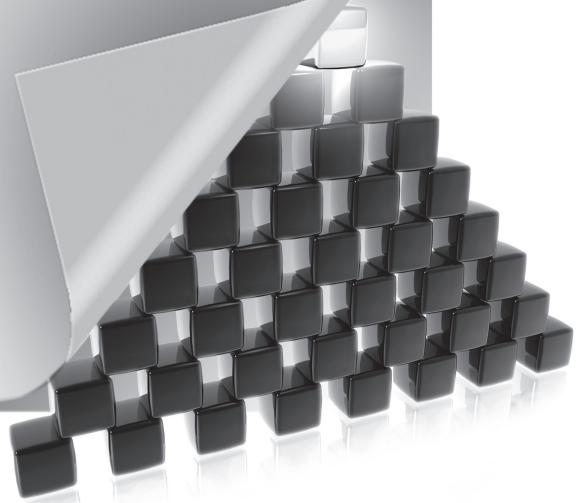


Introduction to Regular Expressions in SAS®

K. Matthew Windham



support.sas.com/bookstore

The correct bibliographic citation for this manual is as follows: Windham, K. Matthew. 2014. *Introduction to Regular Expressions in SAS*[®]. Cary, NC: SAS Institute Inc.

Introduction to Regular Expressions in SAS[®]

Copyright © 2014, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-61290-904-2 (Hardcopy)

ISBN 978-1-62959-498-9 (EPUB)

ISBN 978-1-62959-499-6 (MOBI)

ISBN 978-1-62959-500-9 (PDF)

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

December 2014

SAS provides a complete selection of books and electronic products to help customers use SAS[®] software to its fullest potential. For more information about our offerings, visit support.sas.com/bookstore or call 1-800-727-0025.

SAS[®] and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Contents

About This Book	vii
About The Author	xi
Acknowledgments	xiii
Chapter 1: Introduction	1
1.1 Purpose of This Book	1
1.2 Layout of This Book	1
1.3 Defining Regular Expressions.....	2
1.4 Motivational Examples	3
1.4.1 Extract, Transform, and Load (ETL).....	3
1.4.2 Data Manipulation	4
1.4.3 Data Enrichment	5
Chapter 2: Getting Started with Regular Expressions.....	9
2.1 Introduction	10
2.1.1 RegEx Test Code	11
2.2 Special Characters.....	13
2.3 Basic Metacharacters.....	15
2.3.1 Wildcard.....	15
2.3.2 Word.....	15
2.3.3 Non-word.....	16
2.3.4 Tab.....	16
2.3.5 Whitespace.....	17
2.3.6 Non-whitespace.....	17
2.3.7 Digit.....	17
2.3.8 Non-digit.....	18
2.3.9 Newline	18
2.3.10 Bell	19

2.3.11 Control Character	20
2.3.12 Octal	20
2.3.13 Hexadecimal	21
2.4 Character Classes	21
2.4.1 List	21
2.4.2 Not List	22
2.4.3 Range	22
2.5 Modifiers	23
2.5.1 Case Modifiers	23
2.5.2 Repetition Modifiers	25
2.6 Options	32
2.6.1 Ignore Case	32
2.6.2 Single Line	32
2.6.3 Multiline	33
2.6.4 Compile Once	33
2.6.5 Substitution Operator	34
2.7 Zero-width Metacharacters	34
2.7.1 Start of Line	35
2.7.2 End of Line	35
2.7.3 Word Boundary	35
2.7.4 Non-word Boundary	36
2.7.5 String Start	36
2.8 Summary	37
Chapter 3: Using Regular Expressions in SAS	39
3.1 Introduction	39
3.1.1 Capture Buffer	39
3.2 Built-in SAS Functions	40
3.2.1 PRXPARSE	40
3.2.2 PRXMATCH	42
3.2.3 PRXCHANGE	43
3.2.4 PRXPOSN	46
3.2.5 PRXPAREN	47

3.3 Built-in SAS Call Routines	49
3.3.1 CALL PRXCHANGE	50
3.3.2 CALL PRXPOSN	54
3.3.3 CALL PRXSUBSTR	56
3.3.4 CALL PRXNEXT	57
3.3.5 CALL PRXDEBUG	59
3.3.6 CALL PRXFREE.....	62
3.4 Summary	63
Chapter 4: Applications of Regular Expressions in SAS	65
4.1 Introduction	65
4.1.1 Random PII Generator	66
4.2 Data Cleansing and Standardization.....	72
4.3 Information Extraction	77
4.4 Search and Replacement	80
4.5 Summary	83
4.5.1 Start Small	83
4.5.2 Think Big.....	83
Appendix A: Perl Version Notes	85
Appendix B: ASCII Code Lookup Tables	87
Non-Printing Characters	87
Printing Characters	89
Appendix C: POSIX Metacharacters	97
Index	101

About This Book

Purpose

This book is intended for a wide audience of SAS users, from novice programmer to the very advanced. As not much has previously been published on this topic, many different skill levels can benefit from the content herein. However, the book has been written to ensure that novice programmers can immediately implement every element discussed.

Is This Book for You?

Of course, it is! Do you wish you could process unstructured data sources? Would you like to more effectively process semi-structured data sources? Do you want to one day leverage advanced text mining concepts within your Base SAS code? Of course, you do! This book lays the foundation for all of this and more, making it the ideal text for anyone wanting to enhance their programming prowess.

Prerequisites

Readers should be comfortable using and applying the SAS DATA step, basic PROCs (e.g., PROC PRINT), DO loops, and conditional processing concepts. Readers should be familiar with SAS arrays and the RETAIN statement.

Scope of This Book

This book covers all PRX functions and call routines.

This book does NOT cover advanced concepts requiring MACRO programming, PROC SQL, or system automation.

About the Examples

Software Used to Develop the Book's Content

Base SAS (Microsoft Windows)

Example Code and Data

You can access the example code and data for this book by linking to its author page at <http://support.sas.com/publishing/authors>. Select the name of the author. Then, look for the cover thumbnail of this book, and select Example Code and Data to display the SAS programs that are included in this book.

For an alphabetical listing of all books for which example code and data is available, see <http://support.sas.com/bookcode>. Select a title to display the book's example code.

If you are unable to access the code through the website, e-mail saspress@sas.com.

Output and Graphics Used in This Book

All output used in this book was generated via the SAS log and PROC PRINT.

Additional Help

Although this book illustrates many analyses regularly performed in businesses across industries, questions specific to your aims and issues may arise. To fully support you, SAS Institute and SAS Press offer you the following help resources:

- About topics covered in this book, contact the author through SAS Press:
 - Send questions by e-mail to saspress@sas.com; include the book title in your correspondence.
 - Submit feedback on the author's page at http://support.sas.com/author_feedback.
- About topics in or beyond this book, post questions to the relevant SAS Support Communities at <https://communities.sas.com/welcome>.
- SAS Institute maintains a comprehensive website with up-to-date information. One page that is particularly useful to both the novice and the seasoned SAS user is its Knowledge Base. Search for relevant notes in the "Samples and SAS Notes" section of the Knowledge Base at <http://support.sas.com/resources>.
- Registered SAS users or their organizations can access SAS Customer Support at <http://support.sas.com>. Here you can pose specific questions to SAS Customer Support: Under *Support*, click *Submit a Problem*. You will need to provide an e-mail address to which replies can be sent, identify your organization, and provide a customer site number or license information. This information can be found in your SAS logs.

Keep in Touch

We look forward to hearing from you. We invite questions, comments, and concerns. If you want to contact us about a specific book, please include the book title in your correspondence to saspress@sas.com.

To Contact the Author through SAS Press

By e-mail: saspress@sas.com

Via the Web: http://support.sas.com/author_feedback

SAS Books

For a complete list of books available through SAS, visit <http://support.sas.com/bookstore>.

Phone: 1-800-727-0025

E-mail: sasbook@sas.com

SAS Book Report

Receive up-to-date information about all new SAS publications via e-mail by subscribing to the SAS Book Report monthly eNewsletter. Visit <http://support.sas.com/sbr>.

Publish with SAS

SAS is recruiting authors! Are you interested in writing a book? Visit <http://support.sas.com/saspress> for more information.

About The Author



K. Matthew Windham, CAP, is the director of analytics at NTELX Inc., an analytics and technology solutions consulting firm located in the Washington, DC area. His focus is on helping clients improve their daily operations through the application of mathematical and statistical modeling, data and text mining, and optimization. A longtime SAS user, Matt enjoys leveraging the breadth of the SAS platform to create innovative, predictive analytics solutions. During his career, Matt has led consulting teams in mission-critical environments to provide rapid, high-impact results. He has also architected and delivered analytics solutions across the federal government, with a particular focus on the US Department of Defense and the US Department of the Treasury. Matt is a Certified Analytics Professional (CAP) who received his BS in Applied Mathematics from N.C. State University and his MS in Mathematics and Statistics from Georgetown University.

Learn more about this author by visiting his author page at <http://support.sas.com/publishing/authors/windham.html>. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Acknowledgments

To my brilliant wife, Lori, thank you for always supporting and encouraging me in everything that I do. I couldn't have done this without you. To my friends and family, your advice and encouragement has been treasured.

While I have many people in my professional career to whom I owe a great debt, one in particular stands out. I would like to thank Nick Ferens for throwing me into the deep end of pool all those years ago. You saw more in me than I could, and completely changed my career for the better.

Finally, I would like to thank the editorial team at SAS Press, with whom I have truly collaborated in this endeavor: Shelley Sessoms, John West, Brenna Leath, Joan Keyser, Denise Jones, and Stacey Hamilton. Your patience, insight, and hard work have made this a wonderful experience.

Chapter 1: Introduction

1.1 Purpose of This Book	1
1.2 Layout of This Book	1
1.3 Defining Regular Expressions	2
1.4 Motivational Examples	3
1.4.1 Extract, Transform, and Load (ETL)	3
1.4.2 Data Manipulation	4
1.4.3 Data Enrichment	5

1.1 Purpose of This Book

This book is meant for SAS programmers of virtually all skill levels. However, it is expected that you have at least a basic knowledge of the SAS language, including the DATA step, and how to use SAS PROCs.

This book provides all the tools you need to learn how to harness the power of regular expressions within the SAS programming language. The information provided lays the foundation for fairly advanced applications, which are discussed briefly as motivating examples later in this chapter. They are not presented to intimidate or overwhelm, but instead to encourage you to work through the coming pages with the anticipation of being able to rapidly implement what you are learning.

1.2 Layout of This Book

It is my goal in this book to provide immediately applicable information. Thus, each chapter is structured to walk through every step from theory to application with the following flow: Syntax ► Example. In addition to the information discussed in the coming chapters, a regular expression reference guide is included in the appendix to help with more advanced applications outside the scope of this text.

Chapter 1

In addition to providing a roadmap for the remainder of the book, this chapter provides motivational examples of how you can use this information in the real world.

Chapter 2

This chapter introduces the basic syntax and concepts for regular expressions. There is even some basic SAS code for running the examples associated with each new concept.

Chapter 3

This chapter is designed to walk through the details of implementing regular expressions within the SAS language.

Chapter 4

In this final chapter, we work through a series of in-depth examples—case studies if you will—in order to ‘put it all together.’ They don’t represent the limitations of what you can do by the end of this book, but instead provide some baseline thinking for what is possible.

Appendixes

While not comprehensive, these serve as valuable, substantial references for regular expressions, SAS documentation, and reference tables. I hope everyone can leverage the additional information to enrich current and future regular expressions capabilities.

1.3 Defining Regular Expressions

Before going any further, we need to define *regular expressions*.

Taking the very formal definition might not provide the desired level of clarity:

Definition 1 (formal)

regular expressions: “Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively.”¹

In the pursuit of clarity, we will operate with a slightly looser definition for regular expressions. Since practical application is our primary aim, it doesn’t make sense to adhere to an overly esoteric definition. So, for our purposes we will use the following:

Definition 2 (easier to understand—our definition)

regular expressions: character patterns used for automated searching and matching.

When programming in SAS, regular expressions are seen as strings of letters and special characters that are recognized by certain built-in SAS functions for the purpose of searching and matching. Combined with other built-in SAS functions and procedures, you can realize tremendous capabilities, some of which we explore in the next section.

Note: SAS uses the same syntax for regular expressions as the Perl programming language². Thus, throughout SAS documentation, you find regular expressions repeatedly referred to as “Perl regular expressions.” In this book, I choose the conventions present in the SAS documentation, unless the Perl conventions are the most common to programmers. To learn more about how SAS views Perl, visit this website: http://support.sas.com/documentation/cdl/en/lefunctionsref/67239/HTML/default/viewer.htm#p0s9ila_gexmj18n1u7e1t1jfnzlk.htm. To learn more about Perl programming, visit <http://perldoc.perl.org/perlre.html>. In this book, however, I primarily dispense with the references to Perl, as they can be confusing.

1.4 Motivational Examples

The information in this book is very useful for a wide array of applications. However, that will not become obvious until after you read it. So, in order to visualize how you can use this information in your work, I present some realistic examples.

As you are all probably familiar with, data is rarely provided to analysts in a form that is immediately useful. It is frequently necessary to clean, transform, and enhance source data before it can be used—especially textual data. The following examples are devoid of the coding details that are discussed later in the book, but they do demonstrate these concepts at varying levels of sophistication. The primary goal here is to simply help you to see the utility for this information, and to begin thinking about ways to leverage it.

1.4.1 Extract, Transform, and Load (ETL)

ETL is a general set of processes for extracting data from its source, modifying it to fit your end needs, and loading it into a target location that enables you to best use it (e.g., database, data store, data warehouse). We’re going to begin with a fairly basic example to get us started. Suppose we already have a SAS data set of customer addresses that contains some data quality issues. The method of recording the data is unknown to us, but visual inspection has revealed numerous occurrences of duplicative records, as in the table below. In this example, it is clearly the same individual with slightly different representations of the address and encoding for gender. But how do we fix such problems automatically for all of the records?

First Name	Last Name	DOB	Gender	Street	City	State	Zip
Robert	Smith	2/5/1967	M	123 Fourth Street	Fairfax,	VA	22030
Robert	Smith	2/5/1967	Male	123 Fourth St.	Fairfax	va	22030

Using regular expressions, we can algorithmically standardize abbreviations, remove punctuation, and do much more to ensure that each record is directly comparable. In this case, regular expressions enable us to perform more effective record keeping, which ultimately impacts downstream analysis and reporting.

We can easily leverage regular expressions to ensure that each record adheres to institutional standards. We can make each occurrence of Gender either “M/F” or “Male/Female,” make every instance of the Street variable use “Street” or “St.” in the address line, make each City variable include or exclude the comma, and abbreviate State as either all caps or all lowercase.

This example is quite simple, but it reveals the power of applying some basic data standardization techniques to data sets. By enforcing these standards across the entire data set, we are then able to properly identify duplicative references within the data set. In addition to making our analysis and reporting less error-prone, we can reduce data storage space and duplicative business activities associated with each record (for example, fewer customer catalogs will be mailed out, thus saving

money!). For a detailed example involving ETL and how to solve this common problem of data standardization, see Section 4.2 in Chapter 4.

1.4.2 Data Manipulation

Suppose you have been given the task of creating a report on all Securities and Exchange Commission (SEC) administrative proceedings for the past ten years. However, the source data is just a bunch of .xml (XML) files, like that in Figure 1.1³. To the untrained eye, this looks like a lot of gibberish; to the trained eye, it looks like a lot of work.

Figure 1.1: Sample of 2009 SEC Administrative Proceedings XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <root>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61262.pdf</url>
    <release_number>34-61262</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Stephen C. Gingrich</respondents>
  </administrative_proceeding>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61256.pdf</url>
    <release_number>34-61256</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Gabelli Funds LLC</respondents>
  </administrative_proceeding>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61255.pdf</url>
    <release_number>34-61255</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Gabelli Funds LLC</respondents>
  </administrative_proceeding>
```

However, with the proper use of regular expressions, creating this report becomes a fairly straightforward task. Regular expressions provide a method for us to algorithmically recognize patterns in the XML file, parse the data inside each tag, and generate a data set with the correct data columns. The resulting data set would contain a row for every record, structured similarly to this data set (for files with this transactional structure):

Example Data Set Structure

Release Number	Release Date	Respondents	URL
34-61262	Dec 30, 2009	Stephen C. Gingrich	http://www.sec.gov/litigation/admin/2009/34-61262.pdf
...

Note: Regular expressions cannot be used in isolation for this task due to the potential complexity of XML files. Sound logic and other Base SAS functions are required in order to process XML files in general. However, the point here is that regular expressions help us overcome some otherwise

significant challenges to processing the data. If you are unfamiliar with XML or other tag-based languages (e.g., HTML), further reading on the topic is recommended. Though you don't need to know them at a deep level in order to process them effectively, it will save a lot of heartache to have an appreciation for how they are structured. I use some tag-based languages as part of the advanced examples in this book because they are so prevalent in practice.

1.4.3 Data Enrichment

Data enrichment is the process of using the data that we have to collect additional details or information from other sources about our subject matter, thus enriching the value of that data. In addition to parsing and structuring text, we can leverage the power of regular expressions in SAS to enrich data.

So, suppose we are going to do some economic impact analysis of the main SAS campus—located in Cary, NC—on the surrounding communities. In order to do this properly, we need to perform statistical analysis using geospatial information.

The address information is easily acquired from www.sas.com. However, it is useful, if not necessary, to include additional geo-location information such as latitude and longitude for effective analysis and reporting of geospatial statistics. The process of automating this is non-trivial, containing advanced programming steps that are beyond the scope of this book. However, it is important for you to understand that the techniques described in this book lead to just such sophisticated capabilities in the future. To make these techniques more tangible, we will walk through the steps and their results.

1. Start by extracting the address information embedded in Figure 1.2, just as in the data manipulation example, with regular expressions.

Figure 1.2: HTML Address Information

```
<p>World Headquarters<br>
SAS Institute Inc.<br>
100 SAS Campus Drive<br>
Cary, NC 27513-2414, USA<br>
Phone:919-677-8000<br>
Fax:919-677-4444<br>
</p>
```

Example Data Set Structure

Location	Address Line 1	Address Line 2	City	State	Zip	Phone	Fax
World Headquarters	SAS Institute Inc.	100 SAS Campus Drive	Cary	NC	27513-2414	919-677-8000	919-677-4444

6 Introduction to Regular Expressions in SAS

2. Submit the address for geocoding via a web service like Google or Yahoo for free processing of the address into latitude and longitude. Type the following string into your browser to obtain the XML output, which is also sampled in Figure 1.3.

<http://maps.googleapis.com/maps/api/geocode/xml?address=100+SAS+Campus+Drive,+Cary,+NC&sensor=false>

Figure 1.3: XML Geocoding Results

```
- <geometry>
  - <location>
    <lat>35.8301733</lat>
    <lng>-78.7664916</lng>
  </location>
  <location_type>ROOFTOP</location_type>
  - <viewport>
    - <southwest>
      <lat>35.8288243</lat>
      <lng>-78.7678406</lng>
    </southwest>
    - <northeast>
      <lat>35.8315223</lat>
      <lng>-78.7651426</lng>
    </northeast>
  </viewport>
</geometry>
```

3. Use regular expressions to parse the returned XML files for the desired information—latitude and longitude in our case—and add them to the data set.

Note: We are skipping some of the details as to how our particular set of latitude and longitude points are parsed. The tools needed to perform such work are covered later in the book. This example is provided here primarily to spark your imagination about what is possible with regular expressions.

Example Data Set Structure

Location	...	Latitude	Longitude
World Headquarters	...	35.8301733	-78.7664916

4. Verify your results by performing a reverse lookup of the latitude/longitude pair that we parsed out of the results file using <https://maps.google.com/>. As you can see in Figure 1.4, the expected result was achieved (SAS Campus Main Entrance in Cary, NC).

Figure 1.4: SAS Campus Using Google Maps



Now that we have an enriched data set that includes latitude and longitude, we can take the next steps for carrying out the economic impact analysis.

Hopefully, the preceding examples have proven motivating, and you are now ready to discover the power of regular expressions with SAS. And remember, the last example was quite advanced—some sophisticated SAS programming capabilities were needed to achieve the result end-to-end. However, the majority of the work leveraged regular expressions.

8 *Introduction to Regular Expressions in SAS*

¹ Wikipedia, http://en.wikipedia.org/wiki/Regular_expression#Formal_definition

² For more information on the version of Perl being used, refer to the artistic license statement on the SAS support site here: http://support.sas.com/rnd/base/datastep/perl_regex/regexp.compliance.html

³ This example file was obtained from data.gov here:
http://www.sec.gov/open/datasets/administrative_proceedings_2009.xml

Chapter 2: Getting Started with Regular Expressions

- 2.1 Introduction 10**
 - 2.1.1 RegEx Test Code..... 11
- 2.2 Special Characters..... 13**
- 2.3 Basic Metacharacters 15**
 - 2.3.1 Wildcard 15
 - 2.3.2 Word 15
 - 2.3.3 Non-word 16
 - 2.3.4 Tab 16
 - 2.3.5 Whitespace 17
 - 2.3.6 Non-whitespace 17
 - 2.3.7 Digit 17
 - 2.3.8 Non-digit..... 18
 - 2.3.9 Newline..... 18
 - 2.3.10 Bell 19
 - 2.3.11 Control Character..... 20
 - 2.3.12 Octal 20
 - 2.3.13 Hexadecimal 21
- 2.4 Character Classes..... 21**
 - 2.4.1 List 21
 - 2.4.2 Not List 22
 - 2.4.3 Range 22
- 2.5 Modifiers..... 23**
 - 2.5.1 Case Modifiers..... 23
 - 2.5.2 Repetition Modifiers..... 25
- 2.6 Options 32**
 - 2.6.1 Ignore Case..... 32
 - 2.6.2 Single Line..... 32
 - 2.6.3 Multiline 33
 - 2.6.4 Compile Once 33

2.6.5 Substitution Operator.....	34
2.7 Zero-width Metacharacters.....	34
2.7.1 Start of Line.....	35
2.7.2 End of Line	35
2.7.3 Word Boundary.....	35
2.7.4 Non-word Boundary.....	36
2.7.5 String Start.....	36
2.8 Summary.....	37

2.1 Introduction

This chapter focuses entirely on developing your understanding of regular expressions (RegEx) before getting into the details of using them in SAS. We will begin actually implementing RegEx with SAS in Chapter 3. It is a natural inclination to jump right into the SAS code behind all of this. However, RegEx patterns are fundamental to making the SAS coding elements useful. Without going through the RegEx first, the forthcoming SAS functions and calls could be discussed only at a very theoretical level, which is the opposite of what I am trying to accomplish in this book. Also, trying to learn too many different elements of any process at the same time can simply be overwhelming.

To facilitate the mission of this book—practical application—without becoming overwhelmed by too much information at one time (new functions, calls, and expressions), there is a very short bit of test code to use with the RegEx examples throughout the chapter. I want to stress the point that obtaining a thorough understanding of RegEx syntax is critical for harnessing the full power of this incredible capability in SAS.

RegEx consist of letters, numbers, metacharacters, and special characters, which form patterns. In order for SAS to properly interpret these patterns, all RegEx values must be encapsulated by delimiter pairs—I use the forward slash, /, throughout the text. (Refer to the test code). They act as the container for our patterns. So, all RegEx patterns that we create will look something like this: /pattern/.

For example, suppose we want to match the string of characters “Street” in an address. The pattern would look like /Street/. But we are clearly interested in doing more with RegEx than just searching for strings. So, the remainder of this chapter explores the various RegEx elements that we can insert into // to develop rich capabilities.

Metacharacter

Before going any farther, I should clarify some upcoming terminology. *Metacharacter* is a term used quite frequently in this book, so I need to be clear as to what it actually means. A metacharacter is a character or set of characters used by a programming language like SAS for something other than its literal meaning. For example, \s represents a whitespace character in RegEx

patterns, rather than just being a \ and the letter “s” collocated in the text. We begin our discussion of specific metacharacters in Section 2.3.

All nonliteral RegEx elements are some kind of metacharacter. It is good to keep this distinction clear, as I also make references to *character* when I want to discuss the actual string values or the results of metacharacter use.

Special Character

A *special character* is one of a limited set of ASCII characters that affects the structure and behavior of RegEx patterns. For example, opening and closing parentheses, (and), are used to create logical groups of characters or metacharacters in RegEx patterns. These are discussed thoroughly in Section 2.2.

RegEx Pattern Processing

At this juncture, it is also important to clarify how RegEx are processed by SAS. SAS reads each pattern from left to right in sequential *chunks*, matching each element (character or metacharacter) of the pattern in succession. If we want to match the string “hello”, SAS searches until the first match of the letter “h” is found. Then, SAS determines whether the letter “e” immediately follows, and so on until the entire string is found. Below is some pseudo code for this process, for which the logic is true even after we begin replacing characters with metacharacters (it would simply look more impressive).

Pseudo Code for Pattern Matching Process

