# Equivalencia cruzada

del Valle Vera Nancy Elena, Martínez Hidalgo Paola Mildred, Sánchez Victoria Leslie Paola
*Facultad de Ciencias, Universidad Nacional Autónoma de México*

***Resumen*—La compilación cruzada permite generar código objeto para una arquitectura o plataforma diferente de aquella en la que se realiza el desarrollo.**

**Los compiladores necesitan conocer en detalle la arquitectura y la representación de memoria de la máquina destino para generar un código correcto, es decir, código que conserve el significado semántico del código fuente.**

**Es entonces que surge la siguiente interrogante: ¿cómo garantizan los compiladores cruzados la consistencia entre arquitecturas con diferentes conjuntos de instrucciones y representaciones de memoria?**

***Palabras clave*—compilador cruzado, diagrama T, arquitectura destino, representación intermedia, generación de código**

## I. Introducción

En 1936, Alan Turing introdujo el concepto de máquina automática, respaldado por una rigurosa demostración matemática, esta noción anticipó la capacidad de los sistemas computacionales modernos para almacenar, comunicar y procesar información [1].

En sus primeras etapas, las computadoras únicamente ejecutaban algoritmos fijos destinados a cálculos específicos y carecían de la adaptabilidad necesaria para resolver tareas diversas. Ante el incremento de la demanda de cómputo y la necesidad de mecanismos de reprogramación más flexibles, John von Neumann formuló en 1945 el concepto de programa almacenado, estableciendo así los fundamentos de la arquitectura que lleva su nombre.

La arquitectura de von Neumann se caracteriza por el hecho de que el programa ejecutado por el sistema informático se encuentra almacenado en la memoria principal del propio sistema [2]. En sus inicios, esta arquitectura operaba mediante lenguaje máquina, el cual consistía en asignar valores numéricos a operaciones específicas, siendo este un proceso tedioso y lento. Posteriormente, su funcionamiento fue reemplazado por el lenguaje ensamblador, que introdujo una representación simbólica de las operaciones. Aunque esto mejoró la precisión y la velocidad de desarrollo, el ensamblador también presentaba limitaciones, pues resultaba difícil de escribir, leer y comprender [3].

Ante estas dificultades, surgió la necesidad de formular instrucciones en una notación más cercana a la matemática o al lenguaje natural.

Entre 1954 y 1957 se desarrolló FORTRAN, considerado el primer lenguaje de programación de alto nivel de uso extendido, acompañado de su compilador, dirigido por John Backus. De manera paralela, Noam Chomsky inició el estudio formal de la estructura del lenguaje natural, sus contribuciones facilitaron de manera significativa la construcción de compiladores [3].

Eventualmente surgió la necesidad de generar software capaz de ejecutarse en arquitecturas distintas a aquella utilizada durante su desarrollo, lo que condujo al surgimiento de los compiladores cruzados.

En la actualidad, los compiladores cruzados constituyen una herramienta esencial para la creación de software dirigido a plataformas en las que la compilación directa resulta complicada, lenta o incluso inviable, como ocurre en sistemas embebidos, arquitecturas móviles y dispositivos con recursos computacionales limitados [4].

## II. Más compiladores

Un *compilador* es un programa que traduce el código fuente escrito en un lenguaje de programación a un programa equivalente escrito en código máquina, código intermedio u otro lenguaje de programación [5].

Entre los tipos de compiladores que serán de interés para el desarrollo del presente trabajo se identifican dos: el compilador nativo y el compilador cruzado.

Un ***compilador nativo*** traduce el código fuente al código máquina donde reside el compilador. Este comportamiento garantiza que los programas creados logren una ejecución de alto rendimiento.

Sin embargo, el código generado suele ser dependiente de cada lenguaje y entorno, por lo que, en muchas ocasiones, se requiere de distintos compiladores para adaptarse a cada uno de los posibles casos [6].

Un ejemplo de compilador nativo es uno que se ejecuta en un sistema operativo Windows y genera código ejecutable para el mismo sistema.

Un ***compilador cruzado*** permite compilar código fuente en una plataforma para que pueda ejecutarse en otra plataforma diferente. La principal ventaja de los compiladores cruzados es que permiten compilar para múltiples plataformas desde un solo entorno de desarrollo [7].

