Instituto Tecnológico de Costa Rica

Compiladores e Intérpretes

Grupo 40

Profesor: Dr. Francisco Torres Rojas

Apuntes de clase: 17 de febrero de 2017

Apuntador: Víctor Andrés Chaves Díaz

Carné: 2015107095

2017

Tabla de contenidos

Recordatorios

1. Quiz, 22 de febrero.
2. Resumen del capítulo 3 de *The Language Instinct*, de Steven Pinker. 1^{ero} de marzo, en LaTeX y físico.
3. Proyecto #1, 1^{ero} de marzo.
    a. El compilador también se puede compilar a ARM con el fin de que corra en el simulador de Raspberry Pi.
4. Tarea grupal, ubicada en este link, para el 8 de marzo. Mismos grupos de proyecto, en LaTeX y físico.

Nota:

La ortografía es importante en los trabajos, en especial las tildes.

Sistemas Operativos

Definiciones

- Real: se ve y existe.
- Virtual: se ve, pero no existe. Es simplemente una ilusión del software.
- Transparente: no se ve, pero existe, simplemente no lo percibimos.
- Máquina: dispositivo que realiza una tarea computacional, puede ser real (el hardware de una computadora personal), virtual (sistema operativo) o transparente (servidores, sistemas empotrados).

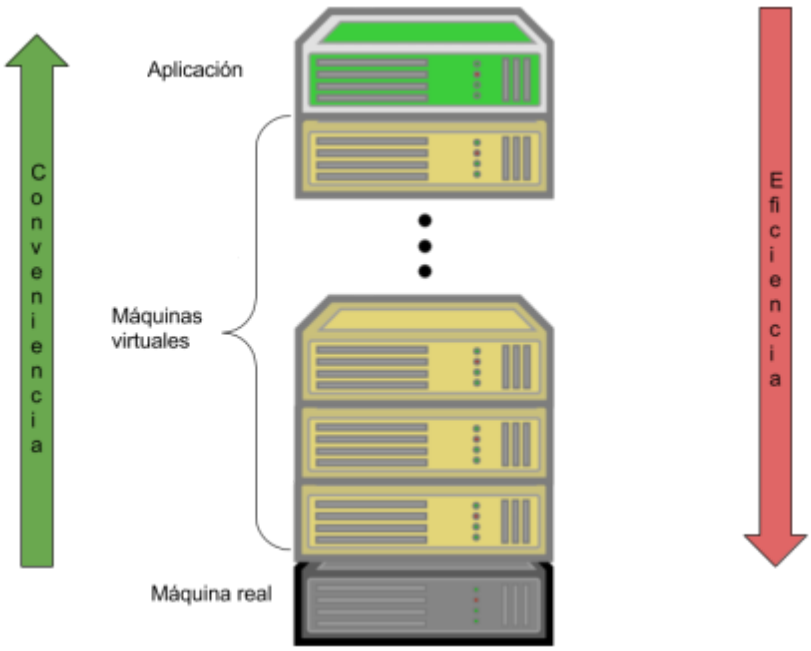
Sistema operativo como máquina virtual

Un sistema operativo consiste en una capa de software (una máquina virtual), que toma rehén al hardware, quitándole el control total al usuario, pero a cambio ofrece una serie de servicios y más facilidad al usuario de utilizar la computadora. Gracias a este intercambio, las capas sobre las que se trabajan cambian:

Generalización: máquina multinivel

¿Y si hacemos máquinas virtuales sobre máquinas virtuales?

Hacer esto nos permite abstraer más y más funciones, cambiando conveniencia por eficiencia, como lo muestra el siguiente diagrama.



Y la abstracción que hemos alcanzado no nos hace pensar en que las aplicaciones también son máquinas virtuales: una caja registradora moderna no existe realmente, es simplemente un software (la caja virtual) trabajando en conjunto con una pieza de hardware (la caja física), pero el cajero humano solo ve una caja registradora. Es por esto que nuestro trabajo será hacer máquinas virtuales (muy probablemente apps).

