Lenguajes de Programación 2019-1*
Nota de clase 1: Introducción

Favio E. Miranda Perea Lourdes Del Carmen González Huesca

Facultad de Ciencias UNAM

1. ¿Por qué estudiar lenguajes de programación?

El siguiente comentario es una adaptación del texto original en inglés de Robert Harper[1].

Nuestro curso trata acerca de principios de lenguajes de programación, los cuales estudiamos debido a que son fundamentales para el diseño, implementación y aplicación de lenguajes de programación.

El diseño de un lenguaje de programación se considera frecuentemente como una cuestión mayoritariamente de opinión por parte de los diseñadores con casi ningún principio organizatorio. Docenas de lenguajes se usan diariamente en laboratorios de investigación y en la industria, cada uno tiene sus seguidores y sus detractores. Los méritos relativos de cada lenguaje se debaten eternamente, pero siempre sin llegar a un resultado concluyente. Algunos sugerirían que todos los lenguajes son equivalentes, siendo su única diferencia un asunto de gusto personal. Sin embargo resulta obvio que la elección de un lenguaje de programación particular es importante.

¿Podemos decir realmente que Java es mejor o peor que C++? o que ¿Scheme es mejor que Lisp, siendo ML aún mejor? ¿Podemos sostener substancialmente alguna de estas preguntas o sólo son adecuadas para una discusión nocturna con un vaso de cerveza ? Mientras que ciertamente hay un elemento subjetivo en el diseño de un lenguaje de programación, también hay una rigurosa teoría científica de los lenguajes de programación que proporciona un marco para enunciar y algunas veces para responder tales preguntas. También hay buenas preguntas para las cuales la teoría actual no ofrece soluciones pero sorpresivamente muchos problemas son susceptibles para un análisis riguroso, que proporcione respuestas definidas para muchas preguntas.

La teoría de lenguajes de programación es fundamental para la implementación de los mismos, así como para su diseño. Mientras que los escritores de compiladores se basan fuertemente en la teoría de gramáticas para el análisis sintáctico y en la teoría de gráficas para el alojamiento de registros, los métodos usados para compilar lenguajes bien conocidos como C no se basan en resultados profundos de la teoría de lenguajes de programación. Para lenguajes relativamente sencillos, bastan métodos de compilación relativamente sencillos. Pero cuando los lenguajes se vuelven más sofisticados, también se complican los métodos empleados para compilarlos.

Por ejemplo, algunos programas pueden eficientarse substancialmente si la generación de código se pospone hasta que algunos datos de tiempo de ejecución estén disponibles. Un ajustado ciclo interno podría "desenrrollarse" mediante una secuencia lineal de instrucciones una vez que la cota de iteración se ha

*Agradecemos la colaboración de Susana Hahn Martin Lunas

[1] `http://www.cs.cmu.edu/~rwh/courses/ppl/phil.html`

determinado. Este es un ejemplo de evaluación parcial, una técnica de especialización de programas cuyos orígenes se encuentran en resultados de la teoría de lenguajes de programación. Para poner otro ejemplo, lenguajes modernos como ML y algunas extensiones propuestas de JAVA incluyen lo que se conocen como tipos parametrizados que soportan la reutilización flexible de código. Los tipos parametrizados complican de manera considerable a los compiladores porque deben tomar en cuenta situaciones donde el tipo de una variable o del argumento de una función se desconocen en tiempo de compilación. Los métodos más efectivos para manejar tipos parametrizados se basan en lenguajes tipados intermedios que poseen sistemas sofisticados de tipos. Nuevamente la teoría de lenguajes de programación proporciona un fundamento para construir tales compiladores.

La teoría de lenguajes de programación tiene muchas aplicaciones en la práctica. Por ejemplo, lenguajes pequeños surgen frecuentemente en sistemas de software, tales como lenguajes de comandos, de script, archivos de configuración o lenguajes de marcado. Los principios básicos de lenguajes de programación se desdeñan frecuentemente en esta clase de lenguajes con resultados por todos conocidos. Despues de todo, se argumenta comúnmente, son sólo lenguajes de script, así que para que tomarse la molestia. Una razón es que lo que inicia como un pequeño lenguaje ad-hoc con frecuencia crece tanto que se convierte en un lenguaje por derecho propio. La teoría de lenguajes de programación puede servir como una guía para el diseño e implementación tanto de estos lenguajes de propósito especial como de lenguajes de propósito general.