Un ejemplo de compilador cruzado es uno que se ejecuta en un sistema operativo Windows pero es capaz de emular y generar código ejecutable para un teléfono Android.

### *II-A. Diagramas T*

Los diagramas T, también llamados diagramas de lápida debido a su forma, constituyen una representación gráfica utilizada para describir la estructura y el proceso de construcción de un compilador.

Este tipo de diagramas permite visualizar de manera clara la relación entre el lenguaje fuente, el lenguaje objetivo y la máquina anfitriona en la que se ejecuta el compilador [3].

Para representar mediante un diagrama T a un compilador nativo, se considera que dicho compilador está escrito en un

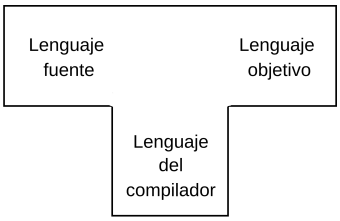


Figura 1. Estructura conceptual de un diagrama T.

lenguaje fuente *S*, se ejecuta en una máquina *M* y genera código objeto destinado a esa misma máquina.

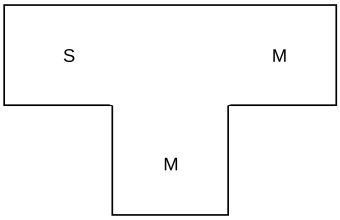


Figura 2. Diagrama T de un compilador nativo.

En contraste, un compilador cruzado realiza la traducción desde el lenguaje fuente *S* hacia un lenguaje destino *T*, mientras que su implementación y ejecución tienen lugar en una máquina *M*.

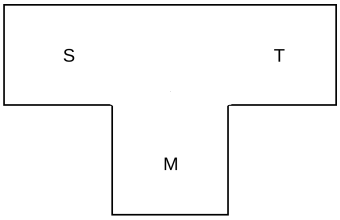


Figura 3. Diagrama T de un compilador cruzado.

## III. Arquitectura de computadoras

Un compilador debe enfocar su traducción a las capacidades de una arquitectura de máquina particular. Cada código máquina corresponde a una arquitectura propia implementada fisicamente en un microprocesador, a excepción de las máquinas virtuales [8].

La arquitectura de computadoras establece la forma en la que el hardware y software se comunican entre sí. Es un conjunto de reglas, principios y estándares que gestionan los recursos, la ejecución de intrucciones, el almacenamiento y el acceso a los datos [9].

Sin importar si la arquitectura es virtual o real, entender los principios que rigen el funcionamiento del procesador es necesario para la generación de código, ya que en este punto, se determina cómo el programa y sus datos se representan dentro de la memoria del procesador.

Conocer las capacidades de cada procesador permite ejecutar programas de una manera rápida y eficaz [9].

### *III-A. Arquitectura destino*

La arquitectura destino es aquella en la que se ejecutará el código objeto generado por el compilador.

La diferencia fundamental entre un compilador cruzado y un compilador nativo radica en la relación entre la arquitectura de la máquina donde se realiza la compilación y la arquitectura para la que se genera el código ejecutable.

- **Compilador nativo**: traduce el código fuente para la misma arquitectura en la que se ejecuta el compilador.
- **Compilador cruzado**: traduce el código fuente para una arquitectura distinta de aquella en la que se ejecuta el compilador.

## IV. Consistencia semántica

El significado semántico del código fuente debe preservarse tras el proceso de traducción al lenguaje destino, es decir, el compilador debe ser correcto [10].

Mientras que los detalles de traducción dependen de las características específicas de la representación intermedia, el lenguaje destino y el entorno de ejecución; tareas como la selección de instrucciones, asignación de registros y el ordenamiento de instrucciones se encuentran en el diseño de mayoría de los generadores de código [11]. Esto se debe a que el criterio más importante para un generador de código es que produzca código correcto.

Dado que el código generado por un compilador cruzado está destinado a una arquitectura distinta de aquella en la que se desarrolla el código fuente, resulta crucial garantizar la existencia de una equivalencia semántica, pues el usuario programador solo tiene certeza sobre el comportamiento del código escrito en el lenguaje fuente.

