ST0244 Lenguajes de Programacion
Introducción

Andrés Sicard Ramírez

Universidad EAFIT

Semestre 2024-1

Pacto pedagógico

Como miembros de la Universidad EAFIT, nos comprometemos a actuar de manera íntegra siguiendo los más altos estándares éticos y morales.

- Respeto
- Tolerancia
- Honradez
- Compromiso

Pacto pedagógico

Página web del curso

`http://www1.eafit.edu.co/asr/cursos/st0244-lenguajes-de-programacion`

Pacto pedagógico

Página web del curso

`http://www1.eafit.edu.co/asr/cursos/st0244-lenguajes-de-programacion`

Programa de la materia

El programa de la materia está en EAFIT Interactiva.

Pacto pedagógico

Página web del curso

`http://www1.eafit.edu.co/asr/cursos/st0244-lenguajes-de-programacion`

Programa de la materia

El programa de la materia está en EAFIT Interactiva.

Conducto regular, fechas y porcentajes de las evaluaciones

La información está en la página web del curso.

Pacto pedagógico

Página web del curso

http://www1.eafit.edu.co/asr/cursos/st0244-lenguajes-de-programacion

Programa de la materia

El programa de la materia está en EAFIT Interactiva.

Conducto regular, fechas y porcentajes de las evaluaciones

La información está en la página web del curso.

Responsabilidad compartida

- Profesor
- Estudiantes

Pacto pedagógico

Orientaciones para el curso

- Se recomienda seis horas de trabajo por semana (dos horas por cada hora de clase).
- Las clases son presenciales.
- La evaluación a la docencia es obligatoria.
- Se recomienda revisar periódicamente los canales de comunicación institucionales (EAFIT Interactiva, correo institucional, Microsoft Teams).
- Se pueden sacar las notas de clase (en papel o digitales) durante los exámenes parciales.
- Las prácticas no se pueden realizar de manera individual y se deben realizar máximo entre dos estudiantes.

Preliminares

Convenciones

- La numeración (capítulos, teoremas, figuras, páginas, etc) en estas diapositivas corresponde a la numeración del texto guía [Lee 2017].
- Los ejemplos que incluyen código fuente están en el repositorio del curso.

Primer párrafo del texto guía

«A career in computer science is a commitment to a lifetime of learning. You will not be taught every detail you will need in your career while you are a student. The goal of a computer science education is to give you the tools you need so you can teach yourself new languages, frameworks, and architectures as they come along.» (pág. v)

Observaciones iniciales

Tomado de: «The Next 7000 Programming Languages» [Chatley, Donaldson y Mycroft 2019].

Observaciones iniciales

Tomado de: «The Next 7000 Programming Languages» [Chatley, Donaldson y Mycroft 2019].

Evolución:

> *«Language implementations have evolved to help humans manage this complexity.» (pág. 255)*

(continua en la próxima diapositiva)

Observaciones iniciales

¿Un lenguaje de programación universal?

> *«We might hope for a single universal language which is suitable for all niches, as has been a recurring hope since Landin's time. However, the evolutionary model does not predict this. It says nothing about the existence of such a language, and past attempts to create universal languages do not add encouragement.» (pág. 279)*

(continua en la próxima diapositiva)

Observaciones iniciales

¿Cuál lenguaje de programación debería usar?

> *«Another decision point in choosing a language is "get it working" versus "get it right" versus "get it fast/efficient". In different situations, each might be appropriate, and the software-system context, or niche, determines the fitness of individual languages and hence guides the language choice. A quick script to do some data-processing is obviously quite different from an I/O driver, or the control system of a safety-critical device.» (pág. 255)*

(continua en la próxima diapositiva)

Popularidad de los lenguajes de programación

Top 10

- Índice TIOBE: https://www.tiobe.com/tiobe-index/
- GitHub: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

Programa del curso

- Introducción
- Sintaxis
- Programación orientada a objetos
- Programación funcional
- Programación lógica

Paradigmas de programación

Definición

Un **lenguaje de programación** es

> *«A notation for the precise description of computer programs or algorithms. Programming languages are artificial languages, in which the syntax and semantics are strictly defined.» (Oxford Dictionary of Computing, 7 ed.)*