```
START IF POS = "h" THEN POS+1 NEXT ELSE POS+1 GOTO START
IF POS = "e" THEN POS+1 NEXT ELSE POS+1 GOTO START
    IF POS = "l" THEN POS+1 NEXT ELSE POS+1 GOTO START
    IF POS = "l" THEN POS+1 NEXT ELSE POS+1 GOTO START
    IF POS = "o" THEN MATCH=TRUE GOTO END ELSE POS+1 GOTO START
END
```

In this pseudo code, we see the START tag is our initiation of the algorithm, and the END tag denotes the termination of the algorithm. Meanwhile, the NEXT tag tells us when to skip to the next line of pseudo code, and the GOTO tag tells us to jump to a specified line in the pseudo code. The POS tag denotes the character position. We also have the usual IF, THEN, and ELSE logical tags in the code.

Again, this example demonstrates the search for “hello” in some text source. The algorithm initiates by testing whether the first character position is an “h”. If it is not true, then the algorithm increments the character position by one—and tests for “h” again. If the first position is an “h”, the character position is incremented, and the code tests for the letter “e”. This continues until the word “hello” is found.

2.1.1 RegEx Test Code

The following code snippet enables you to quickly test new RegEx concepts as we go through the chapter. As you learn new RegEx metacharacters, options, and so on, you can edit this code in an effort to test the functionality. Also, more interesting data can be introduced by editing the `data_lines` portion of the code. However, because we haven’t yet discussed the details of how the pieces work, I discourage

12 Introduction to Regular Expressions in SAS

making edits outside the marked places in the code in order to avoid unforeseen errors arising at run time.

To keep things simple, we are using the DATALINES statement to define our data source and print the source string and the matched portion to the log. This should make it easier to follow what each new metacharacter is doing as we go through the text. Notice that everything is contained in a single DATA step, which does not generate a resulting data set (we are using `_NULL_`). The first line of our code is an IF statement that tests for the first record of our data set. The RegEx pattern is created only if we have encountered the first record in the data set, and is retained using the RETAIN statement. Afterward, the pattern reference identifier is reused by our code due to the RETAIN statement. Next, we pull in the data lines using the INPUT statement that assumes 50-character strings. Don't worry about the details of the CALL routine on the next line for now. We start writing SAS code in Chapter 3.

Essentially, the CALL routine inside the `RegEx Testing Framework` code shown below uses the RegEx pattern to find only the first matching occurrence of our pattern on each line of the `datalines` data. Finally, we use another IF statement to determine whether we found a pattern match. If we did, the code prints the results to the SAS log.

```
/*RegEx Testing Framework*/
data _NULL_;
if _N_=1 then
do;
    retain pattern_ID;
    pattern="/METACHARACTERS AND CHARACTERS GO HERE/"; /*<--Edit the pattern
here.*/
    pattern_ID=prxparse(pattern);
end;
input some_data $50.;
call prxsubstr(pattern_ID, some_data, position, length);
if position ^= 0 then
do;
    match=substr(some_data, position, length);
    put match:$QUOTE. "found in " some_data:$QUOTE.;
end;
datalines;
Smith, BOB A.
ROBERT Allen Smith
Smithe, Cindy
103    Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
;
run;
```

Note: I have provided a jumble of data in the `dataLines` portion of the code above. However, feel free to edit the data lines to thoroughly test each metacharacter as we go through this chapter.

Figure 2.1 shows an example of the SAS log output provided by the previous code. For this example, I used merely the character string `/Street/` for the pattern in order to create the output.

Figure 2.1: Example Output where, `pattern=/Street/`