Máquinas y lenguajes

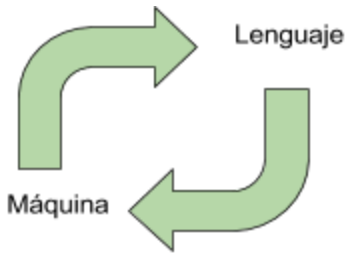
Definiciones

- Programa: es un texto formado por símbolos que describe y prescribe (lo define y limita) el comportamiento de una máquina. En esencia, le indica a la máquina lo que puede hacer, y la máquina entiende el texto directamente y lo ejecuta.

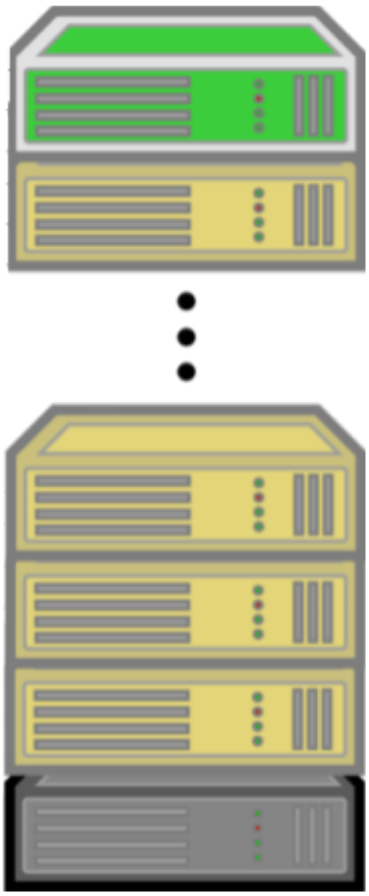
```c
/* This line basically imports the "stdio" header file, part of
 * the standard library. It provides input and output functionality
 * to the program.
 */
#include <stdio.h>

/*
 * Function (method) declaration. This outputs "Hello, world" to
 * standard output when invoked.
 */
void sayHello() {
    // printf() in C outputs the specified text (with optional
    // formatting options) when invoked.
    printf("Hello, world!");
}

/*
 * This is a "main function". The compiled program will run the code
 * defined here.
 */
void main() {
    // Invoke the sayHello() function.
    sayHello();
}
```

- Lenguaje de programación: lenguaje apropiado para escribir programas o cómputos, y ***todas*** las máquinas tienen un lenguaje que entiende directamente. Este lenguaje define las capacidades de la máquina.

Se puede entender que un lenguaje define a una máquina, y una máquina establece a un lenguaje.



Adicional a esto, como se puede tener *n* máquinas virtuales, cada una de estas tiene su propio lenguaje, por lo que podemos ver cada máquina de la siguiente manera:



M_{app} usa L_{app}: las aplicaciones también tienen su lenguaje propio

M_{app-1} usa L_{app-1}

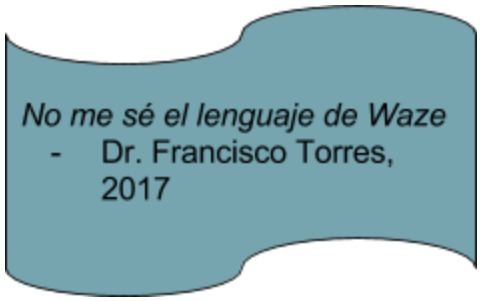
M_3 usa L_3

M_2 usa L_2

M_1 usa L_1

M_0 (hardware) usa L_0 (lenguaje real)

Es interesante notar que las aplicaciones *también* tienen un lenguaje, el cual no es el mismo en el que está programado, este lenguaje de la aplicación se refiere a un lenguaje que usa el usuario para comunicarse con la aplicación, ahora normalmente por medio de una interfaz gráfica, y para poder manejar una aplicación hay que conocer su lenguaje (a relevancia surge la cita en el margen derecho). En resumen, se podría decir que nosotros (los estudiantes de computación) creamos lenguajes.