Paradigmas de programación

Definición

Un **paradigma** es

> *«Teoría o conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra la base y modelo para resolver problemas y avanzar en el conocimiento.» (Diccionario de la RAE)*
>
> *«A model of something, or a very clear and typical example of something.» (Cambridge Dictionary)*

Paradigmas de programación

Descripción

Los paradigmas de programación son:

> *«Ways of thinking about programming.» (pág. v)*

> *«High-level approaches for viewing computation.» (Turbark y Gifford 2008, pág. 16)*

> *«A way to classify programming languages based on their features.» (Wikipedia, 2024-01-23)*

Paradigmas de programación

Motivación

Evitar un sesgo cognitivo al usar una sola forma de programar:

«Si sólo tienes un martillo, todo parece un clavo».

Paradigmas de programación

Un espectro de los lenguajes de programación

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, ...
object-oriented	Smalltalk, Eiffel, Java, ...
scripting	Perl, Python, PHP, ...

Observación

Como es señalado por el autor de la figura arriba [Scott 2015, Fig. 1.1], las categorías son borrosas y están abiertas a debate.

Perspectiva histórica

Observación

El desarrollo de los lenguajes de programación tiene como base tanto los desarrollos teóricos como los desarrollos tecnológicos.

Perspectiva histórica

Línea de tiempo*

c. 1675 Gottfried Wilhelm Leibniz. *Characteristica universalis* (un lenguaje simbólico universal).

1822 Charles Babbage. *Difference engine* (máquina mecánica para tabular funciones polinómicas).

1928 David Hilbert y Wilhelm Ackermann. *The Entscheidungsproblem* (el problema de decisión) [Hilbert y Ackermann 1950].

1935-6 Alonzo Church. Cálculo lambda (modelo de computación) y la solución negativa al *Entscheidungsproblem* [Church 1935, 1936].

1936-7 Alan Turing. Turing machine (modelo de computación) y la solución negativa al *Entscheidungsproblem* [Turing 1936-1937].

(continua en la próxima diapositiva)

*A línea de tiempo debe comenzar en algún momento y por lo tanto es necesariamente incompleta.

Perspectiva histórica

Línea de tiempo (continuación)

1939 John Atanasoff y Clifford Berry. El ABC (Atanasoff-Berry computer). Estados Unidos.

c. 1940 Alonzo Church, Alan Turing y Stephen Kleene. La tesis de Church-Turing-Kleene.

1943 Tommy Flowers. El computador Colossus. Inglaterra

1945 John von Neumann. Arquitectura de von Neumann (hay controversias acerca de los autores de esta idea).

1946 John Mauchly y J. Presper Eckert. El ENIAC (Electronic Numerical Integrator and Computer). Estados Unidos.

1949 Alan Turing. Diseño de programas almacenados y verificación de programas [Turing 1949].

(continua en la próxima diapositiva)

Perspectiva histórica

Línea de tiempo (continuación)

1957 John Backus y otros. FORTRAN [Backus, Beeber, Best, Goldberg, Haibt, Herrick, Nelson, Sayre, Sheridan, Stern, Ziller, Hughes y Nutt 1957].

1958 John McCarthy. Lisp [McCarthy 1960].

1960 John Backus y otros. ALGOL 60 [Backus, Bauer, Green, Katz, McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, Wijngaarden y Woodger 1960].

c. 1960 John Backus y Peter Naur. BNF (Backus-Naur Format) (un meta-lenguaje para escribir lenguajes de programación).

1965 J. A. Robinson. El principio de resolución [Robinson 1965].

1972 Alain Colmerauer y Philippe Roussel. Prolog.

Perspectiva histórica

Lectura

Leer la breve historia de los lenguajes de programación C, C++, Java, Prolog, Python y Standard ML en el texto guía.

Modelos de computación

La arquitectura de von Neumann*

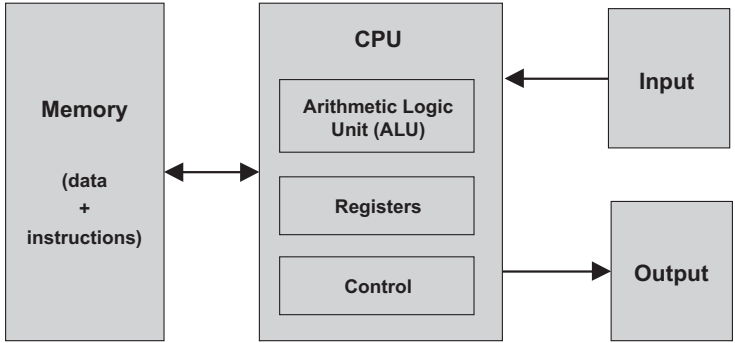


*Fig. 5.1 en [Nisan y Shimon 2005].

Modelos de computación: El modelo imperativo

Características

- División de un programa en sub-programas (funciones, procedimientos, sub-rutinas).
- Programación estructural (diseño top-down o bottom-up).
- Registros de activación para funciones y procedimientos.
- División del área de datos (el *heap*, la *static area* y el *run-time stack*).

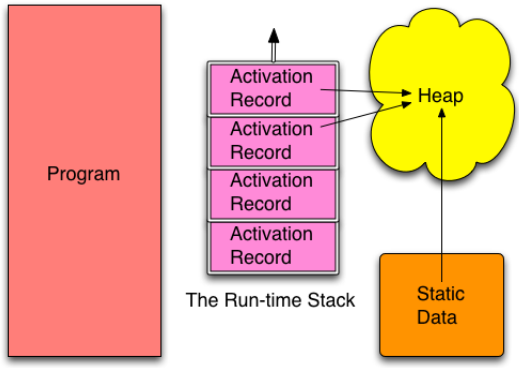


Fig. 1.4

Modelos de computación: El modelo imperativo

Registros de activación para funciones y procedimientos

- Variables locales.
- La dirección de retorno (el valor del PC (*program counter*) antes de que la función o el procedimiento fueran llamados).
- Valor de los parámetros.

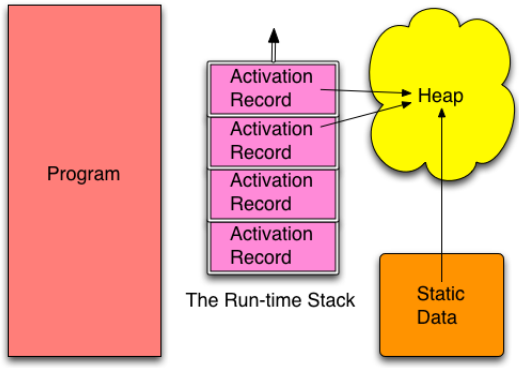


Fig. 1.4

Modelos de computación: El modelo imperativo

División del área de datos

- La *static (global) area*
  Área para almacenar datos y funciones que son globalmente accesibles por el programa (constantes, variables globales, funciones *built-in*).
- El *run-time stack*
  Área para almacenar los registros de activación usando un orden LIFO (*Last In, First Out*).
- El *heap*
  Área para asignación de memoria dinámica (datos creados en tiempo de ejecución).

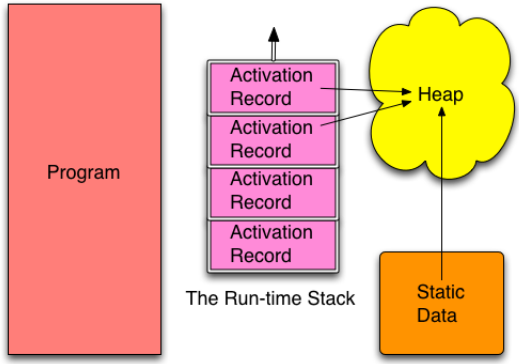


Fig. 1.4

Modelos de computación: El modelo funcional

Características

- Datos persistentes (inmutables) (no pueden ser modificados una vez creados).
- Las funciones son ciudadanos de primera clase (*first-class citizens*).
- No hay diferencia entre el programa y los datos.
- Desde que todo el trabajo es hecho invocando funciones, el *run-time stack* es más importante que en el modelo imperativo. model.
- El programador no interactúa con el *heap*.
- La programación funcional es más abstracta (bueno) pero el programador tiene menos

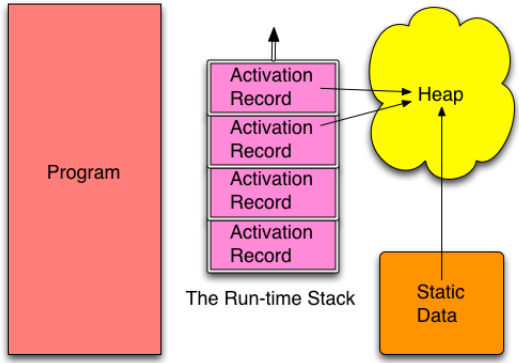


Fig. 1.4

Modelos de computación: El modelo lógico

Características

- El programador no escribe un programa sino una base de datos con hechos y reglas (ambos son axiomas desde el punto de vista lógico).

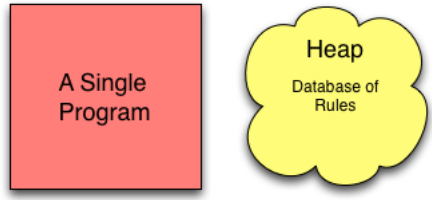


Fig. 1.5

Implementación de lenguajes

Definición

El **lenguaje de máquina** (*machine language*) es el lenguaje binario que es leído, interpretado y ejecutado por la CPU.

Observación

Los lenguaje de máquina son dependientes del *hardware*.

Implementación de lenguajes

Definición

Un **lenguaje ensamblador** (*assembly language*) es una representación símbolo (leíble por los humanos) del lenguaje de máquina.

Observación

Los lenguajes ensambladores son dependientes del *hardware*.

Ejemplo

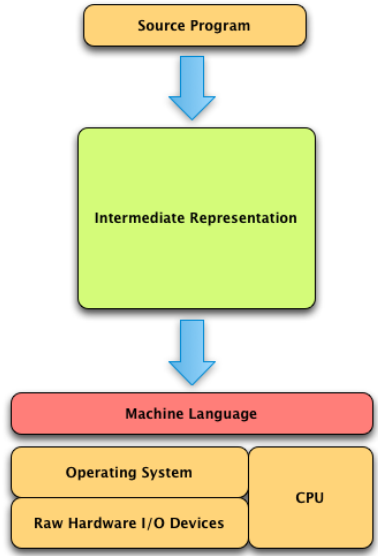
Mirar el archivo `hw/hello-world.asm`.

Implementación de lenguajes

Los lenguajes pueden ser implementados de direrentes maneras

- Compilados
- Interpretados
- Combinación de compilación e interpretación

(Fig. 1.11)



Implementación de lenguajes

Pregunta

La implementación de un lenguaje de programación depende del paradigma al cual pertenece?

Implementación de lenguajes

Pregunta

La implementación de un lenguaje de programación depende del paradigma al cual pertenece?
No!

Implementación de lenguajes

Pregunta

La implementación de un lenguaje de programación depende del paradigma al cual pertenece? No!

Definición

Una **plataforma** (*platform*) es una combinación especifica de *hardware* y sistema operacional.

Implementación de lenguajes: Compilación

Definición

Un **compilador** es un programa que traduce un programa fuente a lenguaje de máquina.

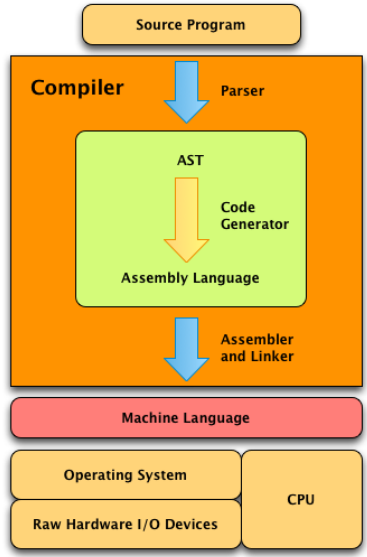
Características

- *Abstract syntax tree (AST)*: Representación interna del programa fuente.
- Si el programa fuente cambia éste debe ser recompilado.

Observación

Los compiladores son dependientes de la plataforma.

(Fig. 1.12)



Implementación de lenguajes: Compilación

Ejemplo

C, C++, COBOL, Fortran, Haskell, Pascal y Rust son lenguajes compilados.

Implementación de lenguajes: Interpretación

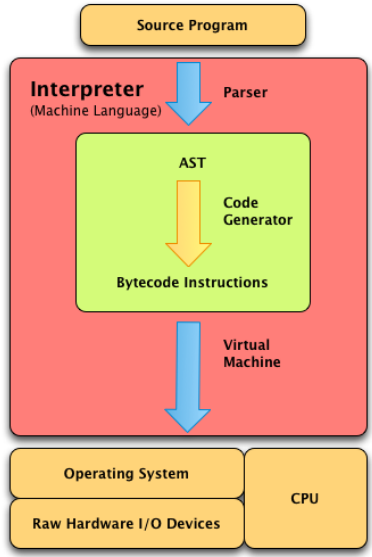
Definición

Un **interpretador** es un programa que ejecuta programas fuente.

Características

- El usuario ejecuta su programa ejecutando el interpretador.
- Ventaja: Portabilidad (el interpretador aísla el programa fuente de la plataforma).
- Desventaja: Velocidad de ejecución.

(Fig. 1.13)



Implementación de lenguajes: Interpretación

Observación

Los interpretadores son dependientes de la plataforma.

Ejemplo

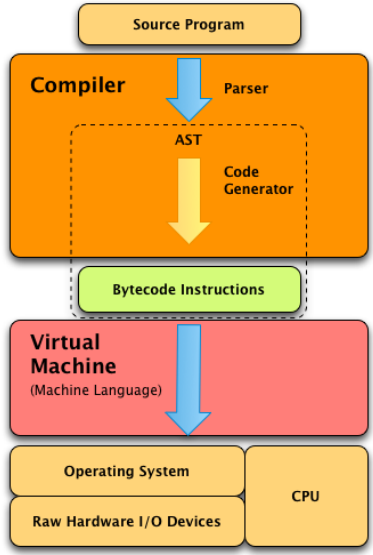
Bash, Haskell, Lisp, Prolog, Python, Ruby y Standard ML son lenguajes interpretados.

Implementación de lenguajes: Máquinas virtuales

Definición

> «A ***virtual machine*** *is a program that provides insulation from the actual hardware and operating system of a machine while supplying a consistent implementation of a set of low-level instructions, often called* ***bytecode****.» (pág. 23)*

(Fig. 1.14)

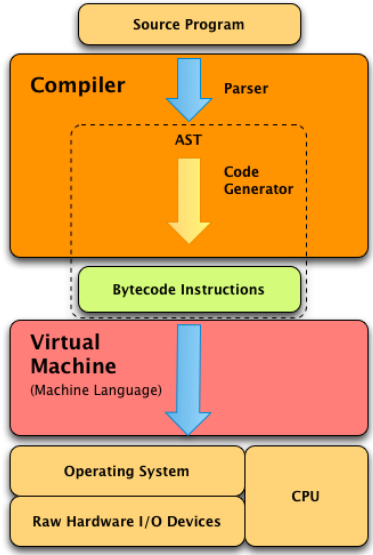


Implementación de lenguajes: Máquinas virtuales

Características

- El programa fuente es compilado a *bytecode*.
- El código *bytecode* es interpretado.
- La interpretación de *bytecode* es más rápida que la interpretación de código fuente.
- Los programas implementados vía máquinas virtuales son más portables que los programas compilados.

(Fig. 1.14)



Implementación de lenguajes: Máquinas virtuales

Observación

Las máquinas virtuales son dependientes de la plataforma.

Observación

Las instrucciones *bytecode* son independientes de la plataforma.

Ejemplo

C#, Java, Python, Standard ML y Visual Basic.Net son lenguajes implementados por máquinas virtuales.

Tipos y chequeo de tipos

Tipos en los lenguajes de programación

> *«They [programming lenguajes] define types to specify which operations make sense on which types of data.» (pág. 26)*

> *«A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.» (Kiselyov y Shan 2008, pág. 8)*

Tipos y chequeo de tipos

Sistemas de tipos en los lenguajes de programación

> *«A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.»* *(Pierce 2002, pág. 1)*

Tipos y chequeo de tipos

Chequeo de tipos estático vs chequeo de tipos dinámico*

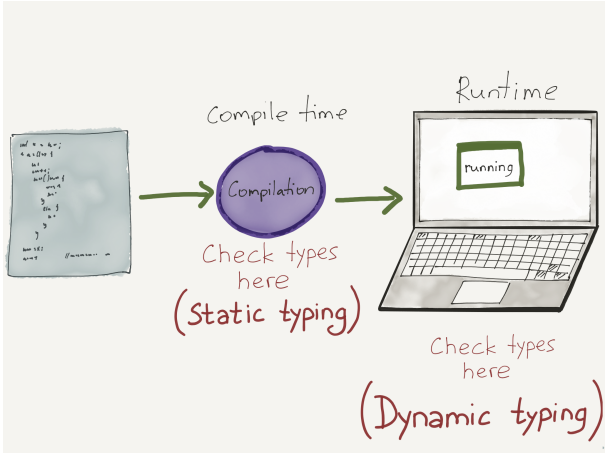


*Figura tomada de en.hexlet.io/courses/intro_to_programming/lessons/types/theory_unit.

Tipos y chequeo de tipos

Ejemplo

- Lenguajes dinámicamente tipeados
  JavaScript, PHP y Python.
- Lenguajes estáticamente tipeados
  C, C++, C#, Haskell, Java, Rust y Standard ML.

Tipos y chequeo de tipos

Discusión

«Which is better, dynamically or statically typed lenguajes? It depends on the complexity of the program you are writing and its size. Static typing is certainly desirable if all other things are equal. But static typing typically does increase the work of a programmer up front. On the other hand, static typing is likely to decrease the amount of time you spend testing.» (pág. 27)

Referencias

- Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Wijngaarden, A. van y Woodger, M. (1960). Report on the Algorithmic Language ALGOL 60. Communications of the ACM 3.5. Ed. por Naur, Peter, págs. 299-314. DOI: 10.1145/367236.367262 (vid. pág. 24).
- Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A. y Nutt, R. (1957). The FORTRAN Automatic Coding System. En: Proceedings Western Joint Computer Conference, págs. 188-198 (vid. pág. 24).
- Chatley, Robert, Donaldson, Alastair y Mycroft, Alan (2019). The Next 7000 Programming Languages. En: Computing and Software Science. State of the Art and Perspectives. Ed. por Steffen, Bernhard y Woeginger, Gerhard. Vol. 10000. Lecture Notes in Computer Science. Springer, págs. 250-282. DOI: 10.1007/978-3-319-91908-9_15 (vid. págs. 10, 11).
- Church, Alonzo (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, págs. 332-333. DOI: 10.1090/S0002-9904-1935-06102-6 (vid. pág. 22).

Referencias

Church, Alonzo (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, págs. 345-363. DOI: 10.2307/2371045 (vid. pág. 22).

Hilbert, D. y Ackermann, W. [1938] (1950). Principles of Mathematical Logic. 2.ª ed. Translation of the second edition of *Grundzüge der Theoretischen Logik*, Springer, 1938. Translated by Lewis M. Hammond, George G. Leckie and F. Steinhardt. Edited and with notes by Robert E. Luce. Chelsea Publising Company (vid. pág. 22).

Kiselyov, Oleg y Shan, Chung-chieh (2008). Interpreting Types as Abstract Values. Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008) (vid. pág. 45).

Lee, Kent D. [2014] (2017). Foundations of Programming Languages. 2.ª ed. Undergraduate Topics in Computer Science. Springer (vid. pág. 8).

McCarthy, John (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. Communications of the ACM 3.4, págs. 184-195. DOI: 10.1145/367177.367199 (vid. pág. 24).

Nisan, Noam y Shimon, Schocken (2005). The Elements of Computing Systems. Building a Modern Computer from First Principles. MIT Press (vid. pág. 26).

Pierce, Benjamin C. (2002). Types and Programming Languages. MIT Press (vid. pág. 46).

Referencias


Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12.1, págs. 23-41. DOI: 10.1145/321250.321253 (vid. pág. 24).


Scott, Michael L. [1999] (2015). Programming Language Pragmatics. 4.ª ed. Morgan Kaufmann (vid. pág. 20).


Turbark, Franklyn y Gifford, David (2008). Design Concepts in Programming Languages. MIT Press (vid. pág. 18).


Turing, Alan M. (1936-1937). On Computable Numbers, with an Application to the Entscheidungsproblem. Proceeding of the London Mathematical Society s2-42, págs. 230-265. DOI: 10.1112/plms/s2-42.1.230 (vid. pág. 22).


— (1949). Checking a Large Routine. En: Report of a Conference on High Speed Automatic Calculating (vid. pág. 23).