```
"Street" found in "3000 K Street NW, Washington, DC 20007"
```

The remaining information in this chapter provides a solid foundation for building robust, complex patterns in the future. Each element discussed is an independently useful building block for sophisticated text manipulation and analysis capabilities. Once we begin to combine these basic elements, we will create some very powerful analytic tools.

2.2 Special Characters

In addition to `/` (the forward slash), the characters `()|` and `\` (the backslash) are special and are thus treated differently than the RegEx metacharacters to be discussed later. Since some of these special characters are so fundamental to the structure of the RegEx pattern construction, we need to briefly discuss them first.

`()`

The two parentheses create logical groups of pattern characters and metacharacters—the same way they work in SAS code for logic operations. It is important to create logical groupings in order to construct more sophisticated patterns. Nesting the parentheses is also possible.

`|`

The vertical bar represents a logical OR (much like in SAS). Again, the proper use of this element creates more sophisticated patterns. We will explore some interesting ways to use this character, starting with the example in Table 2.1. It is important to remember that the first item in an OR condition always matches before moving to the next condition.

`\`

The backslash is a tricky one as it has a couple of uses. It is used as an integral component of many other metacharacters (examples abound in Section 2.3). Think about it as an initiator that tells SAS, “Hey, this is a metacharacter, not just some letter.” But that’s not all it does. Since the special characters defined above also appear in text that we might want to process, the backslash also acts as a blocker that tells SAS, “Hey, treat this special character as just a regular character.” By using `\`, we can create patterns that include parentheses, vertical bars, backslashes, forward slashes, and more—we simply add a `\` in front of each occurrence of all the special characters that we want to treat as characters. For example, if we want our pattern to include open and closed parentheses respectively, the pattern would contain `\(\\)`.

Since you haven't learned any RegEx metacharacters yet, let's revisit strings using some of these new concepts. Notice that we can already start to match useful patterns with the characters and special characters.

**Table 2.1: Examples using (), |, and **

Usage	Matches
/C c at/	"Cat" "cat"
/cat mouse/	"cat" "mouse"
/((S s)tree)t((R r)oad)/	"Street" "street" "Road" "road"
^(This\\) (That\\) /	"(This)" or "(That)"

Note: In Perl parlance, \ is known as an *escape character*. To avoid any unnecessary confusion, we will dispense with this lingo and just refer to it as the backslash. However, be prepared to see that term used quite a bit in the Perl literature and on community websites.

Now, there are some additional special characters that also need the backslash in front of them in order to be matched as normal characters. They are: { } [] ^ \$. * + and ?. All these characters are reserved and are thus treated differently, because they each have a special purpose and meaning in the world of RegEx. Since each one is defined and discussed at length in Sections 2.4 and 2.5, we will not discuss them further here. For now, just remember that they can't be used as part of pattern strings without the backslash immediately preceding them. Table 2.2 shows a few examples of how to use them as normal characters.

Table 2.2: Examples using { } [] ^ \$. * +

Usage	Matches
^\$1\\.00 \\+ \\\$0\\.50 = \\\$1\\.50/	"\$1.00 + \$0.50 = \$1.50"
/2*3 = 6/	"2*3 = 6"
^[2\\]^2/	"[2]^2"
^\\{1,2,3,4,5\\}/	"{1,2,3,4,5}"

Note: Notice that = and , match as characters (i.e., without a backslash) because they are not considered special characters.

2.3 Basic Metacharacters

As you write RegEx patterns in the future, you will find yourself using most of the metacharacters discussed in this section frequently because they are fundamental elements of RegEx pattern creation. Now, we can already build some useful patterns with the information discussed in Section 2.1. However,

the metacharacters in this section create the greatest return on time investment due to how flexible and powerful they can make RegEx patterns.

Notice as we go through the examples how we can obtain some unexpected results. It is important to be very strategic when using some of these RegEx metacharacters as you don't always know what to expect in the text that you are processing. Even when you know the source quite well, there are inevitably errors or unknown changes that can wreck a poorly designed pattern. So, like any good analyst, you need to be thinking a few steps ahead in order to maintain robust RegEx code.

Note: Unlike SAS, all RegEx metacharacters are case sensitive, as you will see shortly. If a letter is defined here as lowercase or uppercase, then it **MUST** be used that way. Otherwise, your programs will do something very different from what you expect. In other words, even though you can be lazy with capitalization when writing SAS code (e.g., DATA vs. data), the same is not true here.

2.3.1 Wildcard

The wildcard metacharacter, which is a period (`.`), matches any single character value, except for a newline character (`\n`). The ability to match virtually any single character will prove useful when you are searching for the superset of associated character strings. You might also want to use it when you have no idea what values might be in a particular character position. Table 2.3 provides examples.

Table 2.3: Examples using `.`

Usage	Matches
<code>/R.n/</code>	“Ran” “Run” “R+n” “R n” “R(n)” “Ron” ...
<code>/.un/</code>	“Fun” “fun” “Run” “run” “bun” “(un)” “-un” ...
<code>/Street./</code>	“Street.” “Street,” “Streets” “Street+” “Street_” ...

Note: The period matches anything except the newline character (`\n`)—including itself. This can be helpful, but must be used wisely. Also note, only `\n` matches the newline character.

2.3.2 Word

The metacharacter `\w` matches any word character value, which includes alphanumeric and underscore (`_`) values. It matches any single letter (regardless of case), number, or underscore for a single character position. But do not be fooled by the underscore inclusion; `\w` does NOT match hyphens, dashes, spaces, or punctuation marks. Table 2.4 provides examples.

Table 2.4: Examples using \w

Usage	Matches
/R\wn/	“Ran” “Run” “Ron” ...
^wun/	“Fun” “fun” “Run” “run” “Bun” “bun” “_un” ...
/Street\w/	“Streets” “Street_”

Note: The \w wildcard should not have any unintentional spaces before or after it. Such spaces result in the pattern trying to match those additional spaces in addition to the \w. (This goes for any RegEx metacharacter.)

2.3.3 Non-word

The metacharacter \W matches a non-word character value (i.e., everything that \w doesn't include, except for the ever-elusive \n). The \W metacharacter is valuable when you are unsure what is in a character cell but you know that you don't want a word character (i.e., alphanumeric and _). Table 2.5 provides examples.

Table 2.5: Examples using \W

Usage	Matches
/Washington\W/	“Washington.” “Washington,” “Washington;” ...
/D\WC\W/	“D.C.” “D,C.” “D C.” “D C” ...
/Street\W/	“Street.” “Street,” “Street+” ...

Note: You will continue to see lowercase and uppercase versions of these RegEx characters acting as near opposites, with some exceptions. It might not be overly clever, but does help simplify matters.

2.3.4 Tab

The metacharacter \t matches only the tab character in a string. Unlike the RegEx characters to follow, this metacharacter matches only the tab whitespace character. This is especially useful when the tab holds some special significance, such as when you are processing tab-delimited text files. Table 2.6 provides examples.

Table 2.6: Examples using \t

Usage	Matches
/SAS\t/	“SAS ”
/SAS\tInstitute\tInc/	“SAS Institute Inc”
/Street\t/	“Street ”

Note: This metacharacter does not have an opposite (i.e., `\T` does not exist).

2.3.5 Whitespace

The metacharacter `\s` matches on a single whitespace character, which includes the space, tab, newline, carriage return, and form feed characters. You must include this when you are matching on anything in text that is separated by white space, and you are unsure of which will occur. Table 2.7 provides examples.

Table 2.7: Examples using `\s`

Usage	Matches
<code>/SAS\s/</code>	“SAS ” “SAS ”
<code>/SAS\sInstitute\sInc/</code>	“SAS Institute Inc” “SAS Institute Inc”
<code>/Street\s/</code>	“Street ” “Street ”

Note: This form of the `\s` metacharacter matches only one whitespace character. We review how to find multiple matches in Section 2.5.2 because that is frequently needed when you are matching text.

2.3.6 Non-whitespace

The metacharacter `\S` matches on a single non-whitespace character—the exact opposite of `\s`. This metacharacter is often used to account for unexpected dashes, apostrophes, commas, and so on, that might otherwise prevent a match. Table 2.8 provides examples.

Table 2.8: Examples using `\S`

Usage	Matches
<code>/Leonato\Ss/</code>	“Leonato’s” “Leonatoas” “Leonato_s” ...
<code>/Washington\S/</code>	“Washingtons” “Washington.” “Washington,” ...
<code>/Street\S/</code>	“Street.” “Street,” “Streets” “Street+” “Street_” ...

2.3.7 Digit

The metacharacter `\d` matches on a numerical digit character (i.e., 0–9). This RegEx metacharacter is probably the most straightforward one as it has a very narrow focus. Just remember that a single occurrence of `\d` is for only one character position in any text. In order to capture larger numbers (i.e., anything greater than 9), you have to build patterns with multiple occurrences of `\d`. Table 2.9 provides examples, but we discuss more sophisticated methods for accomplishing this later in the chapter. (See “Repetition Modifiers” in Section 2.4.2).

Table 2.9: Examples using \d

Usage	Matches
/\dst/	“1st” “9st” “4st” ...
/10\d/	“101” “102” “103” ...
/1-800-\d\d\d-\d\d\d\d/	“1-800-123-4567” “1-800-789-3456” ...

Note: Just remember that even though your pattern might be correct, the data is not necessarily correct (4st and 9st don’t make sense!).

2.3.8 Non-digit

The metacharacter \D matches on any single non-digit character. Again, this is the opposite of the lowercase metacharacter \d. This metacharacter matches on every value that is not a number. Table 2.10 provides examples.

Table 2.10: Examples using \D

Usage	Matches
/1\D800\D123\D4567/	“1-800-123-4567” “1.800.123.4567” ...
/1560\DWilson\DBlvd/	“1560 Wilson Blvd” “1560_Wilson_Blvd” ...
/19\D\DStreet/	“19 th Street” “19 th .Street” “19...Street” ...

2.3.9 Newline

The metacharacter \n matches a newline character. It is quite useful for some patterns to know that you have encountered a new line. For instance, you might be processing addresses in a text file, which often contain different pieces of information on different lines. Table 2.11 provides examples.

Table 2.11: Examples using \n

Usage	Matches
/103 Pennsylvania Ave\. NW,\nWashington, DC 20216/	“103 Pennsylvania Ave. NW, Washington, DC 20216”
/<<html tag>\n/	“<html tag> ” ...
/v\nenr\nt\ni\nc\na\nl\nt\ne\nx\nt/	“v e r t

Usage	Matches
	i
	c
	a
	l
	t
	e
	x
	t” ...

Note: The test code does not enable us to actually try this metacharacter because it uses data lines, which is a feature of SAS that intentionally ignores newline characters when typed (i.e., hitting the Enter key just creates the start of a new data line in the SAS code window). For this reason, newline characters are not present in data lines for you to read and match on. But have faith, for now, that this one works as advertised. You will discover ways to process different text sources in the next chapter, enabling you to process newline characters.

2.3.10 Bell

The metacharacter `\a` matches an alarm “bell” character. The alarm character falls into a class of non-printing or invisible characters that are part of the ASCII character set. ASCII was developed long ago when operating systems used non-printing characters fairly extensively. Today, however, these characters are relatively uncommon, and most often occur only in files meant for computers to read rather than humans—since they are not displayed. When encountered, these characters generate an alarm tone, or “bell,” on a computer’s internal speaker. While they are often associated with errors, they can also be used to alert users that the end of a file or process has been achieved (e.g., in a system log file). You can use this metacharacter when you know to expect such a character in a source file. Table 2.12 provides examples.

Table 2.12: Examples using `\a`

Usage	Matches
<code>^a END OF FILE/</code>	“ BEL END OF FILE”
<code>/PROCESS COMPLETED SUCCESSFULLY\a/</code>	“PROCESS COMPLETED SUCCESSFULLY BEL ” ...
<code>^aERROR/</code>	“ BEL ERROR” ...

Note: Since the alarm character is a non-printing ASCII character, I am representing its location in the matching text with the BEL ASCII character. However, remember that such a code does not appear in our text.

2.3.11 Control Character

The metacharacter `\cA-\cZ` matches a control character for the letter that follows the `\c`. For example, `\cF` matches control-F in the source. This is one of several examples where you might be processing less-often-used file types (i.e., not a file meant for humans to read). Control characters, or non-printing characters, were once used extensively by transactional computing and telecommunications systems. These control characters, while not visible in most text editors, are still part of the ASCII character set, and can still be used by older systems in these regimes. For our examples in Table 2.13, we stick with the convention that is used for the alarm metacharacter above—the standard ASCII abbreviation is used despite the fact that they are never actually seen in text.

Table 2.13: Examples using `\cA-\cZ`

Usage	Matches
<code>^cP/</code>	DEL the non-printing Data Link Escape ASCII control character <code>^P</code>
<code>^cB/</code>	STX the non-printing Start of Text ASCII control character <code>^B</code>
<code>^cBhello^cC/</code>	STX hello ETX the non-printing Start of Text ASCII control character <code>^B</code> followed by the character string “hello” and completed with the non-printing End of Text ASCII control character <code>^C</code>

2.3.12 Octal

The metacharacter `\ddd` matches an octal¹ character of the form *ddd*. It is used to match on the octal code for an ASCII character for which you are searching. It can be especially useful when you need to find specific non-printing ASCII characters in a file. The default behavior by SAS is to return the ASCII character associated with this octal code in the results. Table 2.14 provides examples.

Table 2.14: Examples using `\ddd`

Usage	Matches	Notes
<code>^s\041\s/</code>	“ ! ”	This octal code translates to the ! ASCII character.
<code>^110\105\114\114\117/</code>	“HELLO”	This series of octal codes translate to the “HELLO” string of ASCII characters.
<code>^s\007\011\s/</code>	“ BELTAB ”	These octal codes translate to the two non-printing ASCII characters BEL and TAB . Refer to our discussion of the alarm metacharacter in Section 2.3.10 regarding characters that are not displayed.

Note: You will discover how to search for ranges of these values in the next section (Section 2.4). Also note that the largest ASCII value is decimal 127, octal 177, and hexadecimal 7F.

2.3.13 Hexadecimal

The metacharacter `\xdd` matches a hexadecimal² character of the form *dd*. The purpose of our implementation here is again not about searching through raw hexadecimal files, etc. We are using this to search for the hexadecimal code associated with the ASCII characters that we want in a source (manipulation of raw hex data sources is a different book). Table 2.15 provides examples.

Table 2.15: Examples using `\xdd`

Usage	Matches	Notes
<code>^x2B/</code>	“+”	This hexadecimal code translates to the + ASCII character.
<code>^x31\x2B\x31\x3D\x32/</code>	“1+1=2”	These hexadecimal codes translate to the 1+1=2 ASCII characters.
<code>^x30\x30\x20\x46\x46/</code>	“00 FF”	This is a reminder that we can match hexadecimal numbers stored in ASCII, and that they are not the same.

2.4 Character Classes

In addition to using the built-in RegEx characters to match patterns, users have the ability to create custom character matching. This capability is derived via different uses of [and] (square braces). The square braces essentially create a custom metacharacter, where the items contained between the opening brace and closing brace are possible match values for a single character cell. In addition to putting a list characters inside the braces, you can also include metacharacters. Each metacharacter discussed below includes an example, which includes the use of a metacharacter, and they all have the same match results. Just for fun, they are all identifying a hexadecimal number range present in the ASCII source file (stored as ASCII characters in the source file, but representing the range of possible hexadecimal values).

Note: Remember that some of the components discussed in this section are special characters that must be escaped with `\` in order to be matched in isolation. Specifically, these characters are: `^`, `[`, and `]`.

2.4.1 List

The metacharacter `[...]` matches any one of the specific characters or metacharacters listed within the braces. Being able to define an unordered list of things that you want to appear in a space is very convenient, and can sometimes be more convenient than the metacharacters that identify broad classes of character types. Table 2.16 provides examples.

Table 2.16: Examples using [...]

Usage	Matches
/[abcABC]/	“a” “b” “c” “A” “B” “C”
/[0173]/	“0” “1” “3” “7”
/[CcBbRr]at/	“cat” “Cat” “bat” “Bat” “rat” “Rat”
/[dABCDEF]/	“0” “1” “2” “3” “4” “5” “6” “7” “8” “9” “A” “B” “C” “D” “E” “F”

2.4.2 Not List

The metacharacter `[^...]` matches one of anything not listed within the braces, except for the newline character. Sometimes it is easier to write down what we don't want rather than what we do. And for that reason, we might want to use this metacharacter. We can quickly identify the unwanted items and define them here. Table 2.17 provides examples.

Table 2.17: Examples using [^...]

Usage	Matches
/[^abcABC]/	“d” “e” “f” ...
/[^0173]/	“2” “4” “5” “6” “8” “9”
/[^Cc]at/	“fat” “Fat” “hat” “rat” “mat” “Hat” ...
/[^WGH IJKLMN O P Q R S T U V W X Y Z abcdefghijklmnopqrstuvwxyz_]/	“0” “1” “2” “3” “4” “5” “6” “7” “8” “9” “A” “B” “C” “D” “E” “F”

2.4.3 Range

The metacharacter `[...-...]` matches anything that falls into a range of character values. In other words, case matters for letters listed in the braces. RegEx, and by extension SAS, understands the inherent order of letters and numbers. Therefore, we can define any range of numbers or letters to be matched by this metacharacter. Table 2.18 provides examples.

Table 2.18: Examples using [...-...]

Usage	Matches
/[f-m]/	“f” “g” “h” “i” “j” “k” “l” “m”
/[1-9]/	“1” “2” “3” “4” “5” “6” “7” “8” “9”
/[a-cA-C]/	“a” “b” “c” “A” “B” “C”
/[dA-F]/	“0” “1” “2” “3” “4” “5” “6” “7” “8” “9” “A” “B” “C” “D” “E” “F”

2.5 Modifiers

There are two significant things that you probably notice missing from the previous sections, which are worth further discussion here. First, all of the applicable metacharacters thus far have ignored letter case. In other words, `\w`, `\S`, `\D`, and `.` all match on a letter regardless of whether it is lowercase or uppercase. However, there are situations in which the case of a letter becomes important, but the letter itself is not known in advance.

Second, we can use a single match character as many times as we like, which creates additional fuzziness for our matches. However, there is a downside to just typing them out: *each occurrence must exist in order to match the pattern*. For instance, if the source text for the `\D` examples above contained “19thStreet” with no spaces, we’d never find it by using `\D` three times. And since the primary goal of the RegEx capability is to have automated text processing, we need a robust way to make this kind of matching more flexible.

Over the next two subsections (2.5.1 and 2.5.2), we will work through ways to overcome these limitations by using modifiers. There are two types of modifiers, case modifiers and repetition modifiers. Combining them gives us significant robustness and flexibility in real-world RegEx implementations, and should be considered as fundamental to real-world implementations as the metacharacters that we have discussed thus far.

2.5.1 Case Modifiers

When performing matches on text, there is the obvious consideration of letter case (upper vs. lower). Although I have already introduced a rudimentary way to handle this in situations where the letter is known, there still must be a methodology for accounting for letter case when it is unknown. This section discusses a variety of approaches to dealing with case matching. Depending on the situation, some approaches are more convenient than others, while not necessarily being right or wrong.

Lowercase

The metacharacter `\l` matches when the next character in a pattern is lowercase. This metacharacter applies only to characters (metacharacters, groups, and so on don’t work). In practice, it is more practical to simply type the lowercase version of the desired character value, or provide a list of lowercase letters to match. Table 2.19 provides examples.

Table 2.19: Examples using `\l`

Usage	Matches
<code>/\lStreet/</code>	“street” ...
<code>/s\lS\lA\lS\sInstitute/</code>	“ sas Institute” ...
<code>/(\lS\lF)leet/</code>	“sleet” “fleet” ...

Uppercase

The metacharacter `\u` matches when the next letter in a pattern is uppercase. It functions exactly as the lowercase version introduced above (`\l`), but also applies to uppercase. Table 2.20 provides examples.

Table 2.20: Examples using `\u`

Usage	Matches
<code>^\uinc./</code>	“Inc.” ...
<code>^\ustreet \ust\./</code>	“Street” “St.” ...
<code>^\uave\.\ uavenue./</code>	“Ave.” “Avenue,”

Lowercase Range

The metacharacter `\L...\E` matches when all the characters between the `\L` and `\E` are lowercase. Strings typed between `\L` and `\E` are forced to match on lowercase only, even when they are typed in as capital letters. However, unlike the `\l` metacharacter, `\L...\E` can also contain character classes and repetition modifiers. Table 2.21 provides examples.

Table 2.21: Examples using `\L...\E`

Usage	Matches
<code>^\L[a-z0-9][a-z0-9][a-z0-9]\E/</code>	“sas” “abc” “123” ...
<code>^\LTHESE ARE LOWERCASE\E/</code>	“these are lowercase”
<code>^\sR^\L[a-z][a-z][a-z]\E\s/</code>	“ Read ” “ Road ” “ Rode ” “ Ride ” “ Real ” ...

Note: When applying case modifiers to non-alphabet characters, the modifier is ignored. It doesn’t apply to those characters, so it doesn’t affect the match.

Uppercase Range

The metacharacter `\U...\E` creates a match when all the characters between the `\U` and `\E` are uppercase. Again, this metacharacter functions the same way as the lowercase version discussed above, but applies to uppercase. This metacharacter can be useful for identifying acronyms or other text where capital letters are important. Table 2.22 provides examples.

Table 2.22: Examples using `\U...\E`

Usage	Matches
<code>^\U[a-z][a-z][a-z]\E/</code>	“SAS” “CIA” ...
<code>^\U[a-z][a-z][a-z]\E\sInstitute\sInc\W/</code>	“SAS Institute Inc.” ...
<code>^\s\Uallcaps\E\s/</code>	“ ALLCAPS ”

Note: Notice that other metacharacters are not allowed inside `\L...\E` or `\U...\E` metacharacters. In other words, `\w` can't be used to replace the character classes above.

Quote Range

The metacharacter `\Q...\E` matches all content inside the `\Q` and `\E` as character strings, disabling everything including the backslash character. Metacharacters cannot be used inside `\Q...\E`. The functionality provided by this metacharacter is great for searching within strings that contain a significant number of reserved characters, such as XML, webserver logs, or HTML. Table 2.23 provides examples.

Table 2.23: Examples using `\Q...\E`

Usage	Matches
<code>^Q<html tag name>\E/</code>	<code>"<html tag name>"</code>
<code>^Qf(x) + f(y) = z\E/</code>	<code>"f(x) + f(y) = z"</code>
<code>^Q<!DOCTYPE HTML> <html lang="en-US">\E/</code>	<code>"<!DOCTYPE HTML> <html lang="en-US">"</code>

2.5.2 Repetition Modifiers

Repetition modifiers change the matching repetition behavior of the metacharacters and characters immediately preceding them in a pattern. They can also modify the matching repetition of an entire group—defined using `()` to surround the group of metacharacters and characters before the modifier. Just keep in mind that repetition of the entire group means that it repeats back-to-back (e.g., “haha”), unless we also modify the individual metacharacters.

Now, there are two types of repetition modifiers, *greedy* and *lazy*. Greedy repetition modifiers try to match as many times as possible within the confines of their definition. Lazy modifiers attempt to find a match as few times as possible. They have similar uses, which can make the difference between their results subtle.

Introduction to Greedy Repetition Modifiers

Let's start by discussing greedy modifiers because they are a little more intuitive to use. As we go through the examples, it is important to keep in mind that greedy modifiers match as many times as possible—constantly searching for the last possible time the match is still true. It is therefore easy to create patterns that match differently from what you might expect.

There is a concept in RegEx known as *backtracking*, which is the root cause for potential issues with greedy modifiers (hint: backtracking results in the need for lazy modifiers). As we discuss further when we examine lazy repetition modifiers, a greedy modifier actually tries to maximize the matches of a modified pattern chunk by searching until the match fails. Upon that failure, the system then *backtracks* to the position where the modified chunk last matched. The processing time wasted with backtracking for a single match is insignificant. However, as soon as we introduce a few additional factors, this problem can waste tremendous computing cycles—multiple modified pattern chunks, numerous match

iterations (think loops), and large data sources. It is important to be mindful of these factors when designing patterns as they can have unintended consequences.

Greedy 0 or More

The modifier `*` requires the immediately preceding character or metacharacter to match 0 or more times. It enables us to generate unlimited optional matches within text. For example, we might want to match every occurrence of a word root, along with all of its prefixes and suffixes. By allowing the prefixes and suffixes to be optional, we are able to achieve this goal. Table 2.24 provides examples.

Table 2.24: Examples using `*`

Usage	Matches
<code>/Sing\w*/</code>	“Sing” “Sings” “Singing” “Singer” “Singers” ...
<code>/D\W*C\W*/</code>	“DC” “D.C.” “D C” “D...!\$%^ C-)*&^%”...
<code>/19\D*Street/</code>	“19 th Street” “19 th Street” “19Street” ...
<code>/Hello*/</code>	“Hell” “Hello” “Helloooooooooooooo” ...

Greedy 1 or More

The modifier `+` requires the immediately preceding character or metacharacter to match 1 or more times. The plus sign modifier works similarly to the asterisk modifier, with the exception that it enforces a match of the metacharacter or character at least 1 time. Table 2.25 provides examples.

Table 2.25: Examples using `+`

Usage	Matches
<code>/Ru\w+/</code>	“Run” “Ruin” “Runt” “Runners” ...
<code>^s\U[a-z]+\E\s/</code>	Words with all letters capitalized, and surrounded by spaces.
<code>/19\D+Street/</code>	“19 th Street” “19 th .Street” “19...Street” ...
<code>/(ha)+/</code>	“ha” “hahahahahahaha” ...

Note: Pay special attention to the addition of the `\s` metacharacter in the second example in Table 2.24. If it were not present, the pattern would also match only single capital letters at the beginning of words.

By adding `\s`, the pattern requires a whitespace character to immediately follow the one or more capital letters, thus eliminating matches on single letters at the beginning of words.

Greedy 0 or 1 Time

The modifier `?` creates a match of only 0 or 1 time. The question mark provides us the ability to make the occurrence of a metacharacter optional without allowing it to match multiple times. This can be effective

for matching word pairs that have inconsistent use of dashes or spaces (e.g., short-term vs. short term). Table 2.26 provides examples.

Table 2.26: Examples using ?

Usage	Matches
/1\D?800\D?123\D?4567/	“1-800-123-4567” “18001234567” ...
/1560\sWilson\sBlvd\W?/	“1560 Wilson Blvd.” “1560 Wilson Blvd” ...
/19 th \s?Street/	“19 th Street” “19 th Street” ...

Greedy n Times

The modifier {*n*} creates a match of exactly *n* times. Being able to match on a metacharacter exactly *n* number of times is the same as typing that metacharacter out that many times. However, from the perspective of coding and maintaining the RegEx patterns, using the modifier is a much better approach. It limits the opportunity for us to make typographical errors when initially creating the RegEx pattern, and it improves readability when later editing and sharing the patterns. Table 2.27 provides examples.

Table 2.27: Examples using {*n*}

Usage	Matches
/1-800-\d{3}-\d{4}/	“1-800-123-4567” “1.800.123.4567” ...
/R\w{4}/	“Round” “Runts” “Ruins” ...
/19\D{3}Street/	“19 th Street” “19 th .Street” “19...Street” ...
/(\d{5}-\d{4})+(\d{5})/	“12345-6789” “12345” ...

Greedy *n* or More

The modifier {*n*,} creates a match at least *n* times. By ensuring that we can match something at least *n* times, we are able to create functionality very similar to the plus modifier. However, we are raising the minimum number of times that the metacharacter must match. This is quite useful for certain applications, but must be handled with caution. Also, like the + modifier, we can easily get very long strings of unanticipated matches due to a single logical error in pattern construction. Table 2.28 provides examples.

Table 2.28: Examples using {*n*,}

Usage	Matches
/1-800-\d{1,}-\d{2,}/	“1-800-123-4567” “1-800-789-12” ...
/\d{3,}-\d{2,}-\d{4,}/	“143-25-7689” “12345689-546545654-9820” ...
/19\D{3,}Street/	“19 th Street” “19 th , Not My Street” ...

Note: Be mindful to not type a space after the comma inside the curly braces. It is easy to do out of habit, but it will wreck our pattern!

Greedy n to m Times

The modifier $\{n,m\}$ creates a match at least n , but not more than m times. Creating a match with a specified range is quite useful for ensuring that data quality standards are being maintained. When extracting semi-structured data elements such as ZIP codes, birthdates, and phone numbers, it is important to maintain a certain level of flexibility while also ensuring that the source is within expected tolerances. For instance, a two-digit year might be accepted in lieu of a four-digit year, but a four-digit zip would be unacceptable. Table 2.29 provides examples.

Table 2.29: Examples using $\{n,m\}$

Usage	Matches
<code>/(1-)?8\d\d-\d{3,3}-\d{4,4}/</code>	“1-800-123-4567” ...
<code>^\d{1,2}-\d{1,2}-\d{2,4}/</code>	“10-20-1950” “8-30-52” “4-3-1979” ...
<code>/Was{1,7}/</code>	“Washington” “Wash” “Waste” “Washing” ...

Note: As you can see in the examples above, the $\{n,m\}$ might not always be the best choice of modifier, but these examples are meant to demonstrate the flexibility of implementation. For instance, the year in the second example is allowed to be three digits with this usage. Using an OR clause with the $\{n\}$ modifier is a simple fix.

Introduction to Lazy Repetition Modifiers

Now that you are familiar with greedy modifiers, let’s begin examining the lazy ones. In terms of syntax, they differ from the greedy modifiers only by the addition of a question mark (?). By adding the question mark immediately after each of the greedy modifiers, we are able to subtly change their behavior—sometimes in unexpected ways.

In general, lazy modifiers are used to both avoid overmatching and improve performance when compared to the greedy modifiers. There are situations when matching with greedy modifiers would lead to either grabbing too much information, or simply slowing down system performance. For instance, processing semi-structured text files such as HTML or XML is a great example of when lazy modifiers would come in handy.

Lazy 0 or More

The modifier $*?$ creates a match 0 or more times, but as few times as necessary to create the match. In some situations, it creates the same matches as does the greedy version. However, in other cases, the results are very different. To make it clearer, Table 2.30 describes the details of a few examples.

Table 2.30: Examples using *?

Usage	Matches	Notes
/Sing\w*?/	“Sing”	This matches only the word “Sing” because the modifier is given the option to match nothing. And since it is <i>lazy</i> , it will take that option every time, regardless of whether a word character immediately follows the “g” in “Sing”.
/Sing\w*?\s/	“Singing ” ...	Comparing this to the example above, you see that appending the \s on the pattern creates additional matches. The \s forces the pattern to continue searching for a match that includes white space. This could be “Sing “ or many other combinations (similar to the greedy outcomes).
/(ha)*?/	“”	This example demonstrates why we need to be careful with lazy modifiers. Even when “ha” exists, it is ignored, again because the modifier has the option to do so. The greedy version of this would match as many times as the word “ha” occurred back-to-back, with a minimum of zero times.

Lazy 1 or More

The modifier +? creates a match 1 or more times, but as few times as necessary to create a match. Again, if it is possible, this matches only once. Table 2.31 provides examples.

Table 2.31: Examples using +?

Usage	Matches	Notes
/Sing\w+?/	“Singi”	This matches only “Sing” plus exactly one word character following the “g”. Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
/Sing\w+?\s/	“Singing ” ...	Again, we see that appending the \s on the pattern creates additional matches. The \s forces the pattern to continue searching for a match that includes white space. This could be “Singi “ or many other combinations (similar to the greedy outcomes).
/(ha)+?/	“ha”	This example is less of a cautionary tale than for *?. But it might still provide undesirable results. Even when “ha” exists numerous times back-to-back, it matches only the first time, unless an additional match element follows it. Again, this is because the modifier has the option to match only once. The greedy version of this would match as many times as the word “ha” occurred back-to-back, with a minimum of once.

Lazy 0 or 1 Times

The modifier `??` creates a match 0 or 1 times, but as few times as necessary to create a match. Unless forced, this modifier will match 0 times. Table 2.32 provides examples.

Table 2.32: Examples using `??`

Usage	Matches	Notes
<code>/Sing\w??/</code>	“Sing”	This matches only the word “Sing” because the modifier is given the option to match nothing. And since it is <i>lazy</i> , it will take that option every time, regardless of whether a word character immediately follows the “g” in “Sing”. The reasoning is the same as with the <code>*?</code> modifier.
<code>/Sing\w??\s/</code>	“Sings ” ...	Again, just as with the <code>*?</code> modifier, we see that appending the <code>\s</code> on the pattern creates additional matches. The <code>\s</code> forces the pattern to continue searching for a match that includes white space. This could be “Sings “ or a few other combinations (similar to the greedy outcomes).
<code>/(ha)??</code>	“”	This example demonstrates why we need to be careful with lazy modifiers. Even when “ha” exists, it is ignored, again because the modifier has the option to do so. The greedy version of this would match as many times as the word “ha” occurred back-to-back.

Lazy n Times

The modifier `{n}?` creates a match exactly n times. This modifier functions exactly as the greedy version, making the `?` unnecessary. Using this modifier results in no performance enhancement or change in functionality, which makes it a completely unnecessary addition to the Perl language. It has been included here for the sake of completeness. Table 2.33 shows that the same examples reveal the same results.

Table 2.33: Examples using “`{n}?`”

Usage	Matches
<code>/1-800-\d{3}?\-\d{4}/</code>	“1-800-123-4567” “1.800.123.4567” ...
<code>/R\w{4}?</code>	“Round” “Runts” “Ruins” ...
<code>/19\D{3}?Street/</code>	“19 th Street” “19 th .Street” “19...Street” ...
<code>/(\d{5}?\-\d{4}?)?(\d{5}?)?</code>	“12345-6789” “12345” ...

Lazy n or More

The modifier `{n,}?` creates a match, at least n times and as few times as necessary to create a match. This functions just like the `*?` or `+?` modifiers, except that the minimum number of matches is arbitrary. Again, we see similar behavior resulting from the laziness of the modifier. Table 2.34 provides examples.

Table 2.34: Examples using `{n,}?`

Usage	Matches	Notes
<code>/Sing\w{3,}?/</code>	“Singing” ...	This usage matches exactly $n=3$ times. Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
<code>/0{3,}? \s/</code>	“0000 ” ...	Now that you have the hang of these modifiers, this example should be a little more interesting. Appending <code>\s</code> on the pattern still forces it to match each 0 until the white space is encountered. The pattern is “anchored” to the first occurrence of a 0, thus capturing more than the minimum.
<code>/(ha){4,}?/</code>	“hahahaha”	Without surrounding information in the pattern, this matches only the minimum number of times. By having nothing else to force additional matching, the lazy modifier just stops after the minimum of $n=4$.

Lazy n to m Times

The modifier `{n,m}?` creates a match at least n times, but no more than m times—as few times in that range as necessary to create the match. It functions like many of the other lazy modifiers discussed thus far, but it sets a cap on how many times it can match in addition to having an arbitrary minimum. Table 2.35 provides examples.

Table 2.35: Examples using `{n,m}?`

Usage	Matches	Notes
<code>/Read\w{1,3}?/</code>	“Ready” ...	This usage matches the word metacharacter only one time. Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
<code>/0{2,5}? \s/</code>	“0000 ” ...	Again, the pattern is “anchored” to the first occurrence of a 0, thus capturing the minimum if it exists, up to the maximum.
<code>^sha(ha){0,6}?/</code>	“ ha”	By not having anything after the “anchor” point for the pattern to match on, there is nothing to force additional matching. The lazy modifier just stops after the minimum of $n=0$.

2.6 Options

Options affect the behavior of the entire RegEx pattern with which they are associated. These behavioral changes provide benefits ranging from making RegEx creation more convenient, to providing new or enhanced functionality.

Options occur *after* the closing slash character, but there is one item of significance that occurs *before* the first slash character that we will also discuss—it is not actually an option but this is best place to go over it. And we are not going to cover all of the options for the same reason we haven't covered absolutely all of the metacharacters thus far—this is an introductory text.

2.6.1 Ignore Case

The option `//i` ignores letter case for the entire pattern, even character strings. This is a great option to use when we know exactly what words we are searching for, but we don't want the letter case to be an issue. Table 2.36 provides examples.

Table 2.36: Examples using `//i`

Usage	Matches
<code>/1600 Pennsylvania Avenue/i</code>	“1600 pennsylvania avenue” “1600 PENNSYLVANIA AVENUE” ...
<code>/STREET/i</code>	“street” “Street” “STREET” ...
<code>/CAPS don't MaTtEr/i</code>	“caps don't matter” “CaPs DoN't MATTER” ...

2.6.2 Single Line

The option `//s` forces the dot character (`.`) to match everything, including the newline character, when it occurs in the pattern. This can be very helpful to ensure that we don't miss anything for a particular character position. Table 2.37 provides examples.

Table 2.37: Examples using `//s`

Usage	Matches
<code>/43rd and Times Square.New York, NY 10036/s</code>	“43 rd and Times Square New York, NY 10036” ...
<code>/Bob Smith.\d{3}-\d{3}-\d{4}/s</code>	“Bob Smith 123-456-7891” ...

2.6.3 Multiline

The option `//m` causes `^` and `$` to match on more than just the string start and end respectively. Instead, they match on every newline encountered because the various lines of information are treated as one continuous line. This enhanced functionality really applies to two metacharacters that we haven't covered yet (we'll discuss them in Section 2.7), so if you need to, feel free to peek ahead and come back to this one. Table 2.38 provides examples.

Table 2.38: Examples using `//m`

Usage	Matches
<code>^\w+/m</code>	Words at the beginning of a string and words following a newline character.
<code>^\w+?\s\$/m</code>	Words immediately before a space and the string end, and before a space and newline character.

2.6.4 Compile Once

The option `//o` is known as the *compile once* option. By having the “o” immediately following the closing slash, SAS knows to compile that RegEx only once. This option creates a very nice simplification to SAS code, which I demonstrate by showing updated test code below (see Section 2.1.1 for the original code). Notice how the IF block is removed, and only the two lines that do not include the RETAIN statement remain. These changes are possible due to the compilation happening the first time through the DATA step. Every subsequent loop through reuses the previously compiled expression, if it exists.

Updated Test Code

```
/*RegEx Testing Framework*/
data _NULL_;
*if _N_=1 then
*do;
*   retain pattern_ID;
*   pattern="/Run/"; /*<--Edit the pattern here.*/
*   pattern_ID=prxparse(pattern);
*end;
pattern="/Run/o"; /*<--Edit the pattern here.*/
pattern_ID=prxparse(pattern);
input some_data $50.;
call prxsubstr(pattern_ID, some_data, position, length);
if position ^= 0 then
do;
match=substr(some_data, position, length);
put match:$QUOTE. "found in " some_data:$QUOTE.;
end;
datalines;
Smith, BOB A.
```

