Asignación Práctica 1

Javier Falcón (2016-5265)

1 Ejercicio 1

a) **Eiffel**
El lenguaje de programación Eiffel surgió en el año 1985, desarrollado por Eiffel Software y diseñado por Bertrand Meyer. Eiffel es un lenguaje que sigue principalmente el paradigma de la orientación a objetos, utilizando la *clase* como la unidad de descomposición. Entre sus características resaltan la gestión de memoria automática, la herencia entre clases, tipado estático y, tal vez su contribución más importante, el diseño por contrato. Eiffel estuvo influenciado por lenguajes como Ada y Simula. Simula fue el primer lenguaje que incluyó el concepto de *clase*. Por otro lado, Eiffel ha sido influencia en lenguajes muy conocidos como *Java* y *C#* en su inclusión del *garbage collector*, por ejemplo.

b) **Perl**
Perl es un lenguaje conocido por hacer el procesamiento de texto más sencillo. Fue concebido por Larry Wall en 1987. Es un lenguaje de scripting muy poderoso y aclamado por su versatilidad, al funcionar por módulos que agregan capacidades al lenguaje. Perl toma funcionalidades de lenguajes como *C, Unix Shell, AWK* y *sed*; como por ejemplo los hashes (*AWK*) y las expresiones regulares de *sed*. También tiene un gestor automático de la memoria y forma parte de los lenguajes interpretados. Perl ha influenciado a lenguajes de scripting como *Python*, *JavaScript* y *PowerShell*.

c) **Python**
Python es un lenguaje multiparadigma que apareció en 1991 gracias al holandés Guido van Rossum. Está pensado para trabajar con los paradigmas de Orientación a Objetos, paradigma imperativo y funcional. Entre sus características tenemos que es un lenguaje interpretado, utiliza tipado dinámico, el diseño por contrato y es multiplataforma. Python fue influenciado extensivamente por el lenguaje ABC, tomando cualidades como la indentación para representar bloques en lugar de las llaves. Continuando con influencias, Python ha logrado ser inspiración para lenguajes como ECMAScript y Cobra, los cuales han tomado características como los iteradores y el sistema de indentación.

2 Ejercicio 2

El uso del formato fijo o el formato libre siempre da de qué hablar dentro de la comunidad de programadores. En lo personal, solo he trabajado con lenguajes que utilizan el formato libre, sin embargo puedo entender las facilidades de cada uno. Se dice que el formato fijo fue pensado para mejorar la legibilidad y la facilidad para escribir código.Esto puede ser cierto en algunos aspectos, como por ejemplo la imposibilidad de escribir varias sentencias en una misma línea. Como el formato fijo toma una sentencia cuando exista un final de línea, será imposible colocar varias sentencias en una sola; un hábito que es común en muchos programadores. Adicionalmente, puede que mejore la legibilidad del código debido a que no son necesarias las llaves para representar bloques y demás símbolos que puedan empañar la lectura, así como también la obligación de indentar hace que todo quede bien estructurado. Para los programadores nuevos, que apenas se introducen en el mundo de la computación, el formato libre parece ser mejor para ellos debido a que casi siempre tienden a olvidar colocar el caracter de terminación en las sentencias. Sin embargo, este formato puede ser incómodo al momento de continuar una sentencia en líneas separadas, teniendo que utilizar un caracter especial par indicar la continuación.
Por otro lado, el formato libre veo que tiene un parecido más natural al idioma en el que se suele programar: inglés. Al escribir textos utilizamos el ; para separar diferentes sentencias, por lo que parecería ser más natural para nosotros terminar cada *statement* utilizando un símbolo de terminación explícito, lo cual genera menos ambigüedad.

