only attempted on lines where `PCORRES` does not contain `<`. If the input fails, then we know that `PCORRES` must contain values other than numbers or records containing `<` and we can look into what other situations we must account for in our programming logic.

However, R does not work row-wise. In R, each function is applied to the entire column and then the results are only passed on to the cells which meet the if/else criteria. So, in this case, the `as.numeric(PCORRES)` function will be applied to every row and then the results will only be output on records where `PCORRES` does not contain `<`. This statement will result in the warning "*NAs introduced by coercion"* because `as.numeric` is attempting to convert the `PCORRES` values which contain `<` as well as those which do not. The `if_else()` function only tells R where to apply the ***result*** of the derivation, ***not*** what rows to attempt the derivation on.

An understanding that goes beyond direct translation allows us to find a different approach in R, in which we only pass to each derivation objects which it can handle without error. For example, in this situation we might use:

```
PCSTRESN = if_else(str_detect(PCORRES, '<') == FALSE, as.numeric(gsub('<', '', PCORRES),
NA)
```

Program 8. Corrected Complex R {dplyr} if/else Clause

Here we are only passing number values to the `as.numeric()` function and have already removed the non-numeric characters.

CONCLUSION

For the pharma industry, it is becoming increasingly obvious that the future is multilingual. Leveraging multiple languages in our statistical programming teams allows us to take advantage of the strengths of each language and find solutions for the weaknesses of any given tool. As team leaders and individual programmers, we can leverage an understanding of the process of human language learning in this transition.

Effective techniques and common pitfalls of language learning have already been explored in the human linguistics space, and accessing those insights allows us to identify and overcome the same challenges in the process of learning programming languages.

REFERENCES

Cambridge. (n.d.). Natural Language. In dictionary.cambridge.org dictionary. Retrieved March 24, 2025, from https://dictionary.cambridge.org/us/dictionary/english/natural-language

Google. (n.d.). [Google Maps location 42.83777012569536, 74.60554360563397]. Retrieved March 13,2025, from https://www.google.com/maps

Merriam-Webster. (n.d.). Language. In Merriam-Webster.com dictionary. Retrieved March 24, 2025, from https://www.merriam-webster.com/dictionary/language

Oxford University Press. (n.d.). "Schadenfreude." In Oxford English Dictionary. Retrieved March 25, 2025, from https://www.oed.com/dictionary/schadenfreude

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Danielle Stephenson
Atorus Research
+1.719.314.7547
Danielle.stephenson@atorusresearch.com

Laura Mino
Atorus Research, Atorus Academy
Laura.mino@atorusresearch.com