Natalie Carlson

Kris Carroll

CSCE A331 Spring 2019

Due Friday May 3

Perl

History and Overview

Perl (Practical Extraction and Report Language) was created by Larry Wall, and introduced in 1987. Wall developed Perl as a text processing language for Unix-like operating systems and within five years of its release, Perl was considered a standard Unix language. At the time, Wall was aiming to create a scripting language designed to make report processing easier and more efficient, offering support for complex data structures and OS features.

Since its creation, Perl has gone through a number of changes, leading to the most recent iteration known as Perl 5, maintained to standards released by The Perl Foundation. Though another branch, called Perl 6, was originally developed in 2000, the language ultimately became recognized as its own sister programming language and development and support for Perl 5 continues today through a different development team.

Since Perl's early days as a shell scripting language, the language has grown and been adapted to a range of different purposes, most notably for system

administration and network programming, often noted for its ubiquity amongst server-side programming in the early days of the Web.

Perl's primary structure was derived from C with a target of functioning within a shell programming environment. As such, it features a number of constructs and similarities to both, such as the structure of variables, assignment statements, flow of control constructs from C and syntax common to shell scripting such as leading sigils to all variables that allow for proper conversion of the associated expression evaluation to a string.

After its initial creation, Perl has continued to adopt widely from other programming languages including adding other features and data structures to the language. Lists were implemented according to those found in Lisp, while support for regular expressions was derived from sed. With the release of Perl 5, the language added support for more complex data structures, first-class functions, and support for object-oriented programming through implementation of references, lexically scoped variables and packaging of code to be used as modules elsewhere.

Type System

Perl's type system frequently blurs the lines between major categories and distinctions made in programming language typing. At the highest level, Perl features three primitive types in scalars, arrays and hashes. Each of these primitive types is indicated by its preceding sigil in variable declaration with {$} representing scalar-type values, {@} representing array or list type values, and {%} representing hash type values. These type primitives are statically enforced, using

mismatching values or operations assigned to sigiled identifiers will cause a compilation error.

Each of Perl's primitive types feature a range of accepted values, such as the scalars which can be integers, floating point numbers, strings, or objects. Type checking for values and operations beyond the highest level primitive form occurs dynamically at runtime and assignment statements changing the inferred type of a variable is allowed. Common operators are defined according to their primitive type with scalars, hashes and arrays each following a general pattern or trend behind operator function and implementation. Additionally, basic functions such as determining lengths and memory management operations are defined according to their primitive types and use of these functions on values of other types is usually type checked at compile time (statically). Support for crossover uses of these types has been added in later versions of Perl, such as the use of a variable assigned with the array {@} sigil in a manner where a scalar value is expected resulting in giving the length of the array in scalar form.

With the introduction of lexically scoped variables, use of the strict pragma will throw type exceptions at compile time if variables have not been assigned to an appropriate lexical scope. This is done through the use of the {my} and {our} keywords, specifying whether or not the variable is expected to be globally accessible or limited in scope to a particular block of code in which it was declared. This process is done statically and allows the program to manage memory according to the lifetime of a variable as defined by its scope.

Composing and Executing Code

Writing and running Perl source code is a relatively simple process on most Unix-based systems, as Perl is already pre-installed on many of them and found in the user path. The first line of a source code file can utilize shebang notation such as {#!/usr/bin/perl} to direct the bash shell to the specified perl interpreter, which it then passes the remaining portions of the script to.

Pragmas can be included to inform the Perl interpreter with various compiling directives, such as {use strict;} which enforces the strictest compiling rules or {use warnings;} to increase run-time warnings. Afterwards, importing modules can be done further using the keyword {use} followed by the module name to be included.

Afterwards, the program's statements can be made, the file can be saved with a .pl extension, and executing the program is as simple as passing the file to the Perl interpreter via the command line as in {perl abc.pl}. The interpreter will first compile the code to bytecode and then read the bytecode and perform the necessary operations. If the shebang line was included as the first line of the program, the file itself becomes an executable that can be run directly from the command line with {./abc.pl}.

Perl has often been referred to as the 'Swiss Army chainsaw' of scripting languages. Like natural languages, there are usually more than one way to say the same thing. Most syntactic constructs in Perl can be re-stated two or more ways with the exact same result. This in part is what makes Perl so flexible.

Single line comments in Perl are executed with the hashtag symbol {#}. There are no built in multiline comments, however, the flexibility of Perl has allowed some creative ways to create them. Like many other languages, statements must end with a semicolon, but they can be split onto multiple lines or multiple statements can be on the same line.