```

ROBERT Allen Smith
Smithe, Cindy
103 Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
;
run;

```

2.6.5 Substitution Operator

While the substitution operator `s//` is not technically an option, it belongs here if only because it truly stands apart from the other items discussed in this section. Although the substitution operation is similar in appearance to the other options, it fundamentally changes the RegEx activity from a matching operation to a match-and-replace operation. Placing “s” in front of the surrounding slashes (`//`) signifies that the pattern is being used to replace the text being matched and insert the accompanying replacement text. This operator is another peek at additional functionality that is explored in the next chapter with SAS functions. Once a pattern is matched, we can then do a variety of things with that information. A great analogy for how this works in practice is the find-and-replace functionality provided by many word processing applications—except this is much more powerful. Also, notice that there is a third slash in the examples below (in the middle of the patterns). That additional slash denotes where the matching portion of the RegEx ends and the replacement portion begins. And notice something important in the last example: *everything is a string literal*. That’s right, all the characters that occur between the second and third slash are treated as just characters. Table 2.39 provides some examples, but we cover this in detail in the next chapter, where we also discuss how to insert more than just character strings.

Table 2.39: Examples using `s//`

Usage	Matches	Replaces with
<code>s/Stop/Go/</code>	“Stop”	“Go”
<code>s/Sing/Read/</code>	“Sing”	“Read”
<code>s/1\s?(800)\s?-\s?/1-800-/</code>	“1 (800) - ” ...	“1-800-”

Note: This is a more advanced function that our test code is not set up to handle. You’ll just need to accept it as true until we use it with some SAS code in the next chapter.

2.7 Zero-width Metacharacters

Zero-width characters, often called positional characters, are not matched in isolation because they do not have a width. They are used as an additional piece of information for making a proper pattern match.

There are numerous examples for how these zero-width characters can be used. For instance, perhaps you want to match a particular word, but only if it occurs at the beginning of a line.

2.7.1 Start of Line

The metacharacter `^` matches the beginning of a line or string. Depending on the text that we are processing, we might know a priori that a new line signifies something specific. For example, we might be looking for the beginning of a new paragraph, which could be denoted by a new line in combination with a capital letter and no preceding white space. Or we might need to be prepared to match an address that includes a new line for the city, state, and zip. Table 2.40 provides examples.

Table 2.40: Example using `^`

Usage	Matches
<code>/^Washington, DC 20007/</code>	“ Washington, DC 20007”
<code>/^\w+\b/</code>	The first word in a string.

Note: This metacharacter is often used as the logical NOT symbol, including within the character class metacharacters discussed in Section 2.3 and in SAS code. So be careful not to get confused in its usage when shifting between contexts.

2.7.2 End of Line

The metacharacter `$` matches the end of a line or string. There are numerous situations in which this might become relevant, similar to the reasons for the `^` metacharacter. Table 2.41 provides examples.

Table 2.41: Example using `$`

Usage	Matches
<code>/3000 K Street NW,\$/</code>	“3000 K Street NW, ”
<code>^\\$d+?\.\d{2}\s*?\$/</code>	“\$150.52 ”

2.7.3 Word Boundary

The metacharacter `\b` matches a word boundary. The `\b` RegEx assertion metacharacter is zero-width because it actually represents the invisible gap between two characters, with a `\w` character on one side and `\W` on the other. Therefore, when you use this metacharacter, you won’t generate matches that contain the associated non-word character. Table 2.42 provides examples.

Table 2.42: Example using \b

Usage	Matches
/Street\b/	“Street” from the substrings, “Street,” “Street ” ... But does NOT match from the substring “Streets” etc.
^b8\d{2}\b/	“800” “888” ... from the substrings “(800)” “-888-“ ... But does NOT match from the substrings “18002” ...
^b\U[a-z]+\E\b/	Words in all caps. Without the second \b, the output would also include single capitalized letters from the front of a word.

2.7.4 Non-word Boundary

The metacharacter \B matches a non-word boundary (i.e., anywhere \b does not match). This is especially useful for matching root words or substrings without including the surrounding pieces of information. Table 2.43 provides examples.

Table 2.43: Examples using \B

Usage	Matches
/read\B/	“read” from the substrings, “reads” “reading” “reader” ... But does NOT match from the substring “read”
^Bun\b/	“un” from the substrings, “fun ” “rerun.” “gun,” ... But does NOT match from the substring “un”
^b[a-zA-Z]{3,}\b/	Any word longer than three letters.

2.7.5 String Start

The metacharacter \A matches the beginning of a string. Similar to the word boundary metacharacter (\b), \A occurs between two character cells. It also denotes when a string value occurs to its right with nothing to its left. In the context of data lines (as in our test code for this chapter), that situation occurs at the beginning of each line.

However, suppose we had a more complex task such as stitching together multiple strings of extracted text (stored in SAS variables). In this context, \A could be a key to determining in what order to place or sort them. However, for our test code, the \A matches only on the beginning of each data line, since each line is identified as the beginning of the string. So, this is another one that you have to approach with a little bit of faith until we start doing some more interesting tasks in the next chapter. Table 2.44 provides examples.

Table 2.44: Examples using \A

Usage	Matches
<code>^A\w*?\s/</code>	The first word of a line. In the case of our test code, it matches: “ROBERT ” from line 2; “103 ” from line 4; “508 ” from line 5; “650 ” from line 6; “3000 ” from line 7; and “1560 ” from line 8.

2.8 Summary

We have explored a variety of interesting new concepts in this chapter, and I’ve been doing my utmost to make them tangible along the way. Hopefully, you are now ready to tackle the challenge of implementing these concepts in SAS code in the coming chapters. Following are some takeaways you should keep in mind for the coming pages and beyond.

Flexibility

It should have become clear through reading this chapter that there are many ways to accomplish the same task, making few of them truly right or wrong. You have to decide the most efficient and effective approach for accomplishing your goals to determine what is best for a given situation.

Scratching the Surface

We have only begun to scratch the surface of what RegEx can do. The information you have learned thus far is a solid foundation upon which you can develop sophisticated functionality.

Start Small

As we have explored a variety of RegEx capabilities throughout this chapter, it is easy to become overwhelmed with attempting to do too much at once. As with anything, it is best to start small by experimenting with simple patterns and iteratively evolve them. And remember that leveraging just a few of the elements we have covered can have a tremendous impact on the processing and analysis of textual information.

-
- ¹ Octal is a number system that uses base-8 instead of base-10. This system has only numbers 0–7 represented. Some old microcontrollers and microprocessors used this encoding, but it is extremely rare today.
- ² Hexadecimal is a number system that uses base-16 instead of base-10. The possible values go from “0” to “F” in a single character position (where A=10, B=11, ..., F=15).

Chapter 3: Using Regular Expressions in SAS

3.1 Introduction	39
3.1.1 Capture Buffer	39
3.2 Built-in SAS Functions	40
3.2.1 PRXPARSE	40
3.2.2 PRXMATCH	42
3.2.3 PRXCHANGE	43
3.2.4 PRXPOSN	46
3.2.5 PRXPAREN	47
3.3 Built-in SAS Call Routines	49
3.3.1 CALL PRXCHANGE	50
3.3.2 CALL PRXPOSN	54
3.3.3 CALL PRXSUBSTR	56
3.3.4 CALL PRXNEXT	57
3.3.5 CALL PRXDEBUG	59
3.3.6 CALL PRXFREE	62
3.4 Summary	63

3.1 Introduction

This chapter is focused on developing your understanding of built-in SAS functions and call routines, and on starting to do some real SAS coding. Here, you will learn the mechanics of how to implement the wonderful RegEx metacharacters introduced in Chapter 2. Each function or call routine introduced has associated examples to ensure that their use is clear. We also briefly discuss how each is useful.

3.1.1 Capture Buffer

Now, before we go any farther, we have to address a concept called the *capture buffer*. The capture buffer is a more advanced technique that I have avoided delving into thus far, but it must be understood so that you can use some functions (required for PRXPAREN and PRXPOSN, but optional for PRXCHANGE). As you should recall from Chapter 2, parentheses create logical groupings within a RegEx, but they also do something more interesting. For every set of parentheses used in a particular RegEx pattern, a slot in a memory buffer is created. This slot in memory is then referenceable just like

any variable (a more experienced programmer can think of it like a pointer buffer). Each slot is created in sequential order of parentheses pair occurrence and is referenced accordingly using the \$ sign.

For example, the RegEx `s/(The) (cat) (is) (fat)/$4 $3 $1 $2/` creates the output “fat is The cat”. Now, imagine applying that same ability to unknown data elements instead of just to string literals. This could become a very powerful capability for standardizing or restructuring data to meet specific needs.

3.2 Built-in SAS Functions

In this section, we cover the SAS functions for performing RegEx operations. SAS functions for RegEx have the same usage limitations as other built-in functions. (See SAS documentation.) Also just like all other functions, they can only take arguments and return output in assignment statements and expressions.

Note: Each RegEx function has PRX at the beginning, which represents Perl-Regular-eXpressions.

3.2.1 PRXPARSE

Description

This function takes a RegEx pattern as input and provides a numerical RegEx pattern identifier as output. The unique pattern identifier is used by other functions and call routines to reference the pattern. This function should look familiar since we used it in our example code in Chapter 2.

Syntax

```
RegEx_ID = PRXPARSE (RegEx)
```

RegEx: The pattern to be parsed (input argument, required)

RegEx_ID: Unique numerical RegEx identifier returned by PRXPARSE (output, required)

Now, it is important to understand at this point that PRXPARSE compiles the RegEx in order to create the identifier for SAS to later reference and use. And this is what makes the RegEx `//o` option so important when using PRXPARSE in code. The `//o` option forces SAS to compile the RegEx code once, creating the RegEx identifier the first time only. When a particular RegEx is intended to be reused on every loop through the DATA step, we want to leverage this functionality in order to avoid recompiling the RegEx pattern every time it is encountered in code (i.e., on each iteration of a DATA step). If the pattern is not definitely going to be used every time through the DATA step (e.g., it’s not defined inside an IF statement), then we might not want to waste memory maintaining it. In other words, we might not always want to use the `//o` option—the decision is about tradeoffs. When you’re dealing with very few of these patterns or with a small amount of data, the tradeoffs don’t really apply. But when we scale up to a system using hundreds of patterns, or tens of millions of records, the tradeoffs (speed at the expense of memory usage) become very important.

Example 3.1: Defining Patterns with PRXPARSE

Let's revisit the last bit of example code from Chapter 2 since it is already familiar. The RegEx below is defined as `/Smith/o`, meaning that we are looking for any occurrence of the string literal "Smith" within the data lines provided. This RegEx is the argument for `PRXPARSE`, which creates a pattern identifier that is assigned to the variable `pattern_ID`. This variable is then passed to the call routine `PRXSUBSTR`, which is discussed in the next section. Because you are familiar with the overall function of this code by now, this need not be a distraction.

The output of this RegEx is presented in Output 3.1. As we expected, the code found every occurrence of "Smith" regardless of what was surrounding it—including other letters.

```

/*RegEx Testing Framework*/
data _NULL_;
pattern = "/Smith/o"; /*<--Edit the pattern here.*/
pattern_ID = PRXPARSE(pattern);
input some_data $50.;
call prxsubstr(pattern_ID, some_data, position, length);
if position ^= 0 then
  do;
    match = substr(some_data, position, length);
    put match:$QUOTE. "found in " some_data:$QUOTE.;
  end;
datalines;
Smith, BOB A.
ROBERT Allen Smith
Smithe, Cindy
103   Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
;
run;

```

Output 3.1: Log Output of Pattern /Smith/o

```

"Smith" found in "Smith, BOB A."
"Smith" found in "ROBERT Allen Smith"
"Smith" found in "Smithe, Cindy"

```

3.2.2 PRXMATCH

Description

PRXMATCH returns the numerical position of the first character in the matched RegEx pattern. Additionally, it can be used in IF statements to test for a pattern match without a variable assignment, just like many other familiar SAS functions. The first argument to PRXMATCH is either the RegEx or RegEx_ID. The second is the source text variable or string literal.

Syntax

```
Position = PRXMATCH(RegEx_ID or RegEx, Source_Text)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required if RegEx not used)

RegEx: The pattern to be matched (input argument, required if RegEx_ID not used)

Source_Text: The text variable or literal to be operated upon (input argument, required)

Position: Numerical position variable assignment (output, required)

As we discussed with the PRXPARSE function, RegEx patterns are compiled by SAS for use by other functions. Therefore, in addition to using the actual RegEx, PRXMATCH is able to leverage the previously compiled RegEx via the RegEx_ID argument in lieu of the RegEx itself. This allows us to compile the RegEx once via the PRXPARSE function (using the //o option), minimizing the associated computing cycles. Such small savings in computing cycles can prove significant when processing large volumes of text.

The two different methods for leveraging RegEx patterns create significant flexibility in how PRXMATCH can be used in practice. By not needing to compile the RegEx in advance, PRXMATCH allows us to embed RegEx patterns throughout our code without the extra memory allocation required to maintain them for each loop through the DATA step. This is very useful when you are using PRXMATCH in a dynamic way, such as inside nested IF statements where the RegEx is used only when certain conditions are true. Depending on the implementation, there are implications for speed as well as for memory usage.

Example 3.2: Finding Strings in Source Text with PRXMATCH

Let's try a simple example to see how this function is used in practice. Suppose we want to find a string such as "Street" in a source text. The code below demonstrates how we print the position of each occurrence to the log. Obviously, we need to do more than just print the position in practice (such as by extracting or manipulating the matched text), but this demonstrates the basic functionality of PRXMATCH.

The PRXMATCH function is implemented in this code with the RegEx as the first argument and the `datalines` reference address as the second argument. The result is assigned to the variable `Position`. The value of `Position` is then written to the log using the `PUT` statement.

Count the character positions in the data lines. At what position do we encounter the “S” in “Street” on the lines in which they occur? Comparing the results to Output 3.2, we see that PRXMATCH is returning the position of “S” (i.e., the position for the first character in the pattern match).

```
data _NULL_;
input address $50.;
position = PRXMATCH('/Street/o', address);
if position ^= 0 then
  do;
    put position=;
  end;
datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First Street NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
;
```

Output 3.2: Log Printout for Positions of “Street”

```
position=11
position=8
```

3.2.3 PRXCHANGE

Description

This function searches for the pattern—provided in the first argument by either `RegEx_ID` or `RegEx`—within the source text that is provided in the third argument. The pattern is matched the number of times given in the second argument, `Num_Times`. Upon finding each match, the function then returns the changed text as required by the `RegEx`. If no match is found, `PRXCHANGE` returns the original text unchanged.

Syntax

```
Output_String = PRXCHANGE(RegEx_ID or RegEx, Num_Times, Input_String)
```

`RegEx_ID`: Unique `RegEx` identifier returned by `PRXPARSE` (input argument, required if `RegEx` not used)

`RegEx`: The pattern to be matched (input argument, required if `RegEx_ID` not used)

Num_Times: Number of times the change is to be applied (input argument, required). -1 forces the function to make the changes as many times as the pattern occurs in the source text.

Input_String: Input text variable (input argument, required)

Output_String: Output text variable assignment (output, required)

Just like the PRXMATCH function, PRXCHANGE is able to use the actual RegEx pattern or the RegEx_ID, providing significant flexibility. The preferred use again depends on the desired application.

This function is very useful for data standardization, as you will see in more advanced examples in the next chapter. We will work through two examples below to demonstrate some more basic functionality of the PRXCHANGE function, as well as to demonstrate how to leverage the capture buffer concept introduced earlier.

Example 3.3: Standardizing Data

Data standardization is a relatively simple, yet powerful, capability provided by RegEx in SAS. PRXCHANGE enables us to scrub our data source to ensure that each occurrence of a word or phrase is exactly the same (or removed entirely). See Output 3.3 for the results. Data scrubbing becomes especially important when you are attempting to perform advanced applications such as text mining.

For instance, before doing any analysis of our data, we want to know that each occurrence of the word “street” is exactly the same. If each occurrence were not identical, we might perform a word frequency count on a document with invalid results because “street,” “Street,” “St.,” and so on would all be counted separately. Depending on the eventual use of this information, such problems could prove disastrous.

```
data _NULL_;
input address $50.;
text = PRXCHANGE('s/\s+([sS]t(reet)?|st\.)\s+/ St. /o',-1,address);
put text;

datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
;
run;
```

Output 3.3: Log with Updated Data

```
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St. NE, Washington, DC 20002
3000 K St. NW, Washington, DC 20007
```

Note: There are often a number of ways to achieve the same outcome. Understanding the context of an application will help you determine the best RegEx pattern to use.

Example 3.4: Using the Capture Buffer

Revisiting Example 3.3, suppose we now want to also make the addresses available to a system that accepts only comma separated values (CSV) files. This is a great opportunity to use the capture buffer. With only a couple of minor code changes, we can now process the data lines to be CSV ready.

The new line of code uses PRXCHANGE with a more complex RegEx that chunks the address into the street, city, state, and ZIP code components. And we see that the new line takes the previous output variable Text as the input argument, instead of address. Doing this allows us to make changes to the already changed text. If we were to use address, we would merely update the original data lines rather than building on the prior step.

In reviewing the parentheses elements in the new RegEx, we can see how the four address components are identified. On that same line, each of the four elements is placed via the buffer reference, with a comma and space immediately following. Reviewing Output 3.4, we see that the code produces the expected outcome.

```
data _NULL_;
input address $50.;
text = PRXCHANGE('s/\s+(Street|street|St|st|st\.)\s+/ St. /o',-1,address);
text2 = PRXCHANGE('s/(.+?),*?\s+(\w+?),*?\s+(\w+)\s+(\d+)/$1, $2, $3,
$4/o',-1,text);
put text2;

datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
;
run;
```

Output 3.4: Corrected Data in the Log

```
103 Pennsylvania Ave NW, Washington, DC, 20216
508 First St. NW, Washington, DC, 20001
650 1st St. NE, Washington, DC, 20002
3000 K St. NW, Washington, DC, 20007
```

So, what else could we do to the text? A number of things remain to be performed in order to make these addresses ready for advanced applications. For instance, “Ave” and “1st” should also be standardized. Building on the example code above is the fastest way to explore the options and become more comfortable with some of these concepts.

3.2.4 PRXPOSN

Description

PRXPOSN returns the matched information from specified capture buffers. This RegEx function requires the RXSUBSTR, PRXMATCH, PRXNEXT, or PRXCHANGE functions to be running before being used so that the capture buffer can be referenced. Also, RegEx_ID is required rather than the actual RegEx. Otherwise, PRXPOSN will not work—necessitating the use of PRXPARSE. The N input argument is numeric and refers to the capture buffer (without \$).

Syntax

```
Text = PRXPOSN(RegEx_ID, N, Source_Text)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required)

N: Integer value of the capture buffer (input argument, required)

Source_Text: The text variable or literal to be operated upon (input argument, required)

Text: Character variable assignment of captured text (output, required)

When we know the exact number of existing capture buffer elements (i.e., N is known), then we can use PRXPOSN without an issue. However, what happens when the number of elements is different from what we expect? If there are values in the capture buffer but we make a reference that is larger than those available (maybe there are three, but we make a reference to number 5), then a missing value is returned. However, if we reference capture buffer position 0 (N=0), then the entire pattern match is returned regardless of the buffer length.

The next function, PRXPAREN, is very helpful in creating robust code when you are using the capture buffers in conjunction with PRXPOSN. It is also important to write robust RegEx patterns to ensure that you prevent issues from popping up.

Example 3.5: Extracting Data with Capture Buffers

In order to make both the capture buffer concept and this new function more clear, we're going to walk through a concrete example. Suppose we want to process addresses for which the structure is well known and store various pieces in a SAS data set for later use. Since we know the layout of the address, the capture buffer arrangement and the application of PRXPOSN are both very straightforward.

First, we create the RegEx_ID variable Text by using the PRXPARSE function. Then, we perform a logical test using the PRXMATCH function in the IF statement. Notice that this is an implicit test of a match existing (no equal sign is used). If a match of the RegEx exists within the identified text source, then we assign the various capture buffer values to variables by using PRXPOSN (city, state, and zip).

Output 3.5 displays the values of `extract`, the data set created in our DATA step. As expected, we extracted the city, state, and ZIP code from each `datalines` entry. Later, we're going to build on this

code to create a more sophisticated address extractor that includes the street information as well as the ability to include 9-digit zips.

```

data extract;
input address $50.;
text = PRXPARE('/\s+(\w+),\s+(\w+)\s+(\d+)/o');
  if PRXMATCH(text, address) then
    do;
      city = PRXPOSN(text, 1, address);
      state = PRXPOSN(text, 2, address);
      zip = PRXPOSN(text, 3, address);
      output;
    end;
keep city state zip;
datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
3000 K Street NW, Washington, DC, 20007
;
run;
proc print data=extract;
run;

```

Output 3.5: PROC PRINT Results

The SAS System			
Obs	city	state	zip
1	Washington	DC	20216
2	Washington	DC	20001
3	Washington	DC	20002
4	Washington	DC	20007

3.2.5 PRXPAREN

Description

This function returns the numerical reference value of the largest capture buffer that contains data. It is therefore implicitly required that PRXSUBSTR, PRXMATCH, PRXNEXT, or PRXCHANGE be run prior to this function being used—just like with PRXPOSN. However, the only input argument is the

RegEx_ID. Simply providing the RegEx is not an option, so this function must be used in conjunction with the PRXPARSE function.

Syntax

```
Paren=PRXPAREN (RegEx_ID)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required)

Paren: Numerical reference value of the largest capture buffer (output, required)

Note: Since this function requires a RegEx_ID in lieu of the actual RegEx, it is implied that all precedents are then forced to use RegEx_ID instead of the RegEx as well—otherwise, PRXPAREN cannot be used.

What are we really trying to achieve with this function? Since it provides the length of the capture buffer by telling us the largest buffer position to contain text, we know exactly how many possible buffer values we can access. Because we know this, we can avoid errors when referencing them in code. It is worth noting that effective RegEx coding avoids many potential problems. However, it is always best practice to create fail-safe measures. Additionally, we can use this function to identify which of several options has been triggered inside the source text.

Example 3.6: Identifying Capture Buffers

Ideally, whenever we want to use the PRXPOSN function, the data that we expect to be available in the source is available. However, we know that in reality, that is not always the case. So, we have to write code that can account for a reasonable amount of variability in any data that we might need to process. We are going to explore an advanced example in the next chapter (see Section 4.2) that leverages the basic concepts outlined by this example.

Now, suppose we have a pattern with multiple possible matches embedded in it. How do we know which option allowed the pattern to create a match? In the code below, we see that it is possible to use PRXPAREN to answer this question. We have a simple pattern with three possible matches: “Dog”, “Rat”, and “Cat”. Each is encapsulated by parentheses to create a capture buffer location. However, notice that the entire group is inside yet another set of parentheses. While unnecessary for practical purposes, this was done to demonstrate how capture buffers are numbered. Also note that this is not the most efficient way to write such code. We have sacrificed efficiency here in order to clarify how the buffers work. Notice in our output that “Dog” has a capture buffer of 2 despite being the first item in the OR list. Why? Because the outer set of parentheses is encountered first by SAS, thus creating a capture buffer element at position 1.

If we were to use PRXPOSN under each IF statement with position 1 in our argument list, we would see that each of the three cases below would be provided as output (i.e., when “Dog” is true, “Dog” would be in buffer 1 as well as in buffer 2, and so on). See Output 3.6 for the results.

```
data _null_;
  RegEx_ID=prxparse('/\b((Dog)|(Rat)|(Cat))\b/o');

  position=prxmatch(RegEx_ID, 'The Cat in the Hat');
  if position then paren=prxparen(RegEx_ID);
  put 'I matched capture buffer ' paren;

  position=prxmatch(RegEx_ID, 'The Rat in the Hat');
  if position then paren=prxparen(RegEx_ID);
  put 'I matched capture buffer ' paren;

  position=prxmatch(RegEx_ID, 'The Dog on the Roof');
  if position then paren=prxparen(RegEx_ID);
  put 'I matched capture buffer ' paren;

run;
```

As I have indicated, our goal here is to identify which particular capture buffer is used. This allows us to build more sophisticated functionality in the future, such as conditional information capture or standardization.

Output 3.6: Log Output

```
I matched capture buffer 4
I matched capture buffer 3
I matched capture buffer 2
```

3.3 Built-in SAS Call Routines

In this section, you learn about the PRX call routines available in SAS for performing many of the same RegEx tasks as the functions previously discussed, as well as some new ones. However, just like with all other call routines, PRX call routines cannot be used in expressions or assignment statements. The way they are implemented, and their ultimate functionality, is slightly different when compared to the functions. These differences are explored more thoroughly in the associated examples.