Entender cómo se garantiza la correctez en un compilador cruzado sólo involucra el entendimiento de la fase de síntesis, ya que la fase de análisis es independiente de la arquitectura destino.

## V. Representación intermedia

Durante la compilación, la fase de análisis se encarga de gestionar los detalles específicos del lenguaje fuente, mientras que la fase de síntesis administra aquellos correspondientes a la máquina destino [11].

Entre ambas fases se encuentra una etapa intermedia generada por el análisis, conocida como representación intermedia o simplemente RI, que constituye la base sobre la cual opera la síntesis.

La representación intermedia es una estructura de datos que describe el programa fuente durante la traducción [3].

El diseño o la elección de la RI varía entre compiladores; puede ser un lenguaje o consistir en estructuras de datos internas [11]. Si la representación intermedia se asemeja al código objeto se denomina código intermedio.

Utilizar representaciones intermedias facilita la construcción del código objeto al descomponer la traducción en dos partes más simples y manejables, permite realizar optimizaciones independientes de la máquina y habilitar la incorporación de nuevos generadores de código reutilizando una fase de análisis

ya existente; dando lugar así a la creación de compiladores reorientables [12].

Estas ventajas son las que permiten el desarrollo de compiladores cruzados.

## VI. Generación de código

La fase de generación de código toma como entrada la representacion intermedia producida a partir de la fase de análisis, junto con la información de la tabla de símbolos, la cual se utiliza para determinar las direcciones de los objetos de datos durante la ejecución, denotados por los nombres de la representación intermedia. Es así como esta fase produce el código dependiente de la arquitectura destino [11].

El generador de código tiene tres principales tareas: selección de intrucciones, repartición y asignación de registros, y ordenamiento de intrucciones.

### *VI-A. Seleccion de intrucciones*

La selección de instrucciones es una etapa altamente dependiente de la arquitectura destino. Su propósito es elegir la secuencia de instrucciones más adecuada para implementar las sentencias de la representación intermedia. [8].

La complejidad de traducir la representacion intermedia a un código ejecutable por la máquina destino depende del nivel de abstracción de dicha representacion, la arquitectura del conjunto de intrucciones y la calidad deseada del código generado [11].

### *VI-B. Repartición y asignación de registros*

La repartición y asignación de registros consiste en decidir qué valores mantener en qué registros.

La asignación de registros determina la correspondencia entre los valores temporales generados durante representaciones intermedias y los registros físicos disponibles en la arquitectura destino. Su objetivo es utilizar los registros de manera eficiente, reduciendo al mínimo la necesidad de desbordamientos [8].

Un desbordamiento ocurre cuando los registros disponibles para almacenar variables temporales se han agotado y el compilador se ve obligado mover el exceso de datos de los registros a residir en memoria.

### *VI-C. Ordenamiento de intrucciones*

El ordenamiento de instrucciones determina el orden de ejecución de las instrucciones generadas con el fin de reducir los bloqueos del pipeline en el procesador y sus latencias asociadas. Aunque existan múltiples ordenamientos válidos, no todos son igualmente eficientes [8].

Algunos ordenamientos producen retrasos que afectan el rendimiento del código generado. Estos retrasos suelen depender de las características particulares de la arquitectura y su implementación.

### *VI-D. Representación de memoria*

El compilador traduce el código fuente a instrucciones de máquina, asignando direcciones y tamaños según la representación de memoria. Esta representación define la forma en que los datos e instrucciones están almacenados y organizados. De esta forma, el código objeto puede contener referencias a los datos almacenados en los distintos bloques de memoria del programa [11].

La memoria consiste en un conjunto de registros numerados, cada uno puede almacenar un byte de información y debe ocupar una ubicación específica dentro de la memoria física de la máquina. El menor objeto al que puede hacerse referencia en memoria es el byte, no obstante, es habitual tener instrucciones que pueden leer y escribir palabras de más de un byte [13].

Una palabra es una unidad de datos de longitud fija en bits que se puede direccionar y mover entre el almacenamiento y el procesador [14]. Su tamaño depende de la arquitectura particular de cada procesador.