Perl is case sensitive, hence when creating variables, myVariable, myvariable, MyVariable, and Myvariable are all considered distinct. Uninitialized variables have the special value of {undef}. Execution of an undefined variable result in an empty string or 0, similar to the default in Java. Additionally, variables with a current value can be "cleared" by assigning {undef} to the variable. Applying the function {defined()} to a variable will output a 1 if a true value exists. Evaluation of a Boolean expression into false returns a special value that represents nothing and is implicitly converted into an appropriate type according to the context in which it is used. If used in a string context, it is converted into the empty string ("") while if used in a numeric context, it is converted the numeric 0. It may also be converted to {undef} if appropriate in the context. Similarly, a range of different type values are converted to this nothing value if used in a Boolean expression, such as all forms of the numeric 0 (including 00, 000, and 0.0), {undef}, the empty string, and the string "0" containing a single 0. Conversely, all other values of these types are considered true when used in a Boolean context (i.e. - the numeric 1, the string "1", and even the string "false").

As mentioned earlier, Perl's basic data type is a scalar variable, which can be strings or numbers. Strings can be enclosed with single or double quotes. Single

quoted strings follow a ‘what you see is what you get’ format, with a few exceptions. An interesting feature of single quoted strings are that newline characters are hard coded into the string. For example consider the following:

Statements in c++:

{ “Line one\nLine two.” }	{“The end”<<endl;}

Statement in Perl:

{‘Line one Line two.’}	{‘The end ;}

Numeric literals in Perl can be in integer or floating point form. For readability purposes, large numbers can written as 9_999_999, inserting underscores to visually separate places. Perl will ignore these underscores during computations and remove them for output.

As Perl is based on UNIX systems, input and output are considered files. Input is retrieved using {<STDIN>} and can be directly assigned as a value to a scalar variable or array. Unlike many standard languages, the input includes the succeeding newline character; the built in function {chomp()} will remove this. Output can be executed through the print function. A unique feature of the print function is that scalar variables can be inserted within a string, i.e. {print “Hello my name is $myName, and I am $myAge years old.”;}.

Special/Interesting Features

Arrays

The main data structure in Perl is an array. Semantically they are close to a list in Lisp, however, the syntax is more closely related to an array in C. Like a list, arrays in Perl are dynamically allocated, meaning the can shrink and grow according to the programmers needs. Additionally, like most arrays, elements can be accessed directly by index number with no need to loop through from the beginning. This design gives the programmer a lot of advantages. Another flexible feature regarding Perl arrays is that they are lists of scalars, and since there is no distinction made between a scalar number and a scalar string, arrays can contain a mixture of both, furthermore, arrays can also include other arrays. The naming convention of an array variable is the same as a scalar variable except the {$} symbol is replaced with an {@}.

When an array index is initialized a special scalar variable is automatically created, {$arrayName[i]}, where i is the index number. Furthermore, a scalar variable, {$#arrayName} is also automatically created, which hold the subscript of the last element in the array and the length of the array is {$#arrayName + 1}. You can resize the array by altering the value of {$#arrayName} and any elements at indexes greater than the new value will be truncated. You can completely clear an array by setting this variable to -1 or ( ). To access a group of elements in an array you can uses slices such as {@arrayName[i..j]} (elements at index i to j, where j could be {$#arrayName} for the last element), or {@arrayName[i, k]}, where i and k are specific indexes. Finally, arrays can also be implemented as Stacks and as Queues with built in features for both.

Regular Expressions

Perl natively supports regular expressions and operations involving them. This is important to its design of processing large amounts of data and reports, allowing the programmer to use regular expressions to guide pattern searching within a specified text. Creating a regular expression is as simple as specifying the pattern you're searching for between forward slashes {/}. Brackets {[]} can be used to specify optional portions of the pattern, {^} can be used for negation or exclusion of something in the pattern, as well as a number of escape characters that specify special parts of a pattern such as a "full stop" meaning the pattern exists only at the end of the searched string. Many common regular expressions have been included as a built-in shortcut, such as {\d} for searching for digits or {\s} for a whitespace character as well as their negative forms {\D} for anything not a digit and {\S} for anything not a whitespace character respectively.

The regular expression implementation is robust, fully-featured and can perform pattern analysis and matching dynamically at runtime, even applied to file or input streams as data is being received. This is mostly expected, as the language was designed with report and data processing as the core focus.

Subroutines

Perl features subroutines, or "subs", as much the same construct and purpose as functions in C and other languages. Subroutines are used to designate specific portions of code that are expected to be used again, assigning them a name that can be referenced later in the code to perform their function, and allowing us to pass values as parameters and receive values from sections of code as a result. The naming convention behind calling them subroutines as opposed to functions is that

The Perl Foundation felt a need to distinguish between the operations of native functions and operators from user-defined groups of code.

Perl subroutines were designed to have much the same functionality as functions are they are found in C-based languages, providing a construct with which to assign an identifier to sections of code and allowing us to pass and receive values from these sections. Forwards declarations were made available to allow us to define subroutines in any order we wish, regardless of if calls to the subroutines appear before the definitions of the subroutines is found. This is done in much the same way as it is in C, with a simple {sub subroutineName;} line in the earlier portions of the code as a signifier to the compiler that calls to this name are expected and that a definition can be found somewhere within the code. This prevents compiler and runtime errors when writing code referencing these subroutines and allows us to organize how the code layout is designed.