3.3.1 CALL PRXCHANGE

Description

This call routine performs the match-and-replace operation similar to that of the PRXCHANGE function. However, unlike the function version, the call routine must receive a RegEx identifier, without the option of using the associated RegEx instead. Also, there are some additional routine arguments not available in the function (`result_length` and `trunc_value`). The only required arguments are: `RegEx_ID`, `Num_Times`, and `Input_string`. All remaining arguments are optional.

Syntax

```
CALL PRXCHANGE(RegEx_ID, Num_Times, Input_string, Output_string,  
result_length, trunc_value, num_changes)
```

`RegEx_ID`: Unique RegEx identifier returned by PRXPARSE (input argument, required)

`Num_Times`: Number of times the change is to be applied (input argument, required)

`Input_string`: Input text variable (input argument, required)

`Output_string`: Output text variable (input argument, optional). Default is `Input_string`.

`result_length`: Length of the characters put into `Output_string` (returned value, optional)

`Trunc_value`: Binary integer (0 or 1 only) value (returned value, optional). 1 means that the inserted text is longer than the text replaced. 0 means that the inserted text is either the same length or shorter than the text being replaced.

`num_changes`: The number of times the changes were made (returned value, optional)

Using the call routine in lieu of the function can often be cleaner from a coding perspective, especially when managing large programs. But there is a more practical reason for using this call routine instead of the function: accessing the additional functionality provided by the optional arguments. Since we have the ability to write changes directly back to the original variable, we can avoid creating new variables unnecessarily. This is especially useful when applying multiple data standardization filters to source text.

Note: Writing changes back to the existing variable makes them irreversible in the event of a mistake. So, while our ultimate use of this functionality requires the overwriting approach for sound memory management, creating new variables or data sets is ideal when you are still learning. It allows you to experiment and make mistakes without fear of making permanent changes to source data.

Example 3.7: Transforming Data

Let's look at basic usage for making changes to our source text. This is a simple example of how to use the call routine. Notice how compact this makes our code while maintaining functionality. Output 3.7 demonstrates that we have the anticipated functionality (replacing various forms of "street" with "St.").

```
data _NULL_;
input address $50.;
mypattern = PRXPARSE('s/\s+(Street|street|St|st|st\.)\s+/ St. /o');
CALL PRXCHANGE(mypattern,-1,address);
put address;

datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St NW, Washington, DC 20001
650 1st St NE, Washington DC 20002
3000 K Street NW, Washington, DC 20007
;
run;
```

Output 3.7: Results in the SAS Log

```
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St. NE, Washington DC 20002
3000 K St. NW, Washington, DC 20007
```

Now that we've looked at a basic implementation of CALL PRXCHANGE, let's explore the optional arguments.

Example 3.8: Redacting Sensitive Data

In this example, we focus on developing your understanding of the optional elements in CALL PRXCHANGE. As a change of pace, we're going to develop a basic way to redact sensitive information. This is a frequent need, especially in the medical field, for protecting Personally Identifiable Information (PII)¹. Now, we're not going to eliminate all PII from the provided data because we are just demonstrating the functionality of CALL PRXCHANGE. However, this process is done more rigorously in the next chapter and on a larger scale.

52 Introduction to Regular Expressions in SAS

In the code below, we start by creating a data set to pass into the DATA step, called `example`. This data set contains name, address, and phone number information in various configurations. In the DATA step, we create a `RegEx_ID` using `PRXPARSE`, and then use `CALL PRXCHANGE` to execute the changes prescribed by the `RegEx`. Notice that one `RegEx_ID` is commented out. The `RegEx` in that line behaves very differently from the initial `RegEx_ID` definition, which allows us to demonstrate the `trunc_val` and `num_changes` options. We use this commented `RegEx_ID` to create Output 3.8.2. After the DATA step, we perform a `PROC PRINT` to create the output shown in both Output 3.8.1 and Output 3.8.2.

```
data example;
  input text $80.;
  datalines;
Ken can be reached at (801)443-9876
103 Pennsylvania Ave NW, Washington, DC 20216
JP's address is:
650 1st St NE, Washington DC 20002
Carla's information is: (910)998-8762
3000 K Street NW, Washington, DC 20007
Eric can be reached at: (321) 456-7890
508 First St NW, Washington, DC 20001
;
run;

data changed;
  set example;

  *RegEx_ID = PRXPARSE('s/\d+/**NUMBER REMOVED**/'o');
  RegEx_ID = PRXPARSE('s/\s?([1-9]\d\d)\s?[1-9]\d\d-
\d\d\d\d/*REDACTED*/o');
  Call PRXCHANGE(RegEx_ID, -1, text, text, length, trunc_val,
num_changes);
  put text=;
run;

proc print data=changed;
run;
```

Simple Insert Results

As we can see in the results below, the phone numbers have been redacted in the original text by using the value `*REDACTED*`. The rest of the data set shows our optional variable values. `Length` is the total length of the string written to `Text` (we just wrote back to the old string this time). `trunc_val` is 0 for every row because the inserted value is no longer than the original phone numbers. In fact, lines 1, 5, and 7 shrink because the inserted content is shorter. And finally, `num_changes` records the number of times the phone numbers were redacted on each line (multiple phone numbers per line would have resulted in that number occurring in this column).

Output 3.8.1: SAS PROC PRINT Results

The SAS System					
Obs	text	RegEx_ID	length	trunc_val	num_changes
1	Ken can be reached at *REDACTED*	1	77	0	1
2	103 Pennsylvania Ave NW, Washington, DC 20216	1	80	0	0
3	JP's address is:	1	80	0	0
4	650 1st St NE, Washington DC 20002	1	80	0	0
5	Carla's information is: *REDACTED*	1	77	0	1
6	3000 K Street NW, Washington, DC 20007	1	80	0	0
7	Eric can be reached at: *REDACTED*	1	76	0	1
8	508 First St NW, Washington, DC 20001	1	80	0	0

More Advanced Insert Results

The commented RegEx_ID definition creates very different output for Output 3.8.2—a longer replacement value that occurs for every group of numbers (**NUMBER REMOVED** is inserted). The variables are all the same as in Output 3.8.1, but notice how the values change. For instance, trunc_val now equals 1 every time a redaction occurred, and num_changes is frequently greater than 1. Also, notice something else very important about this output: some lines of text are actually truncated!

Remember, the trunc_val variable being set to 1 does not mean that data loss is certainly going to occur. Instead, it means that it could occur. Think of this as a warning flag telling us, “Hey, keep a look out for a problem.” And a problem is what we would indeed have for some of these lines of text. The insertion of longer text pushes all following text to the right (beyond the 80-character length defined for the variable Text). Now, when there is a significant amount of white space to the right of our text, this doesn't result in an issue. However, when there is valuable information to the right of our inserted text, we will likely have data loss. Regardless how small the loss of data, the integrity of our entire data set is compromised if we do not design code that avoids this problem. We discuss this concept a bit more in the next chapter.

Output 3.8.2: SAS PROC PRINT Results

The SAS System					
Obs	text	regex_id	length	trunc_val	num_changes
1	Ken can be reached at (***)NUMBER REMOVED(***)***)NUMBER REMOVED(***)-***)NUMBER REMOV	1	80	1	3
2	***NUMBER REMOVED*** Pennsylvania Ave NW, Washington, DC ***NUMBER REMOVED***	1	80	1	2
3	JP's address is:	1	80	0	0
4	***NUMBER REMOVED*** ***)NUMBER REMOVED(***)***)st St NE, Washington DC ***NUMBER REMOV	1	80	1	3
5	Carla's information is: (***)NUMBER REMOVED(***)***)NUMBER REMOVED(***)-***)NUMBER REM	1	80	1	3
6	***NUMBER REMOVED*** K Street NW, Washington, DC ***NUMBER REMOVED***	1	80	1	2
7	Eric can be reached at: (***)NUMBER REMOVED(***) ***)NUMBER REMOVED(***)-***)NUMBER RE	1	80	1	3
8	***NUMBER REMOVED*** First St NW, Washington, DC ***NUMBER REMOVED***	1	80	1	2

3.3.2 CALL PRXPOSN**Description**

This call routine takes the RegEx_ID provided by PRXPARSE and the numerical capture buffer position N as inputs. It produces the matching Position and Length as outputs.

Syntax

```
CALL PRXPOSN(RegEx_ID, N, Position, Length)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required)

N: Integer value of the capture buffer (input argument, required)

Position: Integer value of the character position for the first character in the matched pattern (returned value, required)

Length: Integer value for the length of the matched pattern (returned value, optional)

This call routine takes in the RegEx_ID (RegEx is not allowed!) and capture buffer, and returns the exact locations where it occurs in the most recent match. The match results from PRXMATCH, PRXCHANGE, PRXSUBSTR, or CALL PRXNEXT (discussed in Section 3.3.4) must exist in order for CALL PRXPOSN to work properly. We then must use the SUBSTR function to extract the identified text.

Example 3.9: Context-specific Algorithm Development

Sometimes it's useful to condition code behavior on specific words occurring in text. In this example, you'll see how the functionality of CALL PRXPOSN can be used in combination with PRXPAREN, PRXMATCH, PRXPAREN, and SUBSTR to do just that.

First, we create the RegEx_ID by using PRXPAREN, which is then passed to PRXMATCH. If a result from PRXMATCH exists (i.e., a pattern match is found), then we determine which of the capture buffers in the pattern is matched via PRXPAREN. The output of PRXPAREN is used as input to the CALL PRXPOSN routine to create the Position and Length outputs. SUBSTR is then used to extract the specified text. We then print a message, depending on the buffer position. See the results in Output 3.9.

```

data _null_;
input text $50.;
    RegEx_ID=prxparse('/((Dog)|(Rat)|(Cat))/o');

    if prxmatch(RegEx_ID, text) then do;
        paren=prxparen(RegEx_ID);
        CALL PRXPOSN(RegEx_ID, paren, position, length);
        buffer = substr(text, position, length);
        put 'I matched capture buffer ' paren 'with ' buffer;
    end;

    if paren=2 then put 'I love dogs!';
    else put 'I cannot stand a ' buffer!';

datalines;
The Cat in the Hat
The Rat in the Hat
The Dog on the Roof
;
run;

```

Output 3.9: Log Output of Code Behavior

I added the additional commentary about cats, rats, and dogs to show a second way to perform conditioning on the parsed text. Obviously, you could perform more interesting things, and it should be fun to experiment with in the future. We use this concept in the next chapter to build out some interesting functionality.

```

I matched capture buffer 4 with Cat
I cannot stand a Cat !
I matched capture buffer 3 with Rat
I cannot stand a Rat !
I matched capture buffer 2 with Dog
I love dogs!

```

3.3.3 CALL PRXSUBSTR

Description

This call routine takes RegEx_ID and Source_Text as inputs, and returns Position and Length as outputs. Using the actual RegEx is not an option.

Syntax

```
CALL PRXSUBSTR(RegEx_ID, Source_Text, Position, Length)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required)

Source_Text: The text to be operated upon (input argument, required)

Position: Integer value of the character position for the first character in the matched pattern (returned value, required)

Length: Integer value for the length of the matched pattern (returned value, optional)

This call routine is used extensively in Information Extraction applications like those discussed in Chapter 4. Since only the position and length of matches are identified by CALL PRXSUBSTR, it must be used in conjunction with a function like SUBSTR in order to extract the actual text.

Example 3.10: Information Extraction

In this example, we revisit the now-familiar example code from Chapter 2. It is a great example of how you can leverage the CALL PRXSUBSTR in many applications.

The code below creates a RegEx pattern to search for all occurrences of “Smith” in our source text. It then generates a RegEx_ID using PRXPARSE. The code then uses CALL PRXSUBSTR to search through source text with the provided pattern and return the position and length of matching text. As you know by now, this could have been a much more complex pattern, but the simplicity here helps to highlight the functionality that we are focused on learning. After the call routine, the code checks to see whether the position variable (Position) is 0, which is the default value indicating that it did not find a match. If a position does exist, the code proceeds to use SUBSTR to capture text from the source using the position and length obtained by CALL PRXSUBSTR. Results are then output to the log. See Output 3.10.

```
data _NULL_ ;
pattern = "/Smith/o"; /*<--Edit the pattern here.*/
pattern_ID = PRXPARSE(pattern);
input some_data $50.;
CALL PRXSUBSTR(pattern_ID, some_data, position, length);
if position ^= 0 then
  do;
    match = substr(some_data, position, length);
    put match:$QUOTE. "found in " some_data:$QUOTE.;
```

```

end;
datalines;
Smith, BOB A.
ROBERT Allen Smith
Smithe, Cindy
103 Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
;
run;

```

As we expected, the output shows the various occurrences of “Smith” from within the provided data lines. This example brings us full circle with the above code, pulling all of the pieces together.

Output 3.10: Log Results for “Smith”

```

"Smith" found in "Smith, BOB A."
"Smith" found in "ROBERT Allen Smith"
"Smith" found in "Smithe, Cindy"

```

3.3.4 CALL PRXNEXT

Description

This routine searches through `Source_Text`, between the `Start` and `Stop` positions, for the pattern associated with `RegEx_ID`. It returns the `Position` and `Length` of the location.

Syntax

```
CALL PRXNEXT(RegEx_ID, Start, Stop, Source_Text, Position, Length)
```

`RegEx_ID`: Unique RegEx identifier returned by PRXPARSE (input argument, required)

`Start`: Numerical constant, variable, or expression containing the starting character position to begin the search (input argument, required)

`Stop`: Numerical constant, variable, or expression containing the last character position to use in the search. If the value is -1, the stop position becomes the last non-blank character position in the source. (input argument, required)

`Source_Text`: The text to be operated upon (input argument, required)

`Position`: Integer value of the character position for the first character in the matched pattern (returned value, required)

Length: Integer value for the length of the matched pattern (returned value, required)

This call routine can be used for two applications:

1. searching for a pattern within a defined range
2. searching for a pattern iteratively throughout text, including multiple occurrences per line

The first application of CALL PRXNEXT is a straightforward implementation of the routine's parameters. However, the second usage is less apparent from its definition. Therefore, we focus on that usage of the routine in our example.

Example 3.11: Pattern Matching Multiple Times per Line

Being able to identify a pattern any number of times in a particular line of text is valuable for many practical applications. For example, performing word frequency counts clearly requires this ability in order for accurate counts to be obtained.

The code below shows how to use CALL PRXNEXT to identify multiple occurrences of our pattern on each row from the data-lines source. The pattern is defined to match on any string that is three word characters (\w) in length and that ends with "un". The Start and Stop variables are initialized to character positions 1 and Length(some_data) respectively. These variables must be provided with initial values for the routine's first use. However, subsequent calls automatically reset the start position to the character position immediately following the most recent successful match. This is a fact that we take advantage of with the DO WHILE loop below. If we were to eliminate the loop portion of code, we would merely be searching for the pattern in a defined range (use the above application #1), but having the loop allows us to achieve the desired functionality (use the above application #2). See the results in Output 3.11.

```

data _NULL_;
input some_data $50.;

pattern = "\wun/o";
pattern_ID = PRXPARSE(pattern);
start = 1;
stop = length(some_data);

CALL PRXNEXT(pattern_ID, start, stop, some_data, position, length);
do while (position > 0);
    found = substr(some_data, position, length);
    put "Line:" _N_ found= position= length=;

    CALL PRXNEXT(pattern_ID, start, stop, some_data, position, length);
end;

```



```

datalines;
Running Runners who run.
Runners who think running is fun.
"Fun Runs" are not-so-fun runs for me.
Let's run at the next reunion.
;
run;

```

Log Output of Pattern Match Results

Output 3.11 contains the literal string that was found, its position, and its length. As we should expect by now, the pattern that is created ignores the surrounding text—which makes the example slightly more interesting. Review the output (count the character locations in the data lines), and notice that we did indeed achieve the desired results.

Output 3.11: Log Output of Pattern Match Results

```

Line:1 found=Run position=1 length=3
Line:1 found=Run position=9 length=3
Line:1 found=run position=21 length=3
Line:2 found=Run position=1 length=3
Line:2 found=run position=19 length=3
Line:2 found=fun position=30 length=3
Line:3 found=Fun position=2 length=3
Line:3 found=Run position=6 length=3
Line:3 found=fun position=23 length=3
Line:3 found=run position=27 length=3
Line:4 found=run position=7 length=3
Line:4 found=eun position=24 length=3

```

3.3.5 CALL PRXDEBUG

Description

This routine is used to perform debugging of all PRX functions and call routines, and accepts only one input.

Syntax

```
CALL PRXDEBUG (ON-OFF)
```

ON-OFF: Numerical constant, variable, or expression. If it equals 0, then debugging is turned off, but any positive value turns it on. (input argument, required)

This routine prints step-by-step output to the log, enabling a low-level understanding of any PRX program. However, be prepared—this routine can create voluminous output. It is best to use it in a targeted way at first in order to understand how a specific function or routine is working (or not working). If we were to use this routine for an entire program, we should be ready to read very large amounts of procedural output, which is an inefficient approach to diagnosing issues. It is best to perform gross-level diagnostics using PUT statements and dummy variables, thus narrowing the focus to a specific code segment before using CALL PRXDEBUG. In practice, this is the fastest approach to identifying the source of logical errors.

Example 3.12: Debugging the PRXPARSE Function

In keeping with our goal of using the CALL PRXDEBUG in a targeted way to debug code, we are going to apply it only to the PRXPARSE function in the code below. Notice that we have to turn it on and off at different points in the code in order to identify the segment to which we want our debug output limited. See the results in Output 3.12.

```

data _null_;
input text $50.;

CALL PRXDEBUG(1);
  RegEx_ID=prxparse('/((Dog)|(Rat)|(Cat))/o');
CALL PRXDEBUG(0);
  if prxmatch(RegEx_ID, text) then do;
    paren=prxparen(RegEx_ID);
    CALL PRXPOSN(RegEx_ID, paren, position, length);
    buffer = substr(text, position, length);
    put 'I matched capture buffer ' paren 'with ' buffer;
  end;

  if paren=2 then put 'I love dogs!';
  else put 'I cannot stand a ' buffer!';

datalines;
The Cat in the Hat
The Rat in the Hat
The Dog on the Roof
;
run;

```

Debugging Information Printed to the Log

Reviewing the output in Output 3.12, we see that the debugging information for just a single PRX function can be quite large, thus reinforcing my earlier point about limiting the scope of CALL PRXDEBUG.

The first line denotes compilation of a RegEx within the PRXPARSE function. The next line shows us the compiled RegEx size and starting location for the lines that follow. Specifically, the size of 26 refers to the 26 lines of compiled RegEx code (numbers on the left with a trailing semi-colon), and `first` refers to the first line of code execution. The numbers in parentheses to the right of each line correspond to labels for the compiled RegEx (these labels work much like our pseudo code labels in Chapter 2).

Lines 1 through 26 are the compiled steps within our RegEx, and they become easy to follow once we understand what each represents. For instance, the various OPEN and CLOSE statements correspond to our opening and closing parentheses; BRANCH corresponds to the OR tests between the inner three parenthesis pairs; and EXACT is for the string literal match of the associated word. END obviously means the end of the subroutine.

The remaining output is just the rest of our code running as normal. Should our code be malfunctioning, we would not likely see such normal output when using CALL PRXDEBUG.

Output 3.12: Debugging Information Printed to the Log

```

① Compiling REx '((Dog)|(Rat)|(Cat))'
   size 26 first at 3
②   1: OPEN1(3)
     3:  BRANCH(10)
     4:  OPEN2(6)
     6:    EXACT <Dog>(8)
     8:  CLOSE2(24)
    10: BRANCH(17)
    11:  OPEN3(13)
    13:    EXACT <Rat>(15)
    15:  CLOSE3(24)
    17: BRANCH(24)
    18:  OPEN4(20)
    20:    EXACT <Cat>(22)
    22:  CLOSE4(24)
    24: CLOSE1(26)
    26: END(0)
③ minlen 3
④ | matched capture buffer 4 with Cat
   | cannot stand a Cat !
   | matched capture buffer 3 with Rat
   | cannot stand a Rat !
   | matched capture buffer 2 with Dog
   | love dogs!

```

- ① The compilation process begins for the quoted RegEx contained by PRXPARSE.
- ② Notice that each OPEN and CLOSE pair have the same number (OPEN1 and CLOSE1). These numbers correspond to the numerical value of the capture buffer that was formed by that set of parentheses.
- ③ Each line ends with a number enclosed in parentheses, denoting the next line to jump to from that line. However, the END tag shows a jump to 0, which takes us out of the subroutine.
- ④ The minlen field defines the minimum length for the match to be 3. This information is used by subsequent functions and routines when using this compiled pattern.

Moving the placement of our debug routine call should prove to yield some interesting, and potentially rather long, output. Doing so is the best way to become more familiar with the low-level operations SAS is performing behind the scenes of our PRX code.

Significant amounts of information can be provided by the PRXDEBUG output, but a much deeper study of debug output is outside the scope of this text. For more information about debug output and its meaning, visit the SAS Support website².

3.3.6 CALL PRXFREE

Description

This call routine releases memory resources associated with a RegEx, using its unique RegEx_ID. Subsequent references to this identifier return a missing value.

Syntax

```
CALL PRXFREE (RegEx_ID)
```

RegEx_ID: Unique RegEx identifier returned by PRXPARSE (input argument, required)

This routine is used to free up memory for a specified RegEx_ID and becomes very important for managing the memory of large programs. Remember, there is much more happening behind the scenes of the RegEx_ID construction, despite merely having a numerical identifier. (See CALL PRXDEBUG.) It can't be stressed enough that memory management can be a significant problem for large programs if not handled properly. Although SAS still handles memory cleanup to avoid memory leaks when a session ends, it is possible to run into memory limitations within a single session. Think very strategically about which RegEx_IDs—or any other variables for that matter—are necessary for each chunk of code.

Example 3.13: Releasing Memory with CALL PRXFREE

In order to demonstrate the functionality of CALL PRXFREE, we are revisiting a new version of the example code for PRXCHANGE. However, instead of printing output as in the original example, we are going to concern ourselves only with the results related to CALL PRXFREE. (See Output 3.13.)

As we can see in the code below, the PUT statement is used to print the values of Street_RXID and AddParse_RXID to the log for each run through the DATA step (creating four writes to the log). However, using the IF statement, we run the CALL PRXFREE routine on the last record to release the memory associated with both RegEx_IDs. Then, we print the results to the log. This creates a fifth write to the log, but the values are missing this time because our routine was successful at releasing the memory allocated for them—making them unrecoverable.

```

data sample;
input address $50.;
datalines;
103 Pennsylvania Ave NW, Washington, DC 20216
508 First St NW, Washington, DC 20001
650 1st St NE, Washington DC 20002
3000 K Street NW, Washington, DC 20007
;
run;

data _null_;
set sample end=last;
Street_RXID = PRXPARSE('s/\s+(S|s)\w+\s+/ St. /o');
AddParse_RXID = PRXPARSE('s/(.+?),*\s+(\w+),*\s+(\w+)\s+(\d+)/$1,
$2, $3, $4/o');
text = PRXCHANGE(Street_RXID,-1,address);
text2 = PRXCHANGE(AddParse_RXID,-1,text);
put Street_RXID AddParse_RXID;

if last then do;
CALL PRXFREE(Street_RXID);
CALL PRXFREE(AddParse_RXID);
put Street_RXID AddParse_RXID;
end;

run;

```

Output 3.13: Log Printout

```

1 2
1 2
1 2
1 2
. .