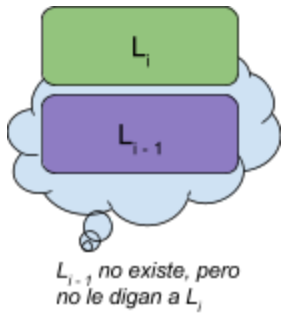


Lenguajes

- Existe una equivalencia entre máquina y lenguaje.
- Si queremos crear una máquina virtual, tenemos que crear un lenguaje, y ese lenguaje tiene que permitir poder comunicarme con el lenguaje directamente debajo.
- Todas las aplicaciones tienen su propio lenguaje, Netflix tiene uno, Waze tiene otro, y así con todos.
- El punto de una aplicación es que tanto la gente por encima mío (en el caso de una aplicación, el usuario) como los debajo de mi me entiendan, y para eso se usa un lenguaje de alto nivel.

Creación de máquinas virtuales

- M_i existe sobre M_{i-1}, y M_i supone que M_{i-1} es una máquina real, pero es meramente una ilusión.
- Con esto surge un problema: como se hace para que L_i se comunique con L_{i-1}? Para esto, se ocupa del mapeo de lenguajes.



L_{i-1} no existe, pero no le digan a L_i

Mapeo de lenguajes

Traducción (se usa un compilador)

- Convierte directamente L_i a L_{i-1}.
- Se debe esperar a que termine la traducción.
- Solo se hace una vez.
- Se vuelve muy rápido al ejecutar.

Interpretación (se usa un intérprete)

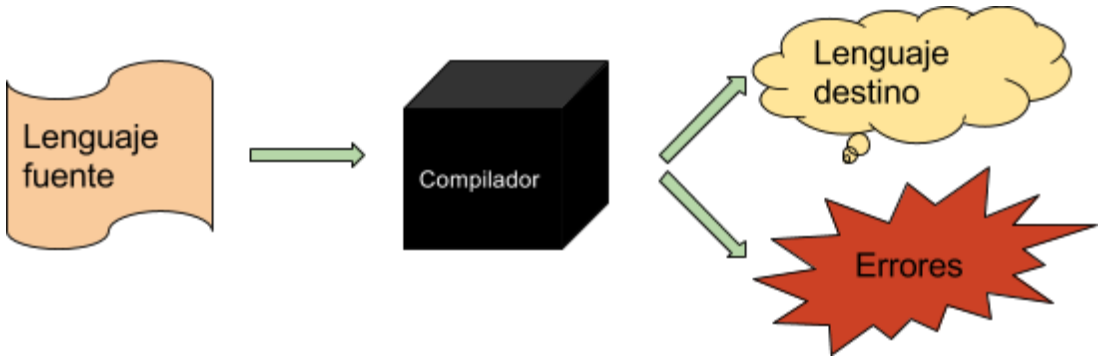
- Toma una instrucción de M_i y se *simula* su ejecución, con código en M_{i-1}.
- No se debe esperar la traducción.
- Lento en ejecutar, pues cada vez que se deba ejecutar se debe interpretar.

Con estos conceptos, podemos describir a las máquinas que realizan cada tarea.
Un compilador es un software que traduce un programa escrito en el lenguaje L_i de una máquina virtual a un programa equivalente en el lenguaje L_{i-1} de otra máquina; por otro lado, un intérprete es un software que ejecuta un programa en L_i para la máquina M_i y corre una simulación de dicha máquina en M_{i-1}, con código en L_{i-1}. Con estos conceptos, se

puede decir que una computadora es un intérprete, implementado en hardware por medio de circuitería.

Diagramas T

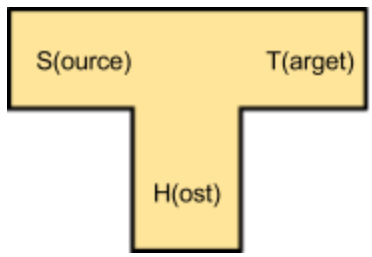
Para comenzar, se puede visualizar el proceso de compilación con el siguiente diagrama:



Y a cada compilador vienen asociados 3 lenguajes:

- Lenguaje fuente (Source): es la entrada del compilador, y es lo que se busca traducir.
- Lenguaje objeto/destino (Target): el lenguaje al que se busca traducir el lenguaje fuente.
- Lenguaje del compilador (Host): el lenguaje en el cual está escrito el compilador.