Perl's subroutines can be passed arguments on their call, however the types and number of these arguments cannot be specified. Instead, items passed as parameters within the parentheses following a subroutine call are placed on a list of command line arguments that can be retrieved individually and processed within the subroutine. As there is no way to enforce the types or number of these arguments passed, documentation of the subroutines is necessary in order to ensure proper operation by other users.

Object-Oriented Programming in Perl

Object-oriented programming support largely came about with the implementation of Perl 5. Changes that arrived with this version allowed for the declaration and definition of objects with programmer-specified types. This is done through a class mentality, with a class representing a package of Perl code that contains all of the necessary methods to create and work with these objects. Classes are declared through the declaration {package ClassName;} and scope of the classes is continuous until the end of the file or until another class declaration is encountered. Class object creation appears similar to other major programming languages with a line {$object = new ClassName(parameters);}, seemingly using a constructor as it is seen in Java or C++. However, the actual operation of this line actually calls a subroutine of the class called “new” and passes an arbitrary number of command line parameters that can then be utilized by the class. In this example, the subroutine new is called, which then uses the {shift} operator to extract the command line arguments that were passed to the subroutine, starting with first the ClassName, then the items found within the parentheses labelled as “parameters”, one at a time. Perl classes do not enforce types of parameters explicitly and thus documentation of the correct order of passed arguments is important for using classes, especially those created by someone other than the programmer using them.

Similar to Python and other largely interpreted, dynamic languages, Perl does not feature any data security features behind class implementation. Variables cannot be declared as private to the class though Perl documentation and best practices still adhere to the idea of using helper methods to interact with class data members (getters and setters). Class data members can, however, be accessed directly and

thus providing any methods needed to interact with the data members expected to be manipulated in a class has high priority.

Inheritance in Perl is managed by a special array known as @ISA (named after the words “is a” as in “An Employee is a Person”. Inheritance is handled by first searching the class for the specified subroutine or variable, if not found, then the @ISA array is consulted for the hierarchical order of classes and Perl continues searching through the subsequent parent classes until the identifier is found and then the appropriate code is executed. In the example of an Employee inheriting from Person, this is done by declaring the Employee class, and then defining the parent class using the line {our @ISA = qw(Person)}. This specifies the Person class within the Employee class’s @ISA array for Perl to reference when searching for identifiers called in relation to an Employee object. Due to the manner and order of Perl’s searching, this allows for overriding of inherited subroutines simply through redefinition of the subroutine using the same name within the derived class. A particularly important case of this behavior can be seen via the definition of the subroutine AUTOLOAD, which is a core Perl subroutine that is called when an identifier cannot be found while searching within a class. This allows the programmer to define the searching behavior for various identifiers directly as well as extraction and manipulation of the name of the missing subroutine. This is frequently used for error handling to output appropriate error messages for undefined subroutine calls.

Additionally, Perl classes feature another default subroutine called DESTROY which is utilized in memory management of class objects. This subroutine can be

overridden to define the appropriate behaviors for freeing memory when the class object is removed.

CGI Operations in Perl

As Perl quickly rose to meet the need of a wide range of web-based applications, Perl utilizes CGI, or Common Gateway Interface, standards to guide interactions with a web server that is operating under the CGI specifications. These specifications are regulated by the NCSA and are used to specify the interface for interacting with information servers. This allows Perl scripts to easily fit into any spaces between two information servers such as a web server and a database. Through these defined standards, Perl has built-in support for both sending and receiving data to the web server. This is done when the web server requests data, which it does through interacting with Perl scripts, which then retrieve and format the data from a database server and send the resulting data back to the web server.

Through the CGI specifications, Perl scripts are able to appropriately format and convert data for use in HTML and can even specify the HTML format of the resulting information being sent back to the web server. The CGI specifications allow Perl to define and standardize various data members and functions that are required by the HTTP protocols utilized throughout the web. These specifications also allow Perl scripts to receive information from the web according to a defined pattern, allowing for efficient interactivity between user client servers and back-end servers.

Conclusion

Perl has been created and developed by carefully assessing what works in other programming languages, giving it unique flexibility and versatility. It can be used to easily write simple code by beginner programmers or extensive programs by experienced veterans. Designed specifically for text processing, Perl is an ideal language choice for web development and the database interface makes it ideal for database management systems. Overall it is a powerful programming language that would be a good addition to the programmers arsenal.

Sources:

http://www.ebb.org/PickingUpPerl

https://www.cs.uaf.edu/users/chappell/public_html/class/2019_spr/csce_a331/lect/cs331-20190128-cat_dynamic.pdf

https://learn.perl.org/books/beginning-perl/

https://perldoc.perl.org/perlintro.html

https://www.perl.com