Al utilizar palabras de más de un byte, su representación en memoria puede seguir alguna de dos políticas de ordenamiento. El formato *big-endian* asigna la dirección de memoria más baja al byte más significativo de la palabra, mientras que el formato *little-endian* asigna la dirección al byte menos significativo [13]. Independientemente del orden de bytes adoptado, la posición interna de los bits dentro de cada byte permanece sin cambios. Es decir, para una misma palabra, si se compara el byte más significativo en una representación big-endian con el correspondiente en little-endian, el orden de sus bits será idéntico [15].

La elección del ordenamiento de bytes en memoria puede afectar la interpretación de los datos en arquitecturas heterogéneas. Por esta razón, los compiladores deben conocer el esquema de almacenamiento que empleará el código objeto con el fin de no alterar el significado semántico del código fuente.

## VII. Simulación

Hasta el momento, se ha llevado a cabo un estudio sobre el funcionamineto de la fase de síntesis en los compiladores, poniendo especial atención en momentos clave que pueden beneficiar o afectar en el desarrollo de compiladores cruzados.

Por ello, es posible abstraer los elementos fundamentales que permiten mantener la equivalencia semántica entre compiladores cruzados: la representación intermedia y la arquitectura destino.

Al separar la fase de análisis de la fase de síntesis mediante una representación intermedia, los compiladores cruzados pueden utilizar esta RI como punto de partida para generar código específico de la máquina destino, siempre que se conozcan las caracteríticas de la arquitectura destino.

Con el fin de ilustrar esta idea, se ha desarrollado la simulación de un mini-compilador cruzado que está escrito en Haskell. Este compilador permite generar dos códigos intermedios distintos para dos arquitecturas diferentes, simulando lo que, en un compilador cruzado real, correspondería a la salida de código objeto para cada arquitectura.

### VII-A. *Gramática*

Con el objetivo de generar una simulación ilustrativa de la producción de código dependiente de la arquitectura, se empleará la siguiente gramática, la cual permite trabajar con asignaciones, operaciones aritméticas y números enteros.

En un compilador cruzado real, la gramática utilizada debería coincidir con la del lenguaje fuente con el que opera el compilador.

$$S ::= \texttt{id} := E$$

$$E ::= E + T \mid E - T \mid T$$

$$T ::= T * F \mid F$$

$$F ::= (E) \mid -F \mid \texttt{num} \mid \texttt{id}$$

Figura 4. Gramática del lenguaje fuente para la simulación.

Se entiende por `id` el conjunto de símbolos terminales que permite la construcción de identificadores mediante letras del alfabeto, con una longitud mínima de un carácter. Esta definición es válida tanto para letras mayúsculas como minúsculas.

Por otra parte, `num` representa el conjunto de números naturales y la producción *F ::= -F* permite la construcción de números negativos, introduciendo así un caso de operación unaria dentro de la gramática. Esta inclusión se realiza de manera intencionada para garantizar la presencia de, al menos, un ejemplo de operación unaria en la representación intermedia.

### VII-B. *Fase de analisis*

El objetivo principal de esta simulación es ilustrar cómo, a partir de una representación intermedia, es posible generar distintos códigos dependientes para cada arquitectura destino. Por esta razón, no se profundiza en la fase de análisis del mini-compilador a nivel implementación.

Para facilitar el desarrollo de esta etapa, se decidió emplear bibliotecas proporcionadas por Haskell. En particular, para el análisis léxico se utiliza la biblioteca `Alex`, mientras que para el análisis sintáctico se recurre a la biblioteca `Happy`.

El análisis semántico no se implementará en este contexto; sin embargo, se asumirá que el árbol de sintaxis generado por el análisis sintáctico ha sido sometido a dicho análisis y, en consecuencia, se encuentra libre de errores semánticos.

Por conveniencia, también se asumirá que la salida del análisis semántico consiste en otro árbol de sintaxis que brinde mayor contexto al compialdor.

Cabe señalar que, en un compilador cruzado real, no es posible omitir ninguna de las fases de análisis, ya que ello podría comprometer la equivalencia semántica y derivar en la generación de código objeto de menor calidad.

### VII-C. *Fase de síntesis*

Una vez completada la fase de análisis, se procederá a transformar el árbol de sintaxis producido por el análisis semántico mediante una representación intermedia independiente.

