Índice general

Capítulo 1

Qué es y qué puede hacer un lenguaje de programación

Un lenguaje de programación es un lenguaje formal diseñado para realizar procesos que pueden ser llevados a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina o para expresar algoritmos con precisión.

1.1. Sintáxis y Semántica

Los lenguajes son sistemas que se sirven de una forma para comunicar un significado. Lo que tiene que ver con la forma recibe el nombre de sintaxis y lo que tiene que ver con el significado recibe el nombre de semántica.

En los lenguajes de programación, que son lenguajes artificiales creados por hombres (lenguajes formales), la forma son los programas y el significado es lo que los prográmas hacen, usualmente, en una computadora. En la definición de arriba, se ha descrito lo que los programas como "controlar el comportamiento físico y lógico de una máquina". Un lenguaje de programación se describe con su sintaxis (qué es lo que se puede escribir legalmente en ese lenguaje) y su semántica (qué efectos tiene en la máquina lo que se escribe en ese lenguaje).

La implementación de un lenguaje de programación debe transformar la sintáxis de un programa en instrucciones de máquina que se pueden ejecutar para que suceda la secuencia de acciones que se pretendía. El compilador hace esa traducción, un intérprete puede combinar traducción y ejecución.

1.2. Sintáxis a traves de gramáticas

Vamos a estar usando gramáticas independientes de contexto, más específicamente EBNF, que es el estándar de facto para estas gramáticas; aunque algunas propiedades de los lenguajes de programación escapan a su expresividad.

Un gramática es un método para definir conjuntos infinitos de expresiones y procesar expresiones. Consisten de:

- símbolo inicial
- no terminales
- terminales
- producciones

Los no terminales son la forma adecuada de describir la composicionalidad de las expresiones, no pueden formar parte de una expresión, siempre se tienen que substituir por terminales.

Capítulo 2

Cómo funcionan los lenguajes de programación

2.1. Compilador

El compilador es un programa que lee un programa escrito en un lenguaje origen y lo traduce a un programa equivalente en un lenguaje destino, normalmente el lenguaje origen es de alto nivel y el destino es de bajo nivel. El mismo posee dos componentes:

- Entender el programa (asegurarse de que es correcto)
- Reescribir el programa

2.1.1. Fases de un compilador

1. Análisis léxico: se divide un programa (secuencia de caracteres) en palabras (tokens).
2. Análisis sintáctico: comprueba si la secuencia de tokens conforma a la especificación gramatical del lenguaje y genera el árbol sintáctico. La especificación gramatical suele representarse con una gramática independiente de contexto, que también le da forma al árbol sintáctico.
3. Análisis semántico: El compilador trata de ver si un un programa tiene sentido analizando su árbol sintáctico. Un programa sin errores gramaticales no siempre es correcto, puede haber problemas de tipo:

$$pos = init + rate * 60 \tag{2.1}$$

   Qué pasa si pos es una clase y init y rate son enteros. El parser no puede encontrar este tipo de errores, el análisis semántico encuentra este tipo de error.

   El compilador hace comprobaciones semánticas estáticas (static semantic checks):

   - comprobación de tipos
   - declaración de variables antes de su uso
   - se usan los identificadores en contextos adecuados
   - comprobar argumentos

   Si hay un fallo en compilación, se genera un error en tiempo de ejecución (dynamic semantic checks) se comprueba:

   - que los valores de los arreglos estén dentro de los límites
   - errores aritméticos (división por 0)
   - no se desreferencian los punteros si no apuntan a un objeto válido
   - se usan variables sin inicialización
   - si hay un fallo en ejecución, se levanta una excepción

Tipado fuerte

Un lenguaje tiene tipado fuerte si siempre se detectan los errores de tipo:

- En tiempo de compilación o de ejecución
- Tipado fuerte: Ada, Java, ML, Haskell
- Tipado débil: Fortran, Pascal, C/C++, Lisp
- Duck typing: Python
  El tipado fuerte hace que el lenguaje sea más seguro y fácil de usar sin errores, pero potencialmente más lento por las comprobaciones dinámicas. En algunos lenguajes algunos errores de tipo se detectan tarde, lo que los hace poco fiables (Basic, Lisp, Prolog, lenguajes de scripting).

4. Generación y optimización de código intermedio: El código intermedio está cerca de la máquina pero sigue siendo fácil de manipular, para poder implementar optimizaciones. Por ejemplo:

$$temp1 = 60 \tag{2.2}$$
$$temp2 = id3 + temp1 \tag{2.3}$$
$$temp3 = id2 + temp2 \tag{2.4}$$
$$id1 = temp3 \tag{2.5}$$

   Se puede optimizar (independientemente de máquina):

$$temp1 = id3 * 60{,}0 \tag{2.6}$$
$$id1 = id2 + temp1 \tag{2.7}$$

5. Generación y optimización de código destino: De la forma independiente de máquina se genera ensamblador:

$$MOVF id3, R2 \tag{2.8}$$
$$MULF 60{,}0, R2 \tag{2.9}$$
$$MOVF id2, R1 \tag{2.10}$$
$$ADDF R2, R1 \tag{2.11}$$
$$MOVF R1, id1 \tag{2.12}$$

   Este código específico de máquina se optimiza para explotar características de hardware específicas.

2.2. Paradigma Imperativo

Un paradigma de programacón es una configuracón frecuente de características de lenguajes de programacón. El paradigma imperativo es el más antiguo y el que estuvo siempre más pegado a la máquina. Tradicionalmente se ha opuesto al paradigma funcional, pero la mayor parte de lenguajes integran ideas de ambos paradigmas.