```

The missing values displayed for each of the RegEx_ID variables demonstrate that the CALL PRXFREE routine released all memory associated with both.

3.4 Summary

In this chapter, we have explored the PRX suite of functions and call routines available in SAS for implementing RegEx patterns. They collectively provide tremendous capability, enabling the advanced applications that we begin to explore in the next chapter.

As we have seen throughout the chapter, PRX functions and call routines cannot replace well-written RegEx patterns, despite providing incredible functionality. Attempting to leverage functions and call routines with poorly written RegEx patterns is like trying to drive a sports car with no fuel.

Also, while the PRX functions and call routines represent flexible, powerful capabilities to be leveraged for a wide variety of applications—basic and advanced—they often cannot stand alone. It is important to leverage them in conjunction with other elements of SAS to develop robust code; a fact that we have merely had a glimpse of in this chapter. For instance, some very advanced RegEx applications benefit from the use of MACRO programming techniques (beyond the anticipated skill level of this book).

Now, at this point, we are done. You have all the basic tools in place to make truly useful, robust SAS programs that leverage regular expressions. However, as promised from the outset, we are going to really pull everything together via a series of advanced, case-study-style examples in the next chapter.

¹ National Institutes of Standards and Technology (NIST) special publication 800-122, April 2010, Guide to Protecting the Confidentiality of Personally Identifiable Information (PII), <http://csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf>

² SAS Support, debug information: https://support.sas.com/rnd/base/datastep/perl_regex/regex.debug.html

Chapter 4: Applications of Regular Expressions in SAS

4.1 Introduction	65
4.1.1 Random PII Generator	66
4.2 Data Cleansing and Standardization	72
4.3 Information Extraction	77
4.4 Search and Replacement	80
4.5 Summary	83
4.5.1 Start Small.....	83
4.5.2 Think Big	83

4.1 Introduction

In this chapter, we explore some real-world applications of RegEx with SAS, demonstrating a wide variety of scenarios in which we can implement what you have learned thus far. The general categories that these examples have been placed under do not to imply that we are limited in what we can do (see Chapter 1), nor do they imply a lack of overlap between some of the examples.

Now, in order to execute some of the applications to follow, we need a good source of addresses, phone numbers, names, birthdates, and Social Security numbers. Obviously, these sources are hard to come by for experimentation (that is a lot of personal information!). Therefore, for the sake of practice, I created some code to randomly generate these more sensitive data items. The code for these random Personally Identifiable Information (PII) elements is presented below and is not repeated for the applicable applications. Access to real data sources is always preferable when developing robust code, but the following random generator does a fair job of inserting some commonly occurring variability into these elements. Feel free to experiment with it to obtain a different number of records or greater variability.

While all of the examples in this chapter are realistic, there is still room for improvement. However, we can do only so much in this book. So, at the end of each section, I assign you some homework—suggested assignments for how to improve the code already provided. These items should prove especially interesting for the advanced programmers among us.

4.1.1 Random PII Generator

The following code was developed to provide a set number of randomly generated elements for the purposes of the following examples. This process is an effort to replicate the kind of data we all see on a regular basis, PII, without encountering the usual privacy issues associated with it. As we will see in the code, every effort was made to make these elements feel real. However, it is worth noting that more advanced techniques (and more efficient techniques) were not employed because of the introductory nature of this text. If you're interested, use this code as a baseline to develop a more sophisticated and efficient random PII generator. Doing so is a great way to support both learning and real-world development work.

Note: All occurrences of PII shown in the coming pages were generated in a random fashion. Any resemblance to actual PII is completely coincidental.

A static snapshot of randomly generated data is used below, and it is not guaranteed to be replicated. But the parameters for any data set created by this code will be the same for any data set. Also, the code uses a few different methods for creating the various data elements. I want to demonstrate the variety of methods available to us for doing any task in SAS.

Much of this code is unavoidably long due to the steps taken to create names, addresses, and other information. Unlike the example code in the previous chapter, all code going forward is more heavily commented and diagrammed to ensure that you fully understand every element.

```
/*First, we create datasets for First and Last names*/
data FirstNames; ❶
input Firstname $20.;
/*Common First Names (male and female) in the United States*/
datalines;
JAMES
JOHN
ROBERT
MICHAEL
WILLIAM
DAVID
RICHARD
CHARLES
JOSEPH
THOMAS
CHRISTOPHER
DANIEL
PAUL
MARK
DONALD
GEORGE
KENNETH
STEVEN
EDWARD
BRIAN
```



```
RONALD  
ANTHONY  
KEVIN  
JASON  
MATTHEW  
MARY  
PATRICIA  
LINDA  
BARBARA  
ELIZABETH  
JENNIFER  
MARIA  
SUSAN  
MARGARET  
DOROTHY  
LISA  
NANCY  
KAREN  
BETTY  
HELEN  
SANDRA  
DONNA  
CAROL  
RUTH  
SHARON  
MICHELLE  
LAURA  
SARAH  
KIMBERLY  
DEBORAH  
;  
run;
```

```
data surnames;  
input Surname $20.;  
/*Common Last Names in the United States*/  
datalines;  
SMITH  
JOHNSON  
WILLIAMS  
JONES  
BROWN  
DAVIS  
MILLER  
WILSON  
MOORE  
TAYLOR  
ANDERSON  
THOMAS  
JACKSON  
WHITE
```

68 Introduction to Regular Expressions in SAS

```
HARRIS
MARTIN
THOMPSON
GARCIA
MARTINEZ
ROBINSON
CLARK
RODRIGUEZ
LEWIS
LEE
WALKER
;
run;

/*Next, we take a simple random sample of a fixed number of names*/
proc surveyselect data=firstnames method=srs n=25 ②
                  out=firstnamesSRS;
run;
proc surveyselect data=surnames method=srs n=25
                  out=surnamesSRS;
run;

/*We must create an index value to perform match-merge on later*/
data firstnamesSRS;
set firstnamesSRS;
    Num=_N_;
run;
data surnamesSRS;
set surnamesSRS;
    Num=_N_;
run;

data PII_Numbers;
n = 25; /*Determines the number of records we will create.*/

/*Arrays used for random day creation below*/③
array x x1-x12 (1:12);
array d d1-d28 (1:28);
array a a1-a30 (1:30);
array y y1-y31 (1:31);
array z z1-z20 (1974:1994);
seed=1234567890123; /*Random Number Seed Value*/
```

```

do i= 1 to n; /*Master Loop for PhoneNumber, Date of Birth, and SSN*/
  Num=i; /*Num is used as a unique index value for dataset merging
  later*/
  /*First, we randomly create the number segments*/④
  CountryCode = Strip(INT(10*rand('UNIFORM')));
  AreaCode =
Compress(INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||INT(10*rand('UN
IFORM')));
  NextThree =
Compress(INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||INT(10*rand('UN
IFORM')));
  LastFour =
Compress(INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||INT(10*rand('UN
IFORM'))||INT(10*rand('UNIFORM')));

  /*Next, we randomly create common separator types*/
  separator = rand('UNIFORM'); ⑤
  if separator >= .66 then do;
    PhoneNumber = Compress(CountryCode||'-'||AreaCode||'-'
||NextThree||'-'||LastFour);
  end;
  else if separator >=.33 AND separator <.66 then do;
    PhoneNumber = Compress(CountryCode||'
||'('||AreaCode||')'||NextThree||'-'||LastFour);
  end;
  else if separator <.33 then do;
    PhoneNumber =
Compress('+'||CountryCode||'.'||AreaCode||'.'||NextThree||'.'||LastFour);
  end;

  /*Social Security Number*/ ⑥
  SSN=Compress(INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||INT(
10*rand('UNIFORM'))||'-'||
    INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||'-'||
    INT(10*rand('UNIFORM'))||INT(10*rand('UNIFORM'))||INT(10*rand('UNIF
ORM'))||INT(10*rand('UNIFORM')));

  /*Date Of Birth*/
  call ranperk(seed, 1, of x1-x12); ⑦
  month=x1;
  if x1=2 then do;
    call ranperk(seed, 1, of d1-d28);
    day=d1;
  end;
  else if x1 in (4,6,9,11) then do;
    call ranperk(seed, 1, of a1-a30);
    day=a1;
  end;
  else if (x1=1|x1=3|x1=5|x1=7|x1=8|x1=10|x1=12) then do;

```

70 Introduction to Regular Expressions in SAS

```
        call ranperk(seed, 1, of y1-y31);
        day=y1;
        end;
        call ranperk(seed, 1, of z1-z20);
        year=z1;
        DOB=compress(month||'/'||day||'/'||year);

        output; /*OUTPUT must be made explicit within a DO LOOP*/
end;      /*The DATA step only runs once because there is no data.*/

keep Num SSN DOB PhoneNumber; /*The only elements we need for the next
step*/
run;

/*Now we extract the addresses from a file using RegEx*/
data Addresses;
infile 'F:\Introduction to Regular Expressions with
SAS\Chapter_4_Example_Source\addresses.txt' length=linelen lrecl=500 pad;
varlen=linelen-0;

input source_text $varying500. varlen; ❸
pattern = "/^(\d+?)\t(.+)/o";
pattern_ID = prxparse(pattern);
position = PRXMATCH(pattern_ID, source_text);

if PRXMATCH(pattern_ID, source_text) then do;
    Num = PRXPOSN(pattern_ID, 1, source_text) * 1;
    Address = PRXPOSN(pattern_ID, 2, source_text);
end;

keep Num Address;
run;

proc print data=addresses;
run;

/*Now, we create the PII dataset with match-merge*/ ❹
data PII;
merge firstnamesSRS surnamesSRS PII_Numbers addresses;
by num;
drop num;
run;

proc print data=PII;
run;
```

- ❶ We start with an easy way to create pseudo random names, by just creating name data sets using data lines. It is not elegant or short, but it gets the job done for our purposes.
- ❷ Here we are sampling the name data sets, using simple random sampling and a sample size of $n=25$. The sample size is completely arbitrary and chosen to match the number of other random values created later in the code.
- ❸ The arrays are created to ensure that legitimate date values can be created for our arbitrary range of years. To avoid any complications, we are ignoring leap years (no Feb 29th in the set of possibilities) and are using an arbitrary set of 4-digit years. The seed value is an arbitrary number.
- ❹ Now, we construct the phone number by using the RAND function (UNIFORM option) to generate the individual digits. The INT function takes the integer portion of a value, so multiplying the random value between 0 and 1 by 10 and applying the INT function yields a single digit between 0 and 9. This method ensures that zero values are not dropped (a leading zero would otherwise not be held). Other methods can achieve the same outcome, but this is a straightforward implementation without the need for arrays. The COMPRESS function is used to remove all spaces between the connected values. However, removing this function is an easy way to make the data messier.
- ❺ After creating the individual chunks of a phone number, we randomly assign different separator types in an effort to demonstrate the various representations that might be expected in practice.
- ❻ Next, we create Social Security numbers (SSNs) by applying the same techniques as with the phone numbers immediately above. However, we are not randomizing the separator. It is less often an issue, but you could do it as an extracurricular exercise.
- ❼ We now build the date of birth (DOB) using the arrays discussed in ❸ and the RANPERK function. This function creates random permutations of the provided arrays and provides k values from the results. Other methods could have been employed, but this is a simple approach to create random date elements within a specific range (i.e., valid dates).
- ❽ The DATA step for addresses uses some familiar RegEx functionality to extract addresses from a text file, along with the Num value that allows us to perform a match-merge in the next step.
- ❾ This final DATA step creates a single data set, PII, from the above elements.

Output 4.1 displays the final data set created by our code, `Rand_PII_Generator.sas`. As expected, it contains 25 pseudo-random PII elements to support some of our upcoming examples.

Output 4.1: Rand_PII_Generator.sas Sample Output

Obs	Firstname	Surname	PhoneNumber	SSN	DOB	Address
1	JAMES	SMITH	2-746-475-4589	539-71-9216	12/17/1986	1776 D St NW, Washington, DC 20006
2	JOHN	JOHNSON	0(439)270-9250	189-03-1020	3/25/1981	1600 Pennsylvania Ave NW, Washington, DC 20500
3	WILLIAM	WILLIAMS	+6.281.794.3626	971-45-0631	10/2/1986	600 14th St NW, Washington, DC 20005
4	DAVID	JONES	9(349)208-5935	277-05-1098	8/9/1985	1321 Pennsylvania Ave NW, Washington, DC 20004
5	THOMAS	BROWN	3(287)870-2874	123-22-9494	1/26/1980	2470 Rayburn Hob, Washington, DC 20515
6	CHRISTOPHER	DAVIS	6(639)100-7721	688-49-1392	5/20/1992	101 Independence Ave SE, Washington, DC 20540
7	DANIEL	MILLER	8(277)323-9564	675-36-2461	9/9/1977	511 10th St NW, Washington, DC 20004
8	PAUL	WILSON	6-863-034-6857	304-59-8869	5/18/1980	450 7th St NW, Washington, DC 20004
9	MARK	MOORE	5(697)801-1886	102-01-9574	7/29/1991	2 15th St NW, Washington, DC 20007
10	DONALD	TAYLOR	3(019)416-1550	439-72-6850	2/1/1982	3700 O St NW, Washington, DC 20057
11	GEORGE	ANDERSON	0(036)200-7891	116-12-3837	8/10/1983	3001 Connecticut Ave NW, Washington, DC 20008
12	KENNETH	THOMAS	1(705)466-8443	549-54-0433	5/22/1976	3101 Wisconsin Ave NW, Washington, DC 20016
13	EDWARD	JACKSON	+7.841.664.8908	461-17-1160	9/6/1977	800 Florida Ave NE, Washington, DC 20002
14	ANTHONY	WHITE	8-451-939-9401	374-43-5208	5/7/1979	1 First St NE, Washington, DC 20543
15	KEVIN	HARRIS	7-690-620-4418	877-45-2254	6/10/1984	600 Independence Ave SW, Washington, DC 20560
16	MATTHEW	MARTIN	+2.623.941.6074	436-07-9380	10/29/1975	10th St. & Constitution Ave. NW, Washington, DC 20560
17	LINDA	THOMPSON	6-564-897-1662	500-98-4809	3/10/1983	555 Pennsylvania Ave NW, Washington, DC 20001
18	MARGARET	GARCIA	5(894)411-7166	772-03-0744	10/16/1992	1000 5th Ave, New York, NY 10028
19	LISA	MARTINEZ	+1.167.511.7529	426-25-2712	8/18/1978	64th St and 5th Ave, New York, NY 10021
20	KAREN	ROBINSON	5-395-540-6931	761-09-3862	10/18/1974	10 Lincoln Center Plaza, New York, NY 10023
21	DONNA	CLARK	7-781-276-1544	131-47-7120	11/3/1991	Pier 86 W 46th St and 12th Ave, New York, NY 10036
22	CAROL	RODRIGUEZ	9(737)785-2677	280-95-9464	12/28/1985	350 5th Ave, New York, NY 10118
23	SHARON	LEWIS	0(918)040-2361	896-33-5968	1/31/1975	405 Lexington Ave, New York, NY 10174
24	MICHELLE	LEE	9(488)783-8608	735-62-9285	10/3/1984	1 Albany St, New York, NY 10006
25	KIMBERLY	WALKER	7-823-096-2389	109-02-1649	9/17/1990	1585 Broadway, New York, NY 10036

4.2 Data Cleansing and Standardization

As data sets go, the randomly generated data set that we are going to work with is fairly clean. The simple fact is we can't explore all of the ways that data can be dirty in the real world (this book would never end!). However, using some realistic data, we can test our ability to develop RegEx code to process and clean some common problems in such data sources. This exercise will prepare you to go out in the real world and tackle virtually anything you encounter because you will have all the necessary tools in your toolbox.

So, let's start by reviewing the data elements we need in order to clean and standardize, and what things we need to check for in such sources.

Firstname

The person's first name. This piece of data should contain character values only.

Surname

The person's last name. This piece of data should contain character values only.

PhoneNumber

The person's phone number. Phone numbers in different countries are written very differently, so we must be prepared to properly parse a variety of formats—especially since it is so easy for a business contact to be from or located in a different country.

SSN

The person's Social Security number. We should see only segments of numbers separated by dashes or spaces, and we need to enforce this formatting.

DOB

The person's date of birth. This can be represented in a few ways, but we primarily see the classic 8-digit format in the US. European dates represent the day before the month. This is where context is very important, because it is difficult to detect this format unless the obvious value thresholds are crossed.

Address

The person's address. This data has the most natural variability and is the most interesting to parse. We primarily need to be concerned with abbreviations, punctuation, and ZIP code lengths.

Now, as I hinted at in the last chapter, data cleansing and standardization is accomplished by creating what amount to filters. These filters are a series of RegEx functions and routines applied in succession so as to yield incremental changes as each one is applied. Implemented in the correct order, we can clean up some very messy data for later use. Fortunately for us, the data set created by the PII generator is relatively tame. But only a few things need to change in order for it to become scary data. Regardless, there are a few things that we must fix in order to make use of the entire data set in its current state. For example, when we look at observation 16 from the data set (Output 4.2), we see a few issues with the address.

Output 4.2: PII Observation 16

16	MATTHEW	MARTIN	+2.623.941.6074	436-07-9380	10/29/1975	10th St. & Constitution Ave. NW, Washington, DC 20560
----	---------	--------	-----------------	-------------	------------	---

In addition to having decimals immediately following the abbreviations for street and avenue, we see that the & symbol is used. Both these issues will become problematic when we attempt to parse the address into street, city, state, and zip.

Developing what needs to be fixed in any data set often can't be done blindly. There are basic things that we can apply to any data source, such as trimming excess spaces, and so on. However, it is advisable to pull samples of data in order to understand its quality issues before you develop RegEx patterns for cleaning.

74 Introduction to Regular Expressions in SAS

In the code below, I created a series of cleansing and standardization steps using PRXPARSE, PRXMATCH, CALL PRXCHANGE, and PRXPOSN. Notice how clean our code is by using CALL PRXCHANGE in lieu of the function version.

```
data CleanPII;
  set PII;

  ChangeAND = PRXPARSE('s/\x26/and/o');
  ChangeSTR = PRXPARSE('s/\s(St\.|St)/ Street/o');
  ChangeAVE = PRXPARSE('s/\s(Ave\.|Ave)/ Avenue/o');
  ChangeRD = PRXPARSE('s/\s(Rd\.|Rd)/ Road/o');
  ChangeDASH = PRXPARSE('s/\s*(\.|\\(|))\s*/-/o');
  ChangePLUS = PRXPARSE('s/\+//o');
  ChangeSPAC = PRXPARSE('s/ //o');

  /*Cleaning Address*/
  CALL PRXCHANGE(ChangeAND,-1,address);
  CALL PRXCHANGE(ChangeSTR,-1,address);
  CALL PRXCHANGE(ChangeAVE,-1,address);
  CALL PRXCHANGE(ChangeRD ,-1,address);

  /*Cleaning Phone Number*/
  CALL PRXCHANGE(ChangeDASH,-1,PhoneNumber);
  CALL PRXCHANGE(ChangePLUS,-1,PhoneNumber);
  CALL PRXCHANGE(ChangeSPAC,-1,PhoneNumber);

  /*Cleaning SSN*/
  CALL PRXCHANGE(ChangeDASH,-1,SSN);
  CALL PRXCHANGE(ChangeSPAC,-1,SSN);

  /*Cleaning DOB*/
  CALL PRXCHANGE(ChangeSPAC,-1,DOB);

  drop ChangeAND ChangeSTR ChangeAVE ChangeRD ChangeDASH ChangePLUS
  ChangeSPAC;
run;

data FinalPII;
  set CleanPII;
```



```

/*Parsing the address into its discrete parts*/
Addr_Pattern =
PRXPARSE('/^(\\w+(\\s\\w+)*\\s\\w+),\\s+(\\w+\\s*\\w+),\\s+(\\w+)\\s+(\\d{5}\\s*?|-
\\s*?\\d{4})|\\d{5})/o');
  if PRXMATCH(Addr_Pattern, address) then      ❸
  do;
    Street = PRXPOSN(Addr_Pattern, 1, address);
    City = PRXPOSN(Addr_Pattern, 3, address);
    State = PRXPOSN(Addr_Pattern, 4, address);
    Zip = PRXPOSN(Addr_Pattern, 5, address);
  end;
drop Addr_Pattern address;
run;

proc print data=finalpii;
run;

```

- ❶ First, we create a series of replacement RegEx pattern identifiers using PRXPARSE. We maintain the “change” naming convention to denote that each identifier represents a RegEx pattern for changing the source.
- ❷ Here we apply specific RegEx_ID’s by using the CALL PRXCHANGE routine to clean each of the variables in different ways.
- ❸ Finally, we parse the original address data field into its constituent parts: street, city, state, and zip. The PRXPOSN function grabs each piece of the address by identifying the associated capture buffer. Notice that we have to skip buffer location 2 because the second bracket set is used for logical separation inside the first bracket set (which creates buffer location 1). Referencing buffer location 2 would provide only a subset of the street information we need.

As we can see in Output 4.3, our resulting data set now has clean, standardized data in each field. Such data makes future analysis and manipulation much easier and more accurate. This exercise should serve as a nice warm-up for many other such applications.