Para este propósito, se adoptará como RI el código de tres direcciones. Esta elección se justifica en que dicho código constituye una representación lineal del árbol sintáctico [11], además, aparece como máximo un operador en el lado derecho de cada instrucción, lo que se adecua directamente al alcance de la gramática previamente definida.

El código de tres direcciones se basa en dos conceptos principales: direcciones e instrucciones.

Una dirección puede corresponder a una variable, una constante o un valor temporal generado por el compilador. Su representación mediante un tipo de dato en Haskell es la siguiente.

Código 1. Definición de las direcciones

```haskell
data Direccion
    = Var String
    | Cons Int
    | Temporal Int
```

Entre las formas más habituales de instrucciones de tres direcciones, se prestará especial atención a las siguientes:

- Instrucciones de asignación: $a = b$ op $c$
- Asignaciones unarias: $a = op\ b$
- Instrucciones de copia: $a = b$

En estas construcciones, *op* denota una operación aritmética, mientras que *a*, *b* y *c* representan direcciones válidas dentro del código intermedio.

Su tipo de dato correspondiente en Haskell es el siguiente.

Código 2. Definición de las instrucciones

```haskell
data InsTresDir
  = InsCopiado Direccion Direccion
  | InsUnaria Direccion Char Direccion
  | InsBinaria Direccion Char Direccion
      Direccion
```

Establecido lo anteiror, se tiene una función que toma un árbol de sintaxis y lo transforma recursivamente en alguna de las instrucciones de la RI independiente, simulando el primer paso de la generación de código.

Código 3. Firma de la función que convierte un árbol de sintaxis en código de tres direcciones equivalente.

```haskell
representacionI :: AS -> [InsTresDir]
```

A partir del conjunto de instrucciones definido en el código de tres direcciones, es posible generar los correspondientes códigos dependientes de cada arquitectura.

Nuestro mini-compilador será capaz de generar código para dos arquitecturas destino: la arquitectura x86 de 32 bits y la arquitectura ARM de 64 bits.

Para los fines de esta simulación, la generación se limitará a producir código intermedio específico de cada arquitectura. No obstante, es importante señalar que, en un compilador cruzado real, esta salida correspondería al código objeto final.

### VII-D. *Primera arquitectura destino*

Para la arquitectura x86 de 32 bits se empleará el lenguaje ensamblador AT&T de 32 bits, una sintaxis utilizada en sistemas tipo UNIX y adoptada por microprocesadores x86 que operan con este tamaño de palabra [16].

La generación del código simulado requiere la definición en Haskell de los siguientes tipos de datos.

Código 4. Tipos de datos necesarios para definir la sintaxis para AT&T de 32 bits

```
data Oper = Registro String
          | Vars String
          | Const Int

data Instx32 = Move Oper Oper
             | Add Oper Oper
             | Sub Oper Oper
             | Mult Oper Oper
```

Asimismo, se puede emplear un diagrama T para ilustrar el objetivo de implementación de esta arquitectura.

El primer caso del mini-compilador cruzado consiste en la traducción de un programa fuente $S$, definido por la gramática previamente presentada, cuya implementación en Haskell genera como salida código en lenguaje ensamblador AT&T de 32 bits.

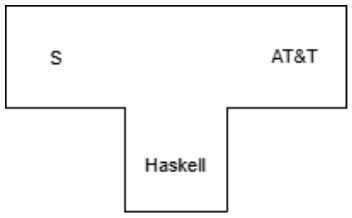


Figura 5. Diagrama T para la primer arquitectura destino que puede soportar nuestro mini-compilador cruzado.

La función que toma la RI independiente y la transforma en una versión dependiente es la siguiente.

Código 5. Firma de la función que convierte un conjunto de código de tres direcciones en código para una arquitectura x86 de 32 bits

```
codigoObjeto32 :: [InsTresDir] -> [Instx32]
```

### *VII-E. Segunda arquitectura destino*

Para la segunda arquitectura se utilizará el lenguaje ensamblador ARM de 64 bits, comúnmente empleado en computadoras de escritorio, portátiles y dispositivos móviles [17].

De manera análoga, para la arquitectura ARM64 se definen los siguientes tipos de datos.