Otra aplicación de la teoría de los lenguajes de programación consiste en proporcionar un fundamento riguroso para la ingeniería de software. Los métodos formales para la ingeniería de software se fundamentan en la teoría de especificación y verificación. Una especificación es una fórmula lógica que describe el comportamiento intensional del programa. Hay diversas clases de especificaciones, desde simples condiciones de tipado hasta invariantes complejos que gobiernan las variables compartidas de un programa concurrente. La verificación es el proceso de checar si la implementación satisface la especificación. Existe actualmente mucho trabajo en el desarrollo de herramientas para especificar y verificar programas. La teoría de lenguajes de programación hace una conexión precisa entre el código y su especificación, y proporciona las bases para construir herramientas para analizar programas.

La teoría de lenguajes de programación proporciona una "verificación de la realidad" en la metodología de programación, esa parte de la ingeniería de software concerniente con la codificación de enfoques éxitosos al desarrollo de software. Por ejemplo, los méritos de la programación orientada a objetos para el desarrollo de software son bien conocidos y pregonados ampliamente. La metodología orientada a objetos depende fuertemente de las nociones de subtipado y herencia. En muchos reportes o recapitulaciones ambas nociones se confunden e inclusive se combinan en un solo concepto, aparentemente porque ambos se relacionan con la idea de una clase como mejora o enriquecimiento de otra. Pero un análisis cuidadoso revela que los dos conceptos son, y deben ser, distintos: confundirlos lleva a programas que violan las fronteras de la abstracción o que incluso incurren en fallas en tiempo de ejecución.

2. Objetivos

Reconocer los conceptos básicos sobre los cuales se fundamentan los lenguajes de programación, lo cual nos facilitará:

- Entender ideas y métodos de programación.
- Comparar lenguajes reconociendo sus diferencias y coincidencias.
- Seleccionar el paradigma mas adecuado para una tarea particular.
- Entender paradigmas y lenguajes en desarrollo.

3. Introducción

3.1. ¿Qué es un lenguaje de programación?

- Un lenguaje natural sirve para comunicar ideas entre personas.
- Análogamente un lenguaje de programación debe servir para comunicar ideas *algorítmicas* entre personas y computadoras.
- Un lenguaje de programación se usa para comunicar instrucciones a una computadora.
- Las instrucciones corresponden a cómputos que la máquina debe llevar a cabo (paradigma imperativo), o bien a especificaciones de dichos cómputos (paradigma declarativo)
- Sabemos que la noción de cómputo puede formalizarse, por ejemplo con máquinas de Turing, funciones recursivas o cálculo lambda.
- Un lenguaje es computacionalmente **completo** si puede expresar todas las funciones formalmente computables.
- Es deseable que los lenguajes de programación sean computacionalmente completos.
- Un lenguaje de programación es esencialmente un sistema notacional para representar cómputos en forma legible tanto para humanos como para computadoras.
- Legible para humanos significa que el lenguaje debe constar de abstracciones fáciles de entender.
- Legible para computadoras significa que debe existir un algoritmo eficiente de traducción del código fuente escrito por un humano al código ejecutable por una máquina.

3.2. Definición de un lenguaje de programación

Para definir completamente un lenguaje de programación se deben considerar tres aspectos básicos:

- *Sintaxis*: define la forma del lenguaje de programación, es decir la descripción del conjunto de cadenas de símbolos que serán considerados programas válidos.
  - Hay dos niveles, sintaxis concreta y sintaxis abstracta.
  - Se fundamenta en la teoría de lenguajes formales, en particular, gramáticas libres de contexto y notación BNF y EBNF, desarrollada por John Backus para Algol 58 y modificada por Peter Naur para Algol 60.
  - Existen diversas herramientas para generar analizadores léxicos (*lexers*) y sintácticos (*parsers*).
- *Semántica*: define el significado de instrucciones y expresiones del lenguaje.
  - Puede ser descrita de manera informal en manuales técnicos de un lenguaje o bien formal, basada en técnicas matemáticas en cuyo caso puede ser operacional, denotativa o axiomática.
- *Pragmática*: define la implementación del lenguaje basada en la metodología y estrategias de programación deseadas.