Output 4.3: Cleaned and Standardized PII Data Set

Obs	Firstname	Surname	PhoneNumber	SSN	DOB	Street	City	State	Zip
1	JAMES	SMITH	2-746-475-4589	539-71-9216	12/17/1986	1776 D Street NW	Washington	DC	20006
2	JOHN	JOHNSON	0-439-270-9250	189-03-1020	3/25/1981	1600 Pennsylvania Avenue NW	Washington	DC	20500
3	WILLIAM	WILLIAMS	6-281-794-3626	971-45-0631	10/2/1986	600 14th Street NW	Washington	DC	20005
4	DAVID	JONES	9-349-208-5935	277-05-1098	8/9/1985	1321 Pennsylvania Avenue NW	Washington	DC	20004
5	THOMAS	BROWN	3-287-870-2874	123-22-9494	1/26/1980	2470 Rayburn Hob	Washington	DC	20515
6	CHRISTOPHER	DAVIS	6-639-100-7721	688-49-1392	5/20/1992	101 Independence Avenue SE	Washington	DC	20540
7	DANIEL	MILLER	8-277-323-9564	675-36-2461	9/9/1977	511 10th Street NW	Washington	DC	20004
8	PAUL	WILSON	6-863-034-6857	304-59-8869	5/18/1980	450 7th Street NW	Washington	DC	20004
9	MARK	MOORE	5-697-801-1886	102-01-9574	7/29/1991	2 15th Street NW	Washington	DC	20007
10	DONALD	TAYLOR	3-019-416-1550	439-72-6850	2/1/1982	3700 O Street NW	Washington	DC	20057
11	GEORGE	ANDERSON	0-036-200-7891	116-12-3837	8/10/1983	3001 Connecticut Avenue NW	Washington	DC	20008
12	KENNETH	THOMAS	1-705-466-8443	549-54-0433	5/22/1976	3101 Wisconsin Avenue NW	Washington	DC	20016
13	EDWARD	JACKSON	7-841-664-8908	461-17-1160	9/6/1977	800 Florida Avenue NE	Washington	DC	20002
14	ANTHONY	WHITE	8-451-939-9401	374-43-5208	5/7/1979	1 First Street NE	Washington	DC	20543
15	KEVIN	HARRIS	7-690-620-4418	877-45-2254	6/10/1984	600 Independence Avenue SW	Washington	DC	20560
16	MATTHEW	MARTIN	2-623-941-6074	436-07-9380	10/29/1975	10th Street and Constitution Avenue NW	Washington	DC	20560
17	LINDA	THOMPSON	6-564-897-1662	500-98-4809	3/10/1983	555 Pennsylvania Avenue NW	Washington	DC	20001
18	MARGARET	GARCIA	5-894-411-7166	772-03-0744	10/16/1992	1000 5th Avenue	New York	NY	10028
19	LISA	MARTINEZ	1-167-511-7529	426-25-2712	8/18/1978	64th Street and 5th Avenue	New York	NY	10021
20	KAREN	ROBINSON	5-395-540-6931	761-09-3862	10/18/1974	10 Lincoln Center Plaza	New York	NY	10023
21	DONNA	CLARK	7-781-276-1544	131-47-7120	11/3/1991	Pier 86 W 46th Street and 12th Avenue	New York	NY	10036
22	CAROL	RODRIGUEZ	9-737-785-2677	280-95-9464	12/28/1985	350 5th Avenue	New York	NY	10118
23	SHARON	LEWIS	0-918-040-2361	896-33-5968	1/31/1975	405 Lexington Avenue	New York	NY	10174
24	MICHELLE	LEE	9-488-783-8608	735-62-9285	10/3/1984	1 Albany Street	New York	NY	10006
25	KIMBERLY	WALKER	7-823-096-2389	109-02-1649	9/17/1990	1585 Broadway	New York	NY	10036

Homework

1. Include more standard abbreviations than the few we currently have (e.g., Parkway, Court, and so on).
2. Standardize two-digit years in the date field.
3. Create a method of handling state abbreviations with decimals.
4. Use CALL PRXFREE to clean up the RegEx_IDs used in the code.
5. Enhance the data set with a Census tract lookup using the address fields.
6. Enhance the RegEx to handle multiple spaces between words, spaces before commas, and punctuation in unexpected places.

4.3 Information Extraction

Parsing large volumes of text to generate structured data sets is a common, valuable use of RegEx capabilities. For example, we might want to collect information from a technology blog or website that contains valuable customer feedback about our product. Such information could not easily or cheaply be gathered by hand for the sake of further analysis. Due to the wide variety of possible sources from which we might need to extract information, as well as the wide array of end goals for the information, there are a number of approaches to accomplishing this task. Given the proliferation of tag-based languages such as HTML and XML, we need to be prepared to effectively extract information from them.

Now, due to the semi-structured nature of tag-based languages, they are certainly easier to process than one might anticipate. All such languages have paired opening and closing tags for defining various pieces of information. We can leverage this fact to properly dissect them and extract the information we need.

With more sophisticated techniques at our disposal (like using the SAS macro facility), we could actually “learn” the embedded data elements and extract the data associated with the discovered variables. However, emphasis on these techniques is beyond the scope of this book. Therefore, we need to know the various tags that we are looking for in the XML or HTML source in advance. We will use this approach to process the XML file in our example.

Going back to the SEC administrative proceedings example from Chapter 1, let’s parse and extract the information from the associated sample file¹.

Figure 4.1: SEC XML Sample

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <root>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61262.pdf</url>
    <release_number>34-61262</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Stephen C. Gingrich</respondents>
  </administrative_proceeding>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61256.pdf</url>
    <release_number>34-61256</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Gabelli Funds LLC</respondents>
  </administrative_proceeding>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61255.pdf</url>
    <release_number>34-61255</release_number>
    <release_date>Dec. 30, 2009</release_date>
    <respondents>Gabelli Funds LLC</respondents>
  </administrative_proceeding>
  - <administrative_proceeding>
    <url>http://www.sec.gov/litigation/admin/2009/34-61252.pdf</url>
    <release_number>34-61252</release_number>
    <release_date>Dec. 29, 2009</release_date>
    <respondents>Banc One Investment Advisors Corporation and Mark A. Beeson</respondents>
  </administrative_proceeding>
```

The primary concern is to effectively extract information from within the known XML tags that contain data, namely: `url`, `release_number`, `release_date`, and `respondents`. As you can see in the figure above, there are some other XML tags in the document, but they aren't relevant to the task at hand. For instance, `root` and `administrative_proceeding` don't contain data independent of the previously mentioned tags. They merely serve administrative functions in the context of XML for properly organizing the information for consumption by a system that reads XML directly.

```

data SECfilings;
infile 'F:\Introduction to Regular Expressions with
SAS\Chapter_4_Example_Source\administrative_proceedings_2009.xml'
length=linelen lrecl=500 pad;
varlen=linelen-0;
input source_text $varying500. varlen; ❶
format ReleaseNumber $20. ReleaseDate $20. Respondents $500. URL $500.;
start = 1;
stop = varlen;
Pattern_ID = PRXPARSE('/\<(\w+)\>(.*?)\</(\w+)\>/o'); ❷
CALL PRXNEXT(pattern_ID, start, stop, source_text, position, length); ❸
  DO WHILE (position > 0);
    tag = PRXPOSN(pattern_ID,1,source_text);
    if tag='url' then URL = PRXPOSN(pattern_ID,2,source_text);
    else if tag='release_number' then ReleaseNumber =
PRXPOSN(pattern_ID,2,source_text);
    else if tag='release_date' then ReleaseDate =
PRXPOSN(pattern_ID,2,source_text);
    else if tag='respondents' then do;
      Respondents = PRXPOSN(pattern_ID,2,source_text); ❹
      put releasenumbe releasedate respondents url;
      output;
      end;
    retain URL ReleaseNumber ReleaseDate Respondents; ❺
    CALL PRXNEXT(pattern_ID, start, stop, source_text, position,
length);
  end;
keep ReleaseNumber ReleaseDate Respondents URL;
run;
proc print data=secfilings;
run;

```

- ❶ We begin by bringing data in from our XML file source via the INFILE statement, using the length, LRECL, and pad options. (See the SAS documentation for additional information about these options.) Next, using the INPUT statement, the data at positions 1-varlen in the Program Data Vector (PDV) are assigned to `source_text`. Because we set LRECL=500, we cannot capture more than 500 bytes at one time, but we can capture less. For this reason, we use the format `$varying500`.
- ❷ Using the PRXPARSE function, we create a RegEx pattern identifier, `Pattern_ID`. This RegEx matches on a pattern that starts with an opening XML tag, contains any number and variety of characters in the middle, and ends with a closing XML tag.

- ③ Just like our example in Chapter 3, we make an initial call the CALL PRXNEXT routine to set the initial values of our outputs prior to the DO WHILE loop.
- ④ Since we are trying to build a single record to contain all four data elements, we have to condition the OUTPUT statement on the last one of these elements that occurs in the XML—which happens to be respondents.
- ⑤ The RETAIN statement must be used in order to keep all of the variable values between each occurrence of the OUTPUT statement. Otherwise, the DO loop will dump their values between each iteration.

Output 4.4: Sample of Extracted Data

Obs	ReleaseNumber	ReleaseDate	Respondents	URL
1	34-61262	Dec. 30, 2009	Stephen C. Gingrich	http://www.sec.gov/litigation/admin/2009/34-61262.pdf
2	34-61256	Dec. 30, 2009	Gabelli Funds LLC	http://www.sec.gov/litigation/admin/2009/34-61256.pdf
3	34-61255	Dec. 30, 2009	Gabelli Funds LLC	http://www.sec.gov/litigation/admin/2009/34-61255.pdf
4	34-61252	Dec. 29, 2009	Banc One Investment Advisors Corporation and Mark A. Beeson	http://www.sec.gov/litigation/admin/2009/34-61252.pdf
5	34-61247	Dec. 29, 2009	Jeffrey C. Young	http://www.sec.gov/litigation/admin/2009/34-61247.pdf
6	33-9101	Dec. 28, 2009	Bahram A. Jafari and Mountain Resources, Inc.	http://www.sec.gov/litigation/admin/2009/33-9101.pdf
7	34-61245	Dec. 28, 2009	Anna M. Baird, CPA	http://www.sec.gov/litigation/admin/2009/34-61245.pdf
8	34-61243	Dec. 28, 2009	Customer Sports, Inc., General Magic, Inc., Leonidas Films, Inc. (n/k/a Consolidated Pictures Group, Inc.), SportsPrize Entertainment, Inc., U.S. Interactive, Inc., and USA Biomass Corp.	http://www.sec.gov/litigation/admin/2009/34-61243.pdf
9	33-9100	Dec. 22, 2009	Applied Minerals, Inc. (Formerly Known as Atlas Mining Company)	http://www.sec.gov/litigation/admin/2009/33-9100.pdf
10	34-61210	Dec. 18, 2009	Invisa, Inc. and Jefferson National Life Insurance Company	http://www.sec.gov/litigation/admin/2009/34-61210.htm
11	34-61209	Dec. 18, 2009	CIHC, Inc., Conseco Services, LLC, and Conseco Equity Sales, Inc.	http://www.sec.gov/litigation/admin/2009/34-61209.htm
12	34-61208	Dec. 18, 2009	Cornerstone Capital Management, Inc., and Laura Jean Kent	http://www.sec.gov/litigation/admin/2009/34-61208.htm
13	33-9097	Dec. 18, 2009	ICAP Securities USA LLC, Ronald A. Purpora, Gregory F. Murphy, Peter M. Agola, Ronald Boccio, Kevin Cunningham, Donald E. Hoffman, Jr., and Anthony Parisi	http://www.sec.gov/litigation/admin/2009/33-9097.pdf
14	34-61199A	Dec. 17, 2009	Bear Wagner Specialists LLC, Fleet Specialist, Inc., LaBranche & Co. LLC, Spear, Leeds & Kellogg Specialists LLC, Van der Moolen Specialists USA, LLC, Performance Specialist Group LLC, and SIG Specialists, Inc.	http://www.sec.gov/litigation/admin/2009/34-61199a.pdf
15	33-9096	Dec. 17, 2009	Ernst & Young LLP	http://www.sec.gov/litigation/admin/2009/33-9096.pdf

As we can see in Output 4.4, the code effectively captured the four elements out of our XML source. This approach is generalizable to many other hierarchical file types such as HTML and should be interesting to explore.

Homework

1. Rearrange the ReleaseDate field to look like a different standard SAS date format.
2. Create a variable named Count that provides the number of Respondents on each row. This will be both fun and tricky.

3. Enhance the current RegEx pattern to force a match of a closing tag with the same name. There are two occurrences in the existing output where a formatting tag called SUB is embedded in the Respondents field. Our existing code stops on the closing tag for SUB instead of on the closing tag for `respondents`. The fix for this is not difficult, but requires more code than you might anticipate.

4.4 Search and Replacement

The specific needs for search and replace functionality can vary greatly, but nowhere is this capability more necessary than for PII redaction. Redacting PII is a frequent concern in the public sector, where information sharing between government agencies or periodic public information release is often mandated. We revisit the data from our cleansing and standardization example here since it includes the kinds of information we would likely want to redact. However, in an effort to make this more realistic, the example data set that we used in Section 4.2 has been exported to a text file. We want to know how to perform this task on any data source, from the highly structured to the completely unstructured. We have already worked with structured data sources for this technique, so exploring unstructured data sources is a natural next step.

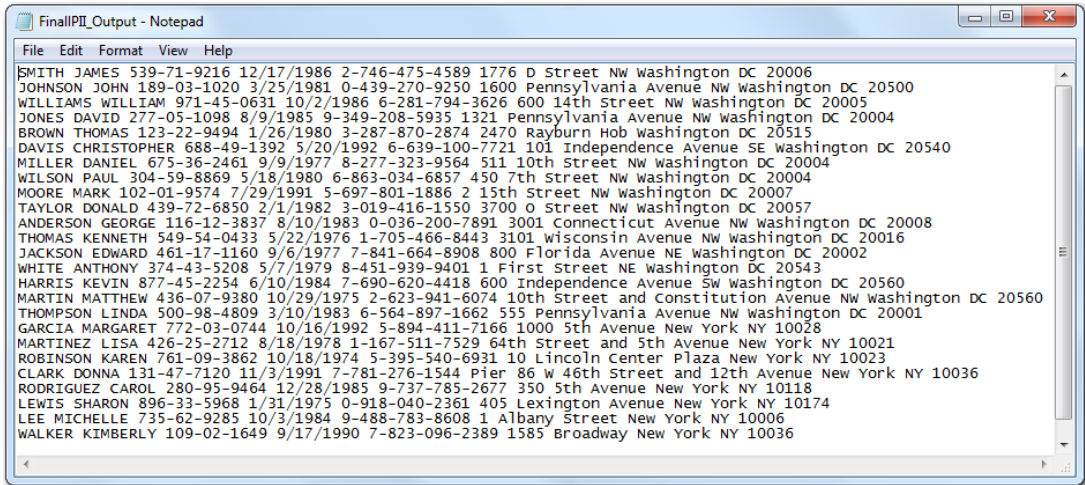
Now, before we get into how to perform the redaction, it is worth showing how the TXT file was created. Despite knowing how it is created in advance, we want to behave as though we have no knowledge of its construction in order to ensure that we are creating reasonably robust code.

You can see in the code snippet below that we simply take the resulting data set from Section 4.2, `FinalPII`, as input via the `SET` statement. Next, we use the `FILE` statement to create the TXT file reference. Once the `FILE` statement is used, the following `PUT` statement automatically writes the identified variables to it.

```
data _NULL_;
set finalpii;
file 'F:\Introduction to Regular Expressions with
SAS\Chapter_4_Example_Source\FinalPII_Output.txt';
put surname firstname ssn dob phonenumber street city state zip;
run;
```

Output 4.5 shows the output provided by this code. As we can see, the structure is largely removed, though not entirely gone. This allows us to more closely approximate what you might encounter in the real world (which could be PII stored in a Microsoft Word file).

Output 4.5: PII Raw Text



```

File Edit Format View Help
SMITH JAMES 539-71-9216 12/17/1986 2-746-475-4589 1776 D Street NW washington DC 20006
JOHNSON JOHN 189-03-1020 3/25/1981 0-439-270-9250 1600 Pennsylvania Avenue NW washington DC 20500
WILLIAMS WILLIAM 971-45-0631 10/2/1986 6-281-794-3626 600 14th Street NW washington DC 20005
JONES DAVID 277-05-1098 8/9/1985 9-349-208-5935 1321 Pennsylvania Avenue NW washington DC 20004
BROWN THOMAS 123-22-9494 1/26/1980 3-287-870-2874 2470 Rayburn Hob washington DC 20515
DAVIS CHRISTOPHER 688-49-1392 5/20/1992 6-639-100-7721 101 Independence Avenue SE washington DC 20540
MILLER DANIEL 675-36-2461 9/9/1977 8-277-323-9564 511 10th Street NW washington DC 20004
WILSON PAUL 304-59-8869 5/18/1980 6-863-034-6857 450 7th Street NW washington DC 20004
MOORE MARK 102-01-9574 7/29/1991 5-697-801-1886 2 15th Street NW washington DC 20007
TAYLOR DONALD 439-72-6850 2/1/1982 3-019-416-1550 3700 O Street NW washington DC 20057
ANDERSON GEORGE 116-12-3833 8/10/1983 0-036-200-7891 3001 Connecticut Avenue NW washington DC 20008
THOMAS KENNETH 549-54-0433 5/22/1976 1-705-466-8443 3101 wisconsin Avenue NW washington DC 20016
JACKSON EDWARD 461-17-1160 9/6/1977 7-841-664-8908 800 Florida Avenue NE washington DC 20002
WHITE ANTHONY 374-43-5208 5/7/1979 8-451-939-9401 1 First Street NE washington DC 20543
HARRIS KEVIN 877-45-2254 6/10/1984 7-690-620-4418 600 Independence Avenue SW washington DC 20560
MARTIN MATTHEW 436-07-9380 10/29/1975 2-623-941-6074 10th Street and Constitution Avenue NW washington DC 20560
THOMPSON LINDA 500-98-4809 3/10/1983 6-564-897-1662 555 Pennsylvania Avenue NW washington DC 20001
GARCIA MARGARET 772-03-0744 10/16/1992 5-894-411-7166 1000 5th Avenue New York NY 10028
MARTINEZ LISA 426-25-2712 8/18/1978 1-167-511-7529 64th Street and 5th Avenue New York NY 10021
ROBINSON KAREN 761-09-3862 10/18/1974 5-395-540-6931 10 Lincoln Center Plaza New York NY 10023
CLARK DONNA 131-47-7120 11/3/1991 7-781-276-1544 Pier 86 w 46th Street and 12th Avenue New York NY 10036
RODRIGUEZ CAROL 280-95-9464 12/28/1985 9-737-785-2677 350 5th Avenue New York NY 10118
LEWIS SHARON 896-33-5968 1/31/1975 0-918-040-2361 405 Lexington Avenue New York NY 10174
LEE MICHELLE 735-62-9285 10/3/1984 9-488-783-8608 1 Albany Street New York NY 10006
WALKER KIMBERLY 109-02-1649 9/17/1990 7-823-096-2389 1585 Broadway New York NY 10036

```

Now that we have an unstructured data source to work with, we can create the code to redact all sensitive data elements. The challenge of doing this effectively is that we can't depend on the structure of surrounding text to inform the redaction decisions of our code. For this reason, it is important that our code takes great care to ensure that we properly detect the individual elements before redacting them.

Below is the code that performs our redaction of the PII elements SSN, DOB, PhoneNumber, Street, City, and Zip. Now, many organizations are allowed to publish small amounts of information that individuals authorize in advance, such as city or phone number. However, we're focusing on how to redact all of them, because keeping a select few is easy. In addition to performing the redaction steps, we also output the redacted text to a new TXT file. Again, this is an effort to support a realistic use of these techniques. For instance, many organizations keep donor or member information in text files that are updated by administrative staff. There are valid reasons for sharing portions of that information either internally or with select external entities, but doing so must be undertaken with great care. Thus, it is useful for such files to be automatically scrubbed prior to being given a final review and then shared with others.

Note: While the following code is more realistic than what we developed in Chapter 3, it still needs to be improved for robust, real-world applications. The homework for this chapter has some suggestions, but there is always room for additional refinement.

```

data _NULL ;
infile 'F:\Introduction to Regular Expressions with
SAS\Chapter_4_Example_Source\FinallPII_Output.txt' length=linelen
lrecl=500 pad;
varlen=linelen-0;
input source_text $varying500. varlen;
Redact_SSN = PRXPARSE('s/\d{3}s*\-s*\d{2}s*\-s*\d{4}/REDACTED/o'); ❶

```

```

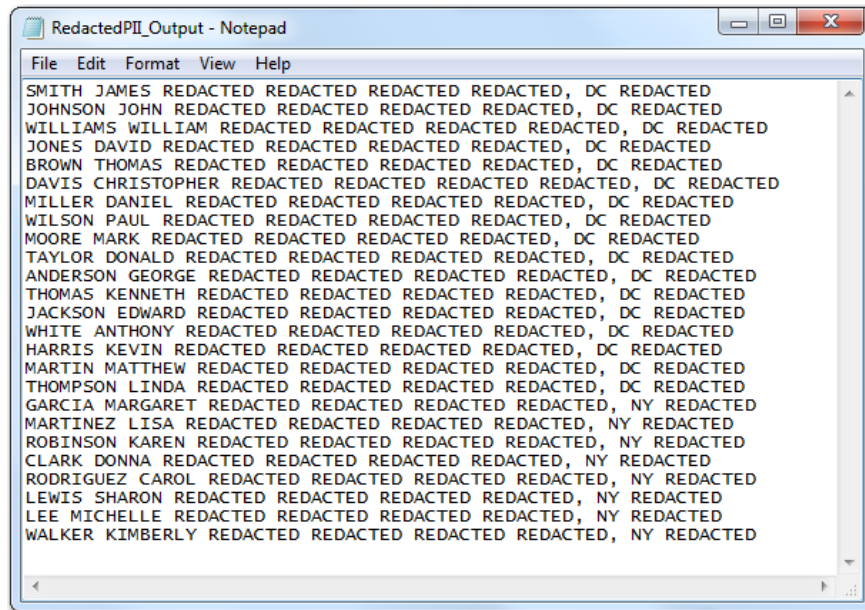
Redact_Phone = PRXPARSE('s/(\d\s*)?\s*\d{3}\s*-\s*\d{3}\s*-\s*\d{4}/REDACTED/o');
Redact_DOB = PRXPARSE('s/\d{1,2}\s*\//\s*\d{1,2}\s*\//\s*\d{4}/REDACTED/o');
Redact_Addr =
PRXPARSE('s/(\s+(\w+(\s\w+)*\s\w+)\s+(\w+\s*\w+)\s+(\w+)\s+(\d{5}\s*?-\s*?\d{4})|\d{5})/ REDACTED, $4 REDACTED/o');
CALL PRXCHANGE(Redact_Addr,-1,source_text);      ❷
CALL PRXCHANGE(Redact_SSN,-1,source_text);
CALL PRXCHANGE(Redact_Phone,-1,source_text);
CALL PRXCHANGE(Redact_DOB,-1,source_text);