Código 6. Tipos de datos necesarios para definir la sintaxis de ARM64

```
data Oper = Registro String
          | Vars String
          | Const Int

data InstARM64 = Move Oper Oper
               | Add Oper Oper Oper
               | Sub Oper Oper Oper
               | Mult Oper Oper Oper
               | Neg Oper Oper
               | LoadVar Oper String
               | StoreVar String Oper
```

El segundo caso del mini-compilador cruzado consiste en la traducción de un programa fuente $S$, definido por la gramática presentada en la Figura 4, cuya ejecución en Haskell produce como salida código en lenguaje ensamblador ARM64.

La función que toma la RI independiente y la transforma en una versión dependiente se presenta en el Código 7.

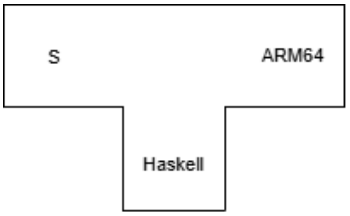


Figura 6. Diagrama T para la segunda arquitectura destino que puede soportar nuestro mini-compilador cruzado.

Código 7. Firma de la función que convierte un conjunto de código de tres direcciones en código dependiente de ARM64

```
codigoObjeto64 :: [InsTresDir] -> [InstARM64]
```

Finalmente, para emplear el compilador cruzado será necesario especificar la arquitectura destino a la cual se desea compilar un determinado código fuente.

Este comportamiento se formaliza mediante la siguiente función:

Código 8. Firma de la función que permite utilizar el compilador cruzado.

```
compilador :: Arquitectura
           -> String
           -> Either [Instx32] [InstARM64]
```

El uso de la mónada *Either* en Haskell resulta particularmente adecuado, pues permite representar explícitamente que un mismo valor puede ser uno de dos tipos distintos. En este caso, un mismo código fuente corresponde a alguna de dos posibles formas de código dependiente, cada una asociada a la arquitectura destino seleccionada.

### *VII-F. Ejemplo*

Considérese la siguiente instrucción, derivable de la gramática definida en la Figura 4:

Código 9. Expresión a compilar.

```
abc := 2 + (5 * (5 + 56) - 7)
```

Esta instrucción será tratada como el código fuente, a modo de simulación de un programa escrito en algún lenguaje de programación.

La cadena de entrada debe someterse a los procesos de análisis léxico, sintáctico y semántico; en nuestra simulación se aplican los dos primeros y se asume implícitamente el tercero.

A partir del árbol de sintaxis generado, la representación intermedia producida por la función del Código 3 muestra el siguiente resultado:

Código 10. Código de tres direcciones para la entrada abc := 2 + (5 * (5 + 56) - 7)

```
t0 = 5 + 56
t1 = 5 * t0
t2 = t1 - 7
t3 = 2 + t2
abc = t3
```

El código ensamblador generado para la arquitectura AT&T de 32 bits, producido por la función del Código 5, se presenta a continuación:

Código 11. Código intermedio en lenguaje ensamblador AT&T para la entrada abc := 2 + (5 * (5 + 56) - 7)

```
movl $5  %eax
addl $56 %eax
movl $5  %ebx
imull %eax %ebx
movl %ebx %ecx
subl $7  %ecx
movl $2  %edx
addl %ecx %edx
movl %edx %eex
movl %eex abc
```

Por su parte, el código ensamblador generado para la arquitectura ARM64, mediante la función del Código 7, es el siguiente:

Código 12. Código intermedio en lenguaje ensamblador ARM64 para la entrada abc := 2 + (5 * (5 + 56) - 7)

```
mov x0, #5
mov x1, #56
add x0, x0, x1
mov x2, #5
mul x1, x2, x0
mov x5, #7
sub x2, x1, x5
mov x6, #2
add x3, x6, x2
str x3, =abc
```

Si bien no se profundiza en la sintaxis particular de cada código dependiente, es posible apreciar similitudes claras entre estos y la representación intermedia independiente.

## VIII. Conclusión

Los compiladores cruzados garantizan la consistencia entre arquitecturas con diferentes conjuntos de instrucciones y representaciones de memoria al separar las fases de análisis y síntesis utilizando representaciones intermedias.

Esto permite generar distintos códigos objetos que corresponden a distintas arquitecturas a partir de un solo programa fuente.