En nuestro curso nos ocuparemos mayoritariamente de los dos primeros aspectos. Una pregunta inmediata es ¿Por qué deben establecerse formalmente la semántica y la sintaxis de un lenguaje de programación? Posibles respuestas son:

- Para entender los programas.
- Para poder mostrar la correctud de un programa.
- Para poder mostrar la correctud de transformaciones entre programas.
- Para poder implementar y verificar compiladores o traductores.

Veamos un ejemplo muy sencillo de sintaxis y semántica de un lenguaje de programación para operaciones con conjuntos:

- Sintaxis:

$$\begin{aligned} < prog > &::= \mathsf{void} \mid (< prog > + < prog >) \mid (< prog > * < prog >) \\ &\quad \mid < elem > \mathsf{in} < prog > \\ < elem > &::= a \mid b \mid c \mid \ldots \mid z \end{aligned}$$

Por ejemplo los siguientes programas son válidos:

$$(a \ \mathsf{in} \ \mathsf{void}),\ (d \ \mathsf{in} \ (a \ \mathsf{in} \ \mathsf{void})),\ ((g \ \mathsf{in} \ \mathsf{void}) \ \mathsf{in} \ \mathsf{void})$$

$$\Big((h \ \mathsf{in} \ (g \ \mathsf{in} \ \mathsf{void})) \ + \ (k \ \mathsf{in} \ \mathsf{void})\Big) \ \big((f \ \mathsf{in} \ \mathsf{void}) \ * (t \ \mathsf{in} \ \mathsf{void})\big)$$

Y los siguientes son inválidos:

$$(a \ \mathsf{in} +),\ (a \ * \ d)$$

Verificar si un programa es válido o inválido es tarea de los analizadores léxico y sintáctico, cuyos fundamentos se encuentran en la teoría de lenguajes libres de contexto.

Veamos ahora la semántica del lenguaje, en este caso una semántica denotativa, del mismo estilo que la semántica de la lógica de primer orden. Tomamos una colección fija de conjuntos $M = \{A, B, C, \ldots, Z\}$ y definimos la función de significado, denotada $[\![\cdot]\!]$, para elementos y programas como sigue:

$$\begin{aligned} [\![a]\!] &= A \\ &\vdots \\ [\![z]\!] &= Z \\ [\![\mathsf{void}]\!] &= \varnothing \\ [\![(P \ + \ Q)]\!] &= [\![P]\!] \cup [\![Q]\!] \\ [\![(P \ * \ Q)]\!] &= [\![P]\!] \cap [\![Q]\!] \\ [\![(E \ \mathsf{in} \ P)]\!] &= \{[\![E]\!]\} \cup [\![P]\!] \end{aligned}$$

4. Clasificación de los lenguajes de programación

Podríamos pensar que la clasificación de los lenguajes de programación está determinada por su sintaxis o por las reglas de la semántica que lo definen. En general, no hay un esquema para su clasificación estricta pero pueden ser agrupados de distintas maneras:

- *Por la forma en la que se describen los cómputos*

  Se dividen en dos ramas: la de los lenguajes descritos imperativamente, a base de instrucciones

que cambian el estado del programa, por ejemplo: C, C++, Java, etc. Por otro lado están los declarativos, a base de condiciones que describen el problema cómo: Prolog, Haskell, Ml, Ocaml, etc.

- *Por su propósito*

  Se pueden dividir en los lenguajes de propósito general, los lenguajes de *script*, los de dominio específico y los concurrentes y distribuidos.

- *Por su ejecución*

  Se pueden dividir en dos: los que son compilados, cuando el programa será traducido a otro lenguaje (en general, lenguaje de máquina) por un compilador (Java, C, Haskell,etc) y por otro lado están los que son interpretados, donde se ejecuta cada instrucción de manera directa (JavaScript, Ruby, Python,etc).

- *Por su paradigma*

  Esta clasificación es la más común. Un paradigma (de programación) es un estilo fundamental de programación definido por la forma de dar soluciones a problemas. Un paradigma proporciona y determina la visión que el programador tiene acerca de la ejecución de un programa. En conclusión, los distintos paradigmas proporcionan diversas maneras de expresar el concepto de *computación*.