3 Ejercicio 3

a) Gramática regular a la derecha
$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$
$num \rightarrow digit\ num|digit$
$subsop \rightarrow -|\epsilon$
$exp \rightarrow E|e$
$point \rightarrow .$
$SNot \rightarrow digit\ point\ num\ exp\ subsop\ num$

b) Expresión regular
$digit[0-9]$
$((" - "?digit+)"."(digit+)(e|E)(" - ")?(digit+))$

c) Autómata Finito Determinístico

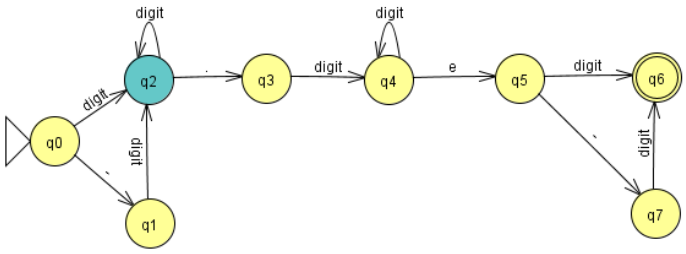

Figure 1: Autómata Finito Determinístico para un número de punto flotante

En mi caso, empecé creando la expresión regular y el AFD. La razón de esto viene dado a que cuando se realizan compiladores se hace primero la parte del análisis léxico, en el cual primero se toma una expresión regular para poder generar un autómata que pueda representarse en código. Finalmente, fue la parte de la gramática o, más bien, de lo que serían sus reglas de producción, ya que éstas son las necesarias para el siguiente paso de compilación (sintaxis).

4 Ejercicio 4

- **Paradigma imperativo**
  Un problema adecuado para este paradigma podría ser el de realizar algoritmos de ordenamiento. Este tipo de algoritmos requiere de constantes cambios de estado mediante asignaciones, manejo de memoria y del uso de estructuras de control. Por estas características considero que es un buen caso para los lenguajes imperativos. Si bien es cierto que podrían realizarse en lenguajes funcionales, seguramente sería mucho más complicado de codificar y no sería tan eficiente.

- **Paradigma de Orientación a Objetos**
  La programación de videojuegos es un problema en el cual la orientación a objetos sería provechosa. En el entorno de los videojuegos se manejan muchas entidades, que con OOP se lograría el nivel de abstracción deseado entre ellas. Muchas de estas entidades tienen métodos comunes, por lo que se puede plantear el uso de la sobreescritura de ellos y herencia.

- **Paradigma Funcional**
  El manejo de concurrencia es un problema en el cual se hace conveniente el uso de lenguajes con paradigma funcional. Estos lenguajes no dependen

de un estado global o compartido, por lo que la concurrencia es más segura y sencilla. Además, al estar basado en funciones, no produce efectos secundarios, los valores de la función dependen únicamente de sus argumentos.

- **Paradigma Declarativo**
  La creación de documentos de cualquier tipo es un problema que encaja con el paradigma declarativo. Cuando estamos en la creación de un documento nos importa que tenga un formato y que podamos acomodar los elementos dentro del documento según lo necesitamos. Por eso el paradigma declarativo es conveniente, se necesita decir qué se mostrará en el documento más no cómo hacer que se muestre.

- **Paradigma visual**
  La construcción de autómatas se ajusta a lo que se puede lograr con el paradigma visual. Diseñar autómatas se convierte en una tarea sencilla cuando se pueden manejar a través de elementos visuales, ya que su construcción se basa en estados y transiciones. Es cierto que lo mismo se puede lograr en otro paradigma, pero se necesitaría de un nivel de experticia mayor.

References

[1] Balakrishnan, A. (2018). What is Perl? How relevant it is and how to get started!. [online] Medium. Available at: https://medium.com/@akbgunner4ever/what-is-perl-how-relevant-it-is-and-how-to-get-started-d802e7aba2cd

[2] Biancuzzi, F. and Warden, S. (2009). Masterminds of Programming. Sebastopol: O'Reilly Media, Inc.

[3] Eiffel Software. (2014, April 22). Eiffel Language. Retrieved from https://www.eiffel.com/resources/faqs/eiffel-language/classic-how

[4] Computernostalgia.net. (n.d.). Eiffel object-oriented programming language, robust software.[online] Available at: http://www.computernostalgia.net/articles/EiffelProgrammingLanguage.htm

[5] Novák, M. (2018). Imperative versus declarative code... what's the difference?. [online] Medium. Available at: https://medium.com/front-end-weekly/imperative-versus-declarative-code-whats-the-difference-adc7dd6c8380

[6] Van Roy, P. (n.d.). Programming Paradigms for Dummies: What Every Programmer Should Know. [pdf] Peter Van Roy. Available at: http://hiperc.buffalostate.edu/courses/ACM612-F15/uploads/ACM612/VanRoy-Programming.pdf