Un compilador cruzado real requierirá información adicional más allá de la simple indicación de la arquitectura destino, incluyendo parámetros relativos a la máquina destino, convenciones de llamada, características del ensamblador y otros aspectos propios del entorno de ejecución.

El ejemplo permitió observar, de manera general, cómo una misma sentencia del código fuente puede producir distintas salidas según la arquitectura destino sin alterar la equivalencia semántica entre etapas.

Si bien el mini-compilador cruzado desarrollado cumple con su objetivo de mostrar, de manera general, el funcionamiento de los compiladores cruzados y cómo pueden aprovecharse las fases de compilación para su construcción, aún presenta limitaciones, entre ellas una gramática restringida y la ausencia de procesos de optimización del código. Como trabajo a futuro, podría ampliarse para soportar una gramática más completa, incorporar nuevas funcionalidades y aplicar optimizaciones al código generado, con el fin de acercarlo más a un compilador de uso real.

## Referencias


[1] G. S. Syed, M. Le Gallo, and A. Sebastian, "Non von neumann computing concepts," in *Phase Change Materials-Based Photonic Computing*, ser. Materials Today, H. Bhaskaran and W. H. P. Pernice, Eds. Elsevier, 2024, pp. 11–35. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128234914000023

[2] V. Carceler. (n.d.) Ud2 — apuntes: Arquitectura de computadoras (na1). Apuntes en línea. [Online]. Available: https://elpuig.xeill.net/Members/vcarceler/c1/didactica/apuntes/ud2/na1

[3] K. C. Louden, *Compiler Construction: Principles and Practice*. Boston, MA: PWS Publishing Company, 1997.

[4] R. Maldonado. (2024, Aug.) ¿qué es un compilador cruzado y cómo usarlo? KeepCoding. Blog post. [Online]. Available: https://keepcoding.io/blog/que-es-un-compilador-cruzado-y-como-usarlo/

[5] R. Awati and R. Sheldon, "What is a compiler?" https://www.techtarget.com/whatis/definition/compiler, 2025, consultado el 15 de septiembre de 2025.

[6] E. Staff. (2024, Jan.) Native compiler. DevX. [Online]. Available: https://www.devx.com/terms/native-compiler/

[7] N. Kumar, S. K. Sharma, A. Agarwal, and E. M. K. Jain, *Fundamentals of Automata Theory and Compiler Construction*. IK International Publishing House, 2022.

[8] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a Compiler*. Boston, MA: Addison-Wesley, 2010.

[9] I. Sulbarán, "¿qué es arquitectura de computadoras?" *Tiffin University Blog*, Feb. 2025, 10/02/2025. [Online]. Available: https://global.tiffin.edu/blog/que-es-arquitectura-de-computadoras

[10] D. Patterson. (2023) Lecture 27: Compiler correctness. Northeastern University, Department of Computer Science. Logic & Computation Course Notes. [Online]. Available: https://course.ccs.neu.edu/cs2800sp23/l27.html

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston: Addison-Wesley, 2008.

[12] R. Toal, "Intermediate representations," https://cs.lmu.edu/ ray/notes/ir/, n.d., accedido: 15 de noviembre de 2025.

[13] M. J. Murdocca and V. P. Heuring, *Principios de Arquitectura de Computadoras*. Buenos Aires: Prentice Hall Argentina, 2002.

[14] Electrónica Online. (2024) ¿qué es una palabra en arquitectura informática? Electrónica Online – Definición de "palabra". [Online]. Available: https://electronicaonline.net/definicion/palabra/

[15] C. Bedell. (2023, Dec.) What are big-endian and little-endian? TechTarget, SearchNetworking. [Online]. Available: https://www.techtarget.com/searchnetworking/definition/big-endian-and-little-endian

[16] Cemendil, *Tutorial de ensamblador AT&T para microprocesadores x86 — [0x02]*, Oct 2003, versión 0.91. [Online]. Available: https://osaberdigital.com.br/wp-content/uploads/2024/11/virus_-assembly02.pdf

[17] S. Wolchok. (2021, Mar.) How to read arm64 assembly language. Artículo en línea. [Online]. Available: https://wolchok-org.translate.goog/posts/how-to-read-arm64-assembly-language/?_x_tr_sl=en_x_tr_tl=es_x_tr_hl=es_x_tr_pto=tc