file 'F:\Introduction to Regular Expressions with
SAS\Chapter_4_Example_Source\RedactedPII_Output.txt';
put source_text;                                ❸
run;

```

- ❶ We create four different RegEx_ID's associated with the different PII elements that we want to redact from our source file—SSN, PhoneNumber, DOB, Street, City, and Zip.
- ❷ We use the CALL PRXCHANGE routine to apply the four different redaction patterns in sequence.
- ❸ Using the FILE statement, we create an output TXT file for writing our resulting text changes to. Since we overwrote the original text using the CALL PRXCHANGE routine (i.e., changes were inserted back into source_text), we need to output only the original variable, source_text, with the PUT statement.

Figure 4.2: Redacted PII Data



As we can see in the resulting file of redacted output (Figure 4.2), only the individual's name and state are left. The redacting clearly worked, but in this context the resulting information might not be the most readable. How can we achieve the same goal while making the output easier to read? It's simple. Instead of inserting REDACTED we can insert "" (i.e., nothing), which effectively deletes the text. Try it out and see what happens.

Homework

1. Update the RegEx patterns to allow City to be shown (tricky with two-word names like New York).
2. Incorporate the results of Section 4.2, Homework item 5 so that the Census tract can be displayed.
3. Use the random PII generator in Section 4.1 to incorporate an entirely new field to then display this output in.
4. Make this code more robust by incorporating zero-width metacharacter concepts such as word boundaries (`\b`) to ensure that word edges are identified properly.

4.5 Summary

In this chapter, we finally put it all together with a series of examples that touch on multiple ways regular expressions can be used in SAS. Hopefully, they were each motivating, educational, and useful—and actually helped the prior chapters all make more sense.

We worked through three long-form examples—"Data Cleansing and Standardization", "Information Extraction", and "Search and Replacement"—that address commonly needed capabilities that should be generalizable to many contexts. These include, but are not limited to: law enforcement, retail, E-commerce, healthcare, finance, and defense.

At this point, you have all the tools and experience that you need in order to apply this information to highly complex problems in the real world. But before you do, here are some reminders.

4.5.1 Start Small

It can't be overstated how beneficial it is to start with a simple task and build from there. By implementing RegEx capabilities on merely a small segment of a much larger problem, you are able to more carefully build each element—ultimately ensuring more rapid progress on the overall problem with less rework. This approach also enables the beginner to develop capabilities without becoming overwhelmed, which is always important for maintaining momentum on development projects.

4.5.2 Think Big

Though you might be solving a small portion of a larger problem, you always have to think about the big picture. Always take the time to understand the context surrounding any development effort; the secondary and tertiary effects of your work can never be fully anticipated. Also, thinking about the various aspects of any project can help you to anticipate the needs of your RegEx patterns.

¹ The file was downloaded manually from the SEC website, in XML format. The original website source can be found here: <http://www.sec.gov/litigation/admin/adminarchive/adminarc2009.shtml>

Appendix A: Perl Version Notes

This appendix contains notes about the limitations of the Perl version that is used by SAS 9.4¹. It is intended for the Perl experts out there who are wondering about some of the missing pieces within SAS. The information below was taken directly from the SAS website and is provided here because it is somewhat difficult information to find but is potentially very useful to the advanced reader of this book.

“The PRX functions use a modified version of Perl 5.6.1 to perform regular expression compilation and matching. Perl is compiled into a library for use with SAS. This library is shipped with SAS® 9. The modified and original Perl 5.6.1 files are freely available in a ZIP file from the Technical Support Web site. The ZIP file is provided to comply with the Perl Artistic License and is not required in order to use the PRX functions. Each of the modified files has a comment block at the top of the file describing how and when the file was changed. The executables were given nonstandard Perl names. The standard version of Perl can be obtained from the Perl Web site.

Only Perl regular expressions are accessible from the PRX functions. Other parts of the Perl language are not accessible. The modified version of Perl [RegEx] does not support the following items:²

- All Perl variables, except for the capture buffer variables \$1 - \$n
- Metacharacters \G, \pP, \PP, and \X
- RegEx options /c and /g, and /e with substitutions
- Named characters (i.e., \N{name})
- Executing Perl code within a regular expression, which includes the syntax (?{code}), (??{code}), and (?p{code})
- Unicode pattern matching
- Pattern delimiters other than the backslash. For example: ?PATTERN?, !PATTERN!, etc.
- Perl code comments between a pattern and replacement text. For example: s{regex} #perl comment {replacement text}
- Using matching backslashes with m\\ instead of m\ to match a backslash

¹ The available Perl RegEx functionality has not changed since SAS 9.1. These notes are current as of the writing of this book. For the most up-to-date information regarding versioning, please visit the SAS documentation website at: <http://support.sas.com/documentation/>

²<http://support.sas.com/documentation/cdl/en/lefunctionsref/67239/HTML/default/viewer.htm#p0tw80fkqqpow5n1f7xwvd6bsonq.htm>

Appendix B: ASCII Code Lookup Tables

Non-Printing Characters

Binary	Hex	Dec	Oct	ASCII Abbr.	Crtl Character (Command Prompt Display)	Description
0000 0000	00	0	000	NUL	^@	Null Character
0000 0001	01	1	001	SOH	^A	Start of Header
0000 0010	02	2	002	STX	^B	Start of Text
0000 0011	03	3	003	ETX	^C	End of Text
0000 0100	04	4	004	EOT	^D	End of Transmission
0000 0101	05	5	005	ENQ	^E	Enquiry
0000 0110	06	6	006	ACK	^F	Acknowledgment
0000 0111	07	7	007	BEL	^G	Bell
0000 1000	08	8	010	BS	^H	Backspace
0000 1001	09	9	011	HT	^I	Horizontal Tab
0000 1010	0A	10	012	LF	^J	Line Feed

Binary	Hex	Dec	Oct	ASCII Abbr.	Crtl Character (Command Prompt Display)	Description
0000 1011	0B	11	013	VT	^K	Vertical Tab
0000 1100	0C	12	014	FF	^L	Form Feed
0000 1101	0D	13	015	CR	^M	Carriage Return
0000 1110	0E	14	016	SO	^N	Shift Out
0000 1111	0F	15	017	SI	^O	Shift In
0001 0000	10	16	020	DLE	^P	Data Link Escape
0001 0001	11	17	021	DC1	^Q	Device Control 1 (oft. XON)
0001 0010	12	18	022	DC2	^R	Device Control 2
0001 0011	13	19	023	DC3	^S	Device Control 3 (oft. XOFF)
0001 0100	14	20	024	DC4	^T	Device Control 4
0001 0101	15	21	025	NAK	^U	Negative Acknowledgment
0001 0110	16	22	026	SYN	^V	Synchronous Idle
0001 0111	17	23	027	ETB	^W	End of Transmission Block
0001 1000	18	24	030	CAN	^X	Cancel
0001 1001	19	25	031	EM	^Y	End of Medium

Binary	Hex	Dec	Oct	ASCII Abbr.	Crtl Character (Command Prompt Display)	Description
0001 1010	1A	26	032	SUB	^Z	Substitute
0001 1011	1B	27	033	ESC	^[Escape
0001 1100	1C	28	034	FS	^\	File Separator
0001 1101	1D	29	035	GS	^]	Group Separator
0001 1110	1E	30	036	RS	^^[j]	Record Separator
0001 1111	1F	31	037	US	^_	Unit Separator
0111 1111	7F	127	177	DEL	^?	Delete

Printing Characters

Binary	Hex	Dec	Oct	Display Character
0010 0000	20	32	040	
0010 0001	21	33	041	!
0010 0010	22	34	042	"
0010 0011	23	35	043	#
0010 0100	24	36	044	\$

Binary	Hex	Dec	Oct	Display Character
0010 0101	25	37	045	%
0010 0110	26	38	046	&
0010 0111	27	39	047	'
0010 1000	28	40	050	(
0010 1001	29	41	051)
0010 1010	2A	42	052	*
0010 1011	2B	43	053	+
0010 1100	2C	44	054	,
0010 1101	2D	45	055	-
0010 1110	2E	46	056	.
0010 1111	2F	47	057	/
0011 0000	30	48	060	0
0011 0001	31	49	061	1
0011 0010	32	50	062	2
0011 0011	33	51	063	3
0011 0100	34	52	064	4

Binary	Hex	Dec	Oct	Display Character
0011 0101	35	53	065	5
0011 0110	36	54	066	6
0011 0111	37	55	067	7
0011 1000	38	56	070	8
0011 1001	39	57	071	9
0011 1010	3A	58	072	:
0011 1011	3B	59	073	;
0011 1100	3C	60	074	<
0011 1101	3D	61	075	=
0011 1110	3E	62	076	>
0011 1111	3F	63	077	?
0100 0000	40	64	100	@
0100 0001	41	65	101	A
0100 0010	42	66	102	B
0100 0011	43	67	103	C
0100 0100	44	68	104	D

Binary	Hex	Dec	Oct	Display Character
0100 0101	45	69	105	E
0100 0110	46	70	106	F
0100 0111	47	71	107	G
0100 1000	48	72	110	H
0100 1001	49	73	111	I
0100 1010	4A	74	112	J
0100 1011	4B	75	113	K
0100 1100	4C	76	114	L
0100 1101	4D	77	115	M
0100 1110	4E	78	116	N
0100 1111	4F	79	117	O
0101 0000	50	80	120	P
0101 0001	51	81	121	Q
0101 0010	52	82	122	R
0101 0011	53	83	123	S
0101 0100	54	84	124	T

Binary	Hex	Dec	Oct	Display Character
0101 0101	55	85	125	U
0101 0110	56	86	126	V
0101 0111	57	87	127	W
0101 1000	58	88	130	X
0101 1001	59	89	131	Y
0101 1010	5A	90	132	Z
0101 1011	5B	91	133	[
0101 1100	5C	92	134	\
0101 1101	5D	93	135]
0101 1110	5E	94	136	^
0101 1111	5F	95	137	_
0110 0000	60	96	140	`
0110 0001	61	97	141	a
0110 0010	62	98	142	b
0110 0011	63	99	143	c
0110 0100	64	100	144	d

Binary	Hex	Dec	Oct	Display Character
0110 0101	65	101	145	e
0110 0110	66	102	146	f
0110 0111	67	103	147	g
0110 1000	68	104	150	h
0110 1001	69	105	151	i
0110 1010	6A	106	152	j
0110 1011	6B	107	153	k
0110 1100	6C	108	154	l
0110 1101	6D	109	155	m
0110 1110	6E	110	156	n
0110 1111	6F	111	157	o
0111 0000	70	112	160	p
0111 0001	71	113	161	q
0111 0010	72	114	162	r
0111 0011	73	115	163	s
0111 0100	74	116	164	t

Binary	Hex	Dec	Oct	Display Character
0111 0101	75	117	165	u
0111 0110	76	118	166	v
0111 0111	77	119	167	w
0111 1000	78	120	170	x
0111 1001	79	121	171	y
0111 1010	7A	122	172	z
0111 1011	7B	123	173	{
0111 1100	7C	124	174	
0111 1101	7D	125	175	}
0111 1110	7E	126	176	~

Appendix C: POSIX Metacharacters

Throughout the book, we discussed metacharacters of all types that adhere to Perl standards (de facto standard across the industry) for implementation since they are what SAS uses. And they are all that you need when you're running within the SAS environment. However, if you ever need to push the RegEx processing to a system outside of SAS, there is no guarantee that they will always work because not all systems use Perl syntax (mostly older systems don't).

Note: When you are attempting this more advanced application, know the parameters of the system you are using. You might not need to change the RegEx coding.

The exact applications of the metacharacters described in this appendix are outside the scope of this text but are provided here for the advanced reader who is interested in them. For example, although we have not covered it, POSIX metacharacters might be needed when you are performing in-database *fuzzy matching* with PROC SQL.

[:alpha:]

This metacharacter matches any alphabetic character and is equivalent to [a-zA-Z].

[:^alpha:]

This metacharacter matches any non-alphabetic character and is equivalent to [^a-zA-Z].

[:alnum:]

This metacharacter matches any alphanumeric character and is equivalent to [a-zA-Z0-9].

[:^alnum:]

This metacharacter matches any non-alphanumeric character and is equivalent to [^a-zA-Z0-9].

[:ascii:]

This metacharacter matches any ASCII character and is equivalent to [0-177] (i.e., it does not match UNICODE).

[:^ascii:]

This metacharacter matches any non-ASCII character and is equivalent to [^0-177] (i.e., it matches UNICODE).

[:blank:]

This metacharacter matches any blank character.

[:^blank:]

This metacharacter matches any non-blank character.

[:cntrl:]

This metacharacter matches any control character.

[:^cntrl:]

This metacharacter matches any non-control character.

[:digit:]

This metacharacter matches any digit character and is equivalent to `\d` or `[0-9]`.

[:^digit:]

This metacharacter matches any non-digit character and is equivalent to `\D` and `[^0-9]`.

[:graph:]

This metacharacter matches any visible character and is equivalent to `[:alnum:][:punct:]`. In other words, if you can see it when printed on a piece of paper, then it is matched by this metacharacter.

[:^graph:]

This metacharacter matches any non-printing character and is equivalent to `[^[:alnum:][:punct:]]`. If you can't see it printed on a piece of paper, then it is matched by this metacharacter.

[:lower:]

This metacharacter matches any lowercase alphabetic character and is equivalent to `[a-z]`.

[:^lower:]

This metacharacter matches anything except a lowercase alphabetic character and is equivalent to `[^a-z]`.

[:print:]

This metacharacter prints a string of characters—any characters encountered.

[:^print:]

This metacharacter does not print any characters.

[:punct:]

This metacharacter matches any visible punctuation or symbol character.

[:^punct:]

This metacharacter matches anything except visible punctuation or symbol characters.

[:space:]

This metacharacter matches any space character and is equivalent to `\s`.

[:^space:]

This metacharacter matches anything except a space character and is equivalent to `\S`.

[:upper:]

This metacharacter matches any uppercase alphabetic characters and is equivalent to `[A-Z]`.

[[:^upper:]]

This metacharacter matches all non-uppercase alphabetic characters and is equivalent to [^A-Z].

[[:word:]]

This metacharacter matches any word character encountered and is equivalent to \w.

[[:^word:]]

This metacharacter matches any non-word characters and is equivalent to \W.

[[:xdigit:]]

This metacharacter matches any hexadecimal character.

[[:^xdigit:]]

This metacharacter does not match a hexadecimal character.

Index

A

ASCII

- about 19
- code lookup tables 87–95

B

- backslash (\) 13
- backtracking 25–26
- bell (\a) metacharacters 19
- built-in call routines 49–63
- built-in functions 40–49

C

- CALL PRXCHANGE 50–54, 74–75, 81
- CALL PRXDEBUG 59–62
- CALL PRXFREE 62–63
- CALL PRXNET 57–59
- CALL PRXNEXT 79
- CALL PRXPOSN 54–55
- CALL PRXSUBSTR 56–57
- CALL routine 12
- capture buffers
 - about 39–40
 - extracting data with 46–47
 - identifying 48–49
 - using 45
- case modifiers 23–25
- case sensitivity, of metacharacters 15
- character classes 21–22
- cleansing data 72–76
- CLOSE statement 61
- compile once (*//o*) option 33–34
- COMPRESS function 71
- context-specific algorithm development 55
- control (cA-cZ) metacharacters 20

D

data

- cleansing 72–76
- extracting with capture buffers 46–47
- redacting sensitive 51–52
- standardizing 44–45, 72–76
- transforming 51
- data enrichment 5–7
- data manipulation 4–5
- DATALINES statement 12
- debugging
 - information printed to log 60–62
 - PRXPARSE function 60
- digit (d) metacharacters 17–18
- DO WHILE loop 58, 79
- dot character (.) 32

E

- ELSE tag 11
- end of line (\$) metacharacter 35
- END tag 11, 61
- escape character 14
- examples
 - data enrichment 5–7
 - data manipulation 4–5
 - Extract, Transform, and Load (ETL) 3–4
- Extract, Transform, and Load (ETL) 3–4
- extracting
 - data with capture buffers 46–47
 - information 56–57, 77–80

F

- FILE statement 80, 81
- forward slash (/) 13
- functions, 40–49
 - See also specific functions*
- fuzzy matching 97

G

GOTO tag 11
 greedy 0 or 1 time (?) modifier 26–27
 greedy 0 or more (*) modifier 26
 greedy 1 or more (+) modifier 26
 greedy n or more ({n,}) modifier 27–28
 greedy n times ({n}) modifier 27
 greedy n to m times ({n,m}) modifier 28
 greedy repetition modifiers 25–26

H

hexadecimal (\xdd) metacharacters 21
 hexadecimal number system 38
 HTML 77

I

IF statement 11, 42, 46–47, 49
 ignore case (/i) option 32
 INFILE statement 78
 information
 debugging 60–62
 extracting 56–57, 77–80
 INPUT statement 12, 78
 INT function 71

L

lazy 0 or 1 times (??) modifier 30
 lazy 0 or more (*?) modifier 28–29
 lazy 1 or more (+?) modifier 29
 lazy n or more ({n,}?) modifier 31
 lazy n times ({m}?) modifier 30
 lazy n to m times ({n,m}?) modifier 31
 lazy repetition modifiers 28–31
 list ([...]) metacharacter 21–22
 lowercase (\l) metacharacter 23
 lowercase range (\L...\E) metacharacter 24

M

memory, releasing with CALL PRXFREE 62–63
 metacharacters
 about 10–11, 15
 bell (\a) 19

 case sensitivity of 15
 control (\cA-\cZ) 20
 digit (\d) 17–18
 end of line (\$) 35
 hexadecimal (\xdd) 21
 list ([...]) 21–22
 lowercase (\l) 23
 lowercase range (\L...\E) 24
 newline (\n) 18–19
 non-digit (\D) 18
 non-whitespace (\S) 17
 non-word (\W) 16
 non-word boundary (\B) 36
 not list ([^...]) 22
 octal (\ddd) 20
 POSIX 97–99
 quote range (\Q...\E) 25
 range ([...-...]) 22
 start of line (^) 35
 string start (\A) 36–37
 tab (\t) 16–17
 uppercase (\u) 24
 uppercase range (\U...\E) 24–25
 whitespace (\s) 10–11, 17, 26
 word (\w) 15–16
 word boundary (\b) 35–36
 zero-width 34–37

modifiers

 case 23–25
 greedy repetition 25–26
 lazy repetition 28–31
 repetition 25–31
 multiline (/m) option 33

N

newline (\n) metacharacter 15, 18–19
 non-digit (\D) metacharacter 18
 non-printing characters, ASCII codes for 87–89
 non-whitespace (\S) metacharacters 17
 non-word boundary (\B) metacharacter 36
 non-word (\W) metacharacters 16
 not list ([^...]) metacharacter 22

O

octal (\ddd) metacharacters 20
 octal number system 38n1
 OPEN statement 61
 options 32–34
 OUTPUT statement 79

P

parentheses () 13
 pattern processing 11
 patterns
 defining with PRXPARSE function 41
 matching multiple times per line 58–59
 period (.) 15
 Perl
 about 2
 escape character 14
 version notes 85
 Personally Identifiable Information (PII) 51, 65
 POSIX metacharacters 97–99
 PRINT procedure 47, 53, 54
 printing characters, ASCII codes for 89–95
 PRX (Perl-Regular-eXpressions) 40
 PRXCHANGE function 39–40, 43–45
 PRXMATCH function 42–43, 74–75
 PRXPAREN function 39–40, 47–49
 PRXPARSE function 40–41, 60, 74–75, 78
 PRXPOSN function 39–40, 46–47, 74–75
 PUT statement 80

Q

question mark (?) 28
 quote range (\Q...\E) metacharacter 25

R

RAND function 71
 random PII generator 66–72
 range ([...-...]) metacharacter 22
 RANPERK function 71
 redacting sensitive data 51–52
 regular expressions (RegEx)
 about 2, 10–11
 applications of 65–84

character classes 21–22
 metacharacters 15–21
 modifiers 23–25
 options 32–34
 special characters 13–14
 test code 11–13
 using in SAS 39–64
 zero-width metacharacters 34–37

repetition modifiers 25–31
 replacement and search 80–83
 results, inserting 52–54
 RETAIN statement 12, 33–34, 79

S**SAS**

built-in call routines 49–63
 built-in functions 40–49
 CALL PRXCHANGE 50–54
 CALL PRXDEBUG 59–62
 CALL PRXFREE 62–63
 CALL PRXNET 57–59
 CALL PRXPOSN 54–55
 CALL PRXSUBSTR 56–57
 capture buffer 39–40
 PRXCHANGE function 43–45
 PRXMATCH function 42–43
 PRXPAREN function 47–49
 PRXPARSE function 40–41
 PRXPOSN function 46–47
 using regular expressions in 39–64
 website 5
 search and replacement 80–83
 SET statement 80
 single line (/s) option 32
 slashes (/) 34
 source text, finding strings in with PRXMATCH
 function 42–43
 special characters 11, 13–14
 SQL procedure 97
 square brackets ([]) 21
 standardizing data 44–45, 72–76
 start of line (^) metacharacter 35
 START tag 11

string start (\A) metacharacter 36–37
strings, finding in source text with PRXMATCH
 function 42–43
substitution (s/) operator 34

T

tab (\t) metacharacters 16–17
test code, for regular expressions (RegEx) 11–13
THEN tag 11
transforming data 51

U

uppercase (\u) metacharacter 24
uppercase range (\U...\E) metacharacter 24–25

V

vertical bar (|) 13

W

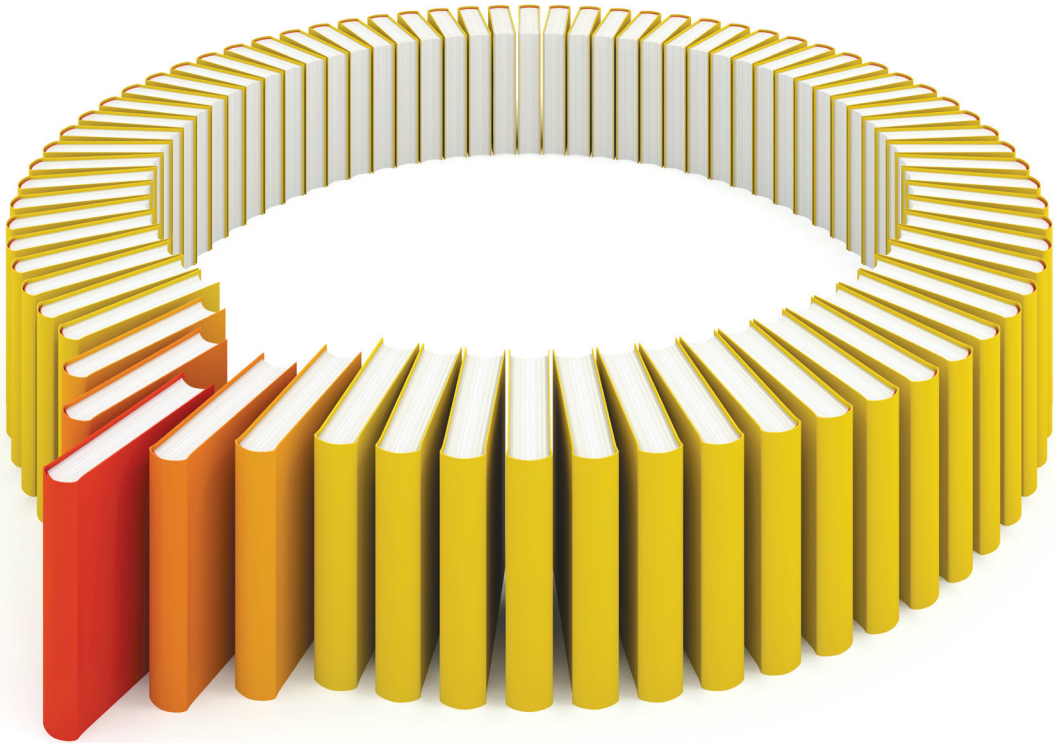
whitespace (\s) metacharacters 10–11, 17, 26
wildcard metacharacter 15
word boundary (\b) metacharacter 35–36
word (\w) metacharacters 15–16

X

XML 77

Z

zero-width metacharacters 34–37



Gain Greater Insight into Your SAS[®] Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

 support.sas.com/bookstore
for additional books and resources.


THE POWER TO KNOW.®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0413