Veamos ahora con mas detalle algunos ejemplos y características de los principales paradigmas de programación.

4.1. Paradigma Procedimental

- Programa: serie de instrucciones (cálculos, entradas, salidas)
- La orientación es hacia los estados.
- Elementos de programación: abstracción procedimental, asignación, ciclos, condicionales.
- Lenguajes: Cobol, Fortran, Pascal, C, C++,etc.
- Evaluación:
  - Un cómputo se expresa mediante modificaciones implicitas a la memoria.
  - Las variables sirven como abstracción de celdas de memoria.
  - Los resultados intermedios se almacenan en memoria.
  - Las estructuras de datos son efímeras.
  - El control se basa en la iteración.
- Ejemplo:

```
% Calcular m^n

result := 1;
while n > 0 do
 result := result * m;
 n := n-1;
end while
```

4.2. Paradigma Orientado a Objetos

- Programa: colección de objetos que interactúan intercambiando mensajes que transforman estados.
- La orientación es, obviamente, hacia los objetos.
- Elementos de programación: modelado de objetos, clases, herencia, encapsulamiento.
- Lenguajes: SMALLTALK, C++, JAVA, C#, EIFFEL,etc.
- Evaluación:
  - Un cómputo se expresa mediante intercambio de mensajes entre objetos.
  - Los objetos se agrupan en clases, las cuales se agrupan en jerarquías.
  - Los resultados se pasan como parámetros a mensajes.
- Ejemplo:

```
public class FuncExp{

 public double expv1(int x, int y){
   double resp = 1;
   for(int i=0 ; i<y ; i++)
   resp *= x;
   return resp;
 }

 public double expv2(int x, int y){
  if(y>0)
   return x*(this.expv2(x,(y-1)));
  else
   return x;
 }
}

public static void main(String x[]){
 FuncExp fnexp = new FuncExp();
 System.out.println(fnexp.expv1(2,3));
}
```

4.3. Paradigma Lógico

- Programa: colección de declaraciones lógicas que especifican las características que debe tener la solución buscada.
- Un programa es declarativo: no importa el cómo sino el qué.
- La orientación es hacia las pruebas formales.
- Elementos de programación: cláusulas de Horn, unificación , retroceso.
- Lenguajes: PROLOG y sus derivados.
- Evaluación:
  - Un cómputo se expresa mediante la búsqueda de pruebas o por definición de predicados recursivos.
  - No hay memoria implícita.
  - Los resultados intermedios se pasan mediante unificación.
  - Control basado en recursión.
- Ejemplo:

```
% Calcular m^n
% resta/3, prod/3 predefinidos

pot(M,0,1).

pot(M,N,R) :- resta(N,1,N1),
              pot(M,N1,RT),
              prod(M,RT,R).
```

4.4. Paradigma Funcional

- Programa: colección de funciones que se combinan mediante composición de forma compleja para construir nuevas funciones.
- Un programa es declarativo: no importa el cómo sino el qué.
- La orientación es hacia la evaluación.
- Elementos de programación: composición, orden superior, curryficación, recursión.
- Lenguajes:
  - Impuros y ansiosos: LISP, SCHEME, ML
  - Puros y perezosos: MIRANDA[2], GOFER, HASKELL 98.
- Evaluación:
  - Un cómputo se expresa a través de la aplicación y composición de funciones.

[2] MIRANDA es marca registrada de Research Software Limited

- No hay memoria implícita.
- Los resultados intermedios se pasan directamente a otras funciones.
- Las estructuras de datos son persistentes (en lenguajes puros)
- Control basado en recursión.

- Ejemplo:

```
-- Calcular m^n

pot m 0 = 1

pot m (n+1) = m * (pot m n)
```

Como hemos notado, la teoría de lenguajes de programación es una rama de las ciencias de la computación que se dedica al diseño, implementación, análisis, caracterización y clasificación de los lenguajes de programación. Los campos que se relacionan con esta teoría son: semántica formal, teoría de tipos, análisis y transformación de programas, compiladores, estudio de los lenguajes de dominio específico, los sistemas en tiempo de ejecución, entre otros.
En nuestro curso nos dedicaremos exclusivamente a los paradigmas funcional, procedimental (imperativo) y orientado a objetos, dado que son la base de paradigmas más modernos y en desarrollo.