2.2.1. Conceptos fundamentales

- Operación básica: **asignación**. La asignación tiene efectos secundarios ya que cambia el estado de la máquina.
- Sentencias de **control** de flujo: condicionales y sin condición (GO TO), ramas, ciclos.
- Bloques, para obtener **referencias locales**
- **Parametrización**

2.2.2. Elementos básicos

1. Definiciones de **tipos**
2. Declaraciones de **variables** (normalmente tipadas)
3. Expresiones y sentencias de **asignación**
4. Sentencias de **control de flujo** (normalmente estructuradas)
5. **Alcance léxico** y bloques, para poder tener variables con referencias locales
6. Declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

2.2.3. Declaraciónes de variables

Las declaraciones tipadas restringen los posibles valores de una variable en la ejecución del programa:

- Jerarquía de tipos built-in o personalizada
- inicialización

Uso de memoria: cuánto espacio de memoria reservar para cada tipo de variable.

2.2.4. Ubicación y valores de variables

Al declarar una variable la estamos ligando a una **ubicación en memoria**, de forma que el nombre de la variable es el identificador de la ubicación en memoria. La ubicación puede ser global, en la pila o en el heap.

- l-valor: ubicación en memoria (dirección de memoria)
- r-valor: valor que se guarda en la ubicación de memoria identificada por el l-valor

La asignación: A (objetivo) = B (expresión), es una actualización destructiva, ya que reescribe la ubicación de memoria identificada por A con el valor de la expresión B.

2.2.5. Semántica de copia vs. Semántica de referencia

- Semántica de copia: la expresión se evalúa a un valor, que se copia al objetivo (lenguajes imperativos).
- Semántica de referencia: la expresión se evalúa a un objeto, cuyo puntero se copia al objetivo (lenguajes orientados a objetos).

2.2.6. Variables y asignación

En la parte derecha de una asignación está el r-valor de la variable, en la parte izquierda está su l-valor. Una expresión que no tenga un l-valor no puede aparecer en la parte izquierda de una asignación. El r-valor de un puntero es el l-valor de otra variable (el valor de un puntero es una dirección).

- las constantes sólo tienen r-valor
- las funciones sólo tienen l-valor

L-valor y R-valor en C

- &x devuelve el l-valor de x
- *p devuelve el r-valor de p. Si p es un puntero, esto es el l-valor de otra variable

2.2.7. Control de flujo estructurado

Se piensa como secuencial, las instrucciones se ejecutan en el orden en el que están escritas, en algunos casos soporta ejecución concurrente.

Un programa es estructurado si el flujo de control es evidente en la estructura sintáctica del texto del programa. Esto es útil para poder razonar intuitivamente leyendo el texto del programa, ya que se crean construcciones del lenguaje para patrones comunes de control: iteración, selección, procedimientos, funciones, etc.

Estilo moderno

Construcciones estándar que estructuran los saltos:

- if ... then ... else ... end
- while ... do ... end
- for .....
- case ...

Se agrupa el código en bloques lógicos, se evitan saltos explícitos (excepto retorno de función), no se puede saltar al medio de un bloque o función.

Capítulo 3

Estructura en bloques

3.1. Estructura de bloque

Manejo de memoria

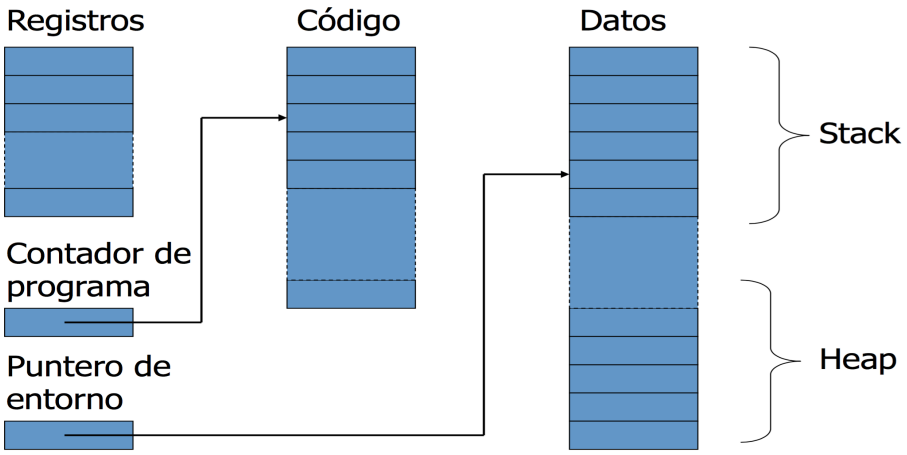
- Al stack tiene los datos sobre entrada y salida de bloques
- El heap tiene datos de diferente lifetime
- El puntero de entorno (environment) apunta a la posición actual en el stack
- Al entrar a un bloque: se añade un nuevo activation record al stack
- Al salir de un bloque: se elimina el activation record más reciente del stack

3.1.1. Alcance y lifetime

- Alcance: región del texto del programa donde una declaración es visible.
- Lifetime: período de tiempo en que una ubicación de memoria es asignada a un programa.

3.2. Activation records

Para describir la semántica de los lenguajes de programación, más específicamente lenguajes estructurados por bloques, usamos una semántica operacional, es decir, describimos los efectos de las diferentes expresiones del lenguaje sobre una máquina. Para eso usamos un modelo simplificado de la computadora, el que se ve en la figura.