Estos tres lenguajes son ubicados en forma de T para formar lo que se conoce como *Diagrama T*, como se ejemplifica en la imágen a la derecha. Los diagramas T se leen: El compilador toma código escrito en *S* y lo traduce a *T*; pero propiamente el compilador se encuentra escrito en *H*.



GCC

- Originalmente llamado **GNU C Compiler**, ahora significa **GNU Compiler Collection**.
- Como su nombre lo indica, consiste en una serie de diferentes compiladores (C, C++, FORTRAN, ADA, etc.)
- Creado por Richard Matthew Stallman (o rms), programador estadounidense, activista del software libre, creador del proyecto GNU, GCC, Emacs y GPL, además de estar loco, en “una tarde lluviosa” en x86… o en otro lenguaje?
- Es un estándar para GNU/Linux.

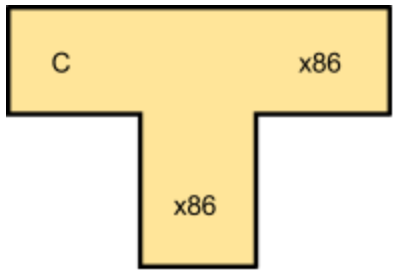


Diagrama T de GCC

Loco, pero no idiota

F2C

- FORTRAN 77 to C/C++.
- ¿Por qué? *Because I can*.
    - Una razón más explicativa es que resulta más sencillo hacer un compilador de FORTRAN a C, que uno de FORTRAN a x86.

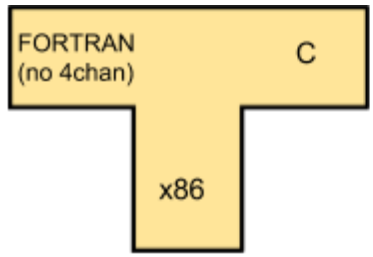
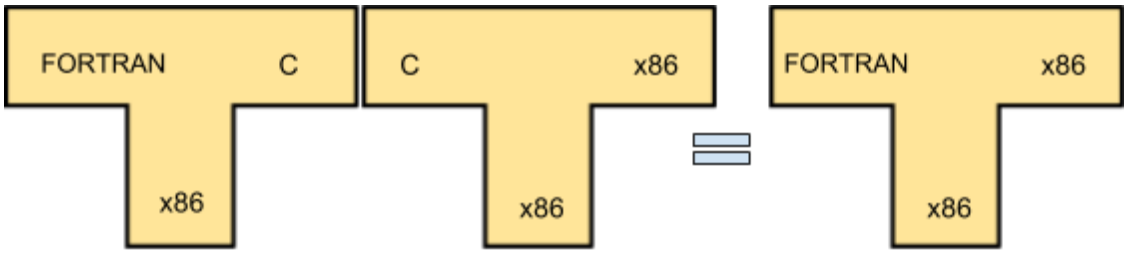


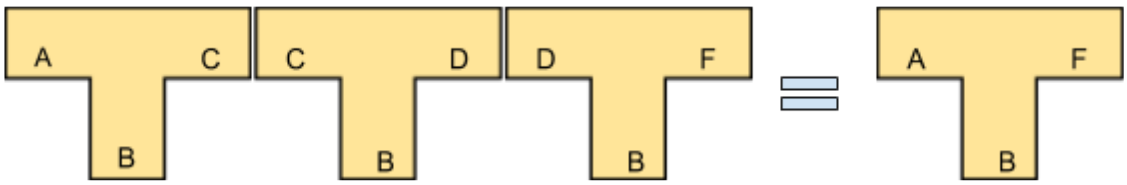
Diagrama de F2C

Interesantemente, existe un álgebra para diagramas T, que permiten describir procesos de compilación más complejos, por ejemplo:



F2C GCC F2C + GCC

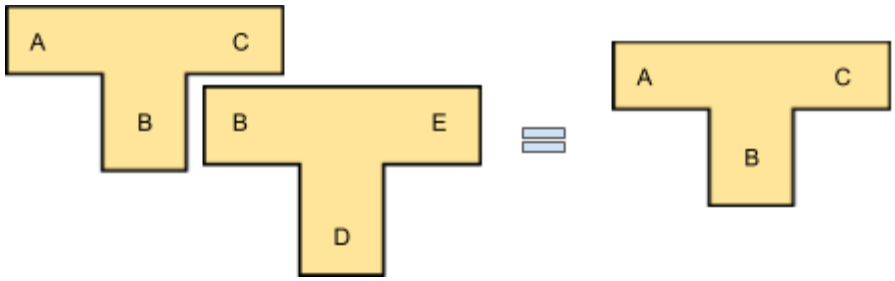
Este proceso de concatenación nos permite, al unir dos compiladores, tener un compilador que convierte FORTRAN a x86, pasando por C. Aunque el proceso sea más lento (pues hay que hacer dos compilaciones), el rendimiento del programa no se debería de ver severamente afectado si los dos compiladores están bien optimizados, pero la ventaja principal de esto es la conveniencia: si tengo que compilar algo en FORTRAN hacia x86 y no tengo a mano un compilador FORTRAN -> x86 o no existe, puedo hacer esta concatenación y compilo mi programa.
Este proceso no tiene un límite, de hecho, un diagrama como el siguiente es perfectamente válido:



E inclusive es posible que el lenguaje *Target* sea igual que el lenguaje *Host*, como se puede visualizar en el diagrama de F2C + GCC.

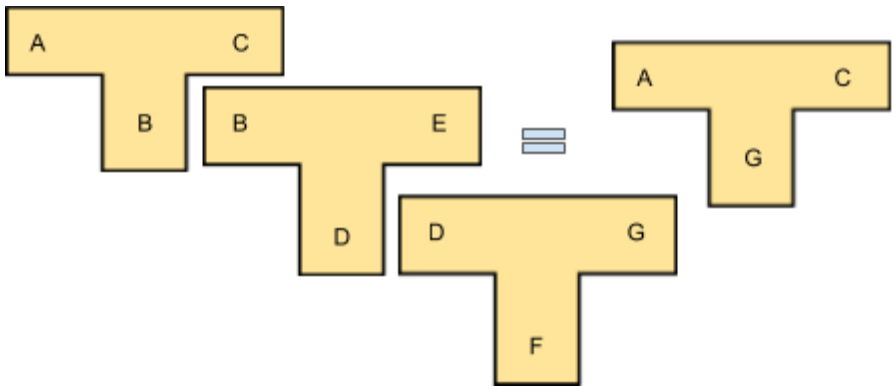
Lenguaje de implementación

Adicional a la concatenación, es posible describir el comportamiento cuando se cambia el lenguaje host de un compilador, de la siguiente manera:

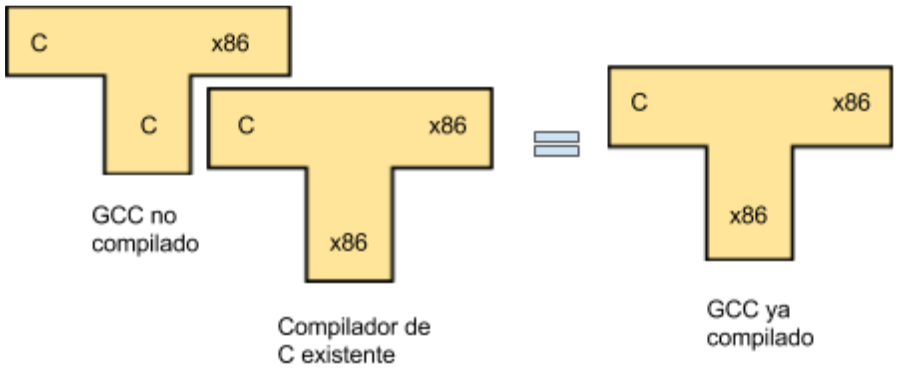


Se puede notar que el primer compilador está escrito en B, pero al compilarlo, pasa a estar escrito en E. Este proceso de conversión se puede extender a usar un compilador (A) para compilar el compilador (B) que compila nuestro compilador (C).

Este proceso se puede concatenar en varios niveles, por ejemplo:



Ahora, como se había mencionado previamente, Richard Stallman está loco, pero no es idiota: el GCC original fue escrito en C, no en x86, y el proceso que (muy probablemente) se siguió fue el siguiente:

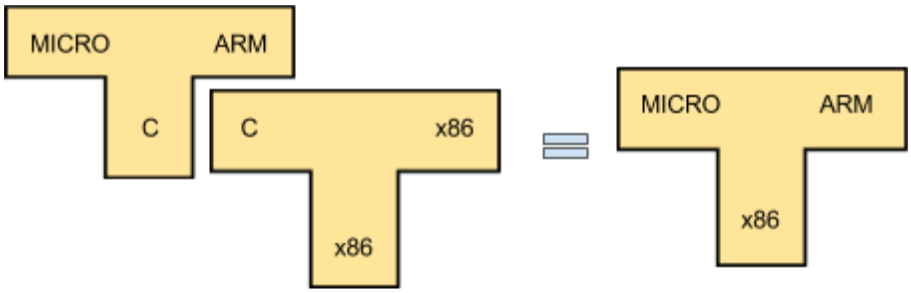


Y es muy probable que el compilador usado para compilar gcc no fuese escrito originalmente en x86, y también fue compilado por otro compilador, y este compilador también fue compilado por otro compilador, y así sucesivamente.

Lenguaje objeto

- Estos compiladores se utilizan cuando el lenguaje Target y el lenguaje Host son diferentes.
- Al traductor, sin embargo, le vale un pepino en que lenguaje está escrito, mientras funcione.
- Un ejemplo es la arquitectura ARM, que por naturaleza es de pocos recursos, por lo que compilar en un procesador ARM es desperdiciar recursos.
- Adicionalmente, permite simular arquitecturas que aún no existen, para cuando la tecnología lo permita (en unos 10-20 años) el software se pueda compilar (en 5 minutos ya tenemos un sistema operativo, los compiladores y otros programas misceláneos).

Este comportamiento lo podemos ver presente en el compilador del proyecto pendiente a la fecha (compilador de MICRO), ejemplificado en este diagrama:



A este compilador generado se le conoce como *cross compiler*.

Cross compiler

- Permite convertir de una arquitectura X a una arquitectura Y.
- El código objeto que genera el compilador no es el mismo que utiliza la máquina sobre la que corre el compilador.
- Máquina 1 ≠ Máquina 2.
- Se puede ocupar en ciertos casos:
    - La máquina 1 no existe físicamente, pero el código se puede probar en simuladores.
    - La máquina 1 no tiene suficientes recursos para compilar el programa (como un sistema empotrado).

En conclusión, nuestro proyecto será un *cross compiler* de MICRO a ARM, escrito en C y corriendo sobre x86.

ARM

- Consiste en una familia de arquitecturas.
- Introducida en 1985, y diseñada por ARM Holdings.
- Es una arquitectura de *reduced instruction set computing* (RISC). Por naturaleza, no es microprogramada, es alambrada.
- Eficiente en uso de energía y espacio, perfecto para dispositivos pequeños como celulares, tabletas y sistemas empotrados.

- ARM Holdings solamente diseña el procesador, la construcción la hacen compañías que licencian el procesador.
- Existen procesadores de tanto 32 bits como de 64 bits.
- Es la arquitectura más usada del mundo:
    - Según reportado en su página principal, los socios de ARM Holdings han producido más de 90 mil millones de procesadores ARM.
    - Según su página de mercados, el 95% de todos los dispositivos inteligentes usan un ARM.

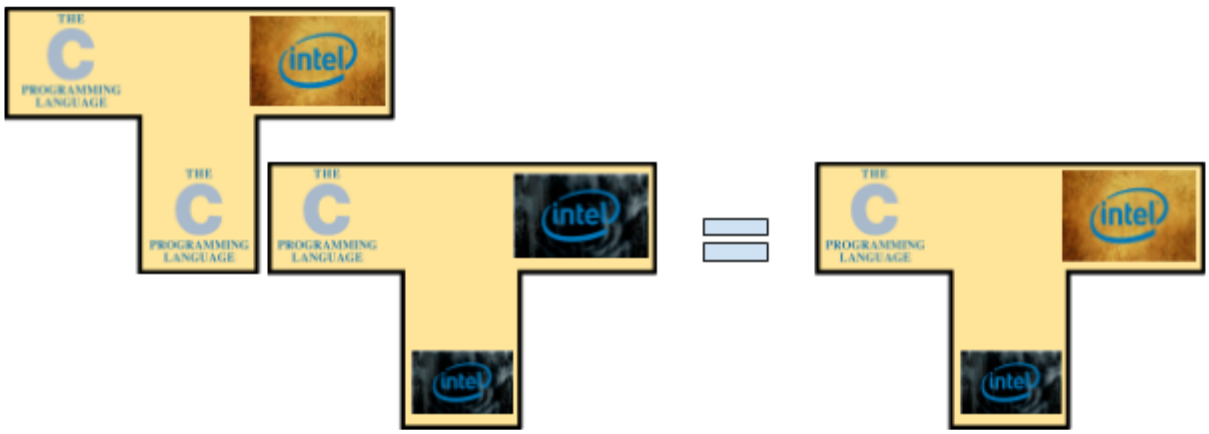
Para Linux en arquitecturas x86, se puede usar `gcc-arm-linux-gnueabihf` para compilar C a ARM.

Lenguajes de implementación

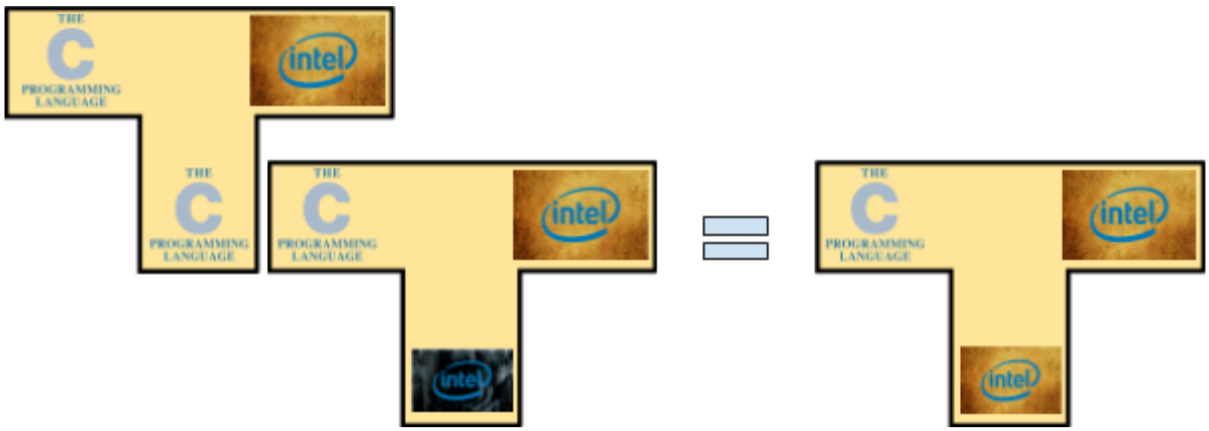
No es de extrañarse que algunos compiladores originalmente se escriban en el lenguaje fuente que va a traducir, como fue en el caso de gcc, que originalmente se tuvo que compilar usando un compilador ya existente, o como puede ser el caso cuando se compila una nueva versión de gcc (por razones de producir código más eficiente), siguiendo los siguientes pasos:

Cómo compilar GCC usando GCC

1) Se tiene a mano GCC 4.5.3, originalmente lanzado el 28 de abril del 2011.
2) Se quiere cambiar a una versión más nueva (el ejemplo dado en clase menciona 5.2, lanzado el 16 de julio de 2015, la versión más nueva a la fecha, 6.3, fue lanzada el 21 de diciembre de 2016).
3) Porque somos todos profesionales, buscamos y descargamos el código fuente de la versión que queremos instalar (para ejemplo, supongamos descargar de este sitio web, pero una plétora de diferentes sitios se detalla en la página de mirrors de gcc).
4) Con el código fuente, debemos compilarlo, pero, ¿como hacemos? Sencillo: utilizamos la versión vieja de GCC para compilar.



5) Ahora tenemos un compilador de C que produce x86 muy eficiente, pero está escrito en x86 de baja calidad y no tan optimizado, entonces… volvemos a compilar el código fuente, usando el compilador que acabamos de crear.



6) Con esto, ya tenemos un compilador que produce código de buena calidad y además es muy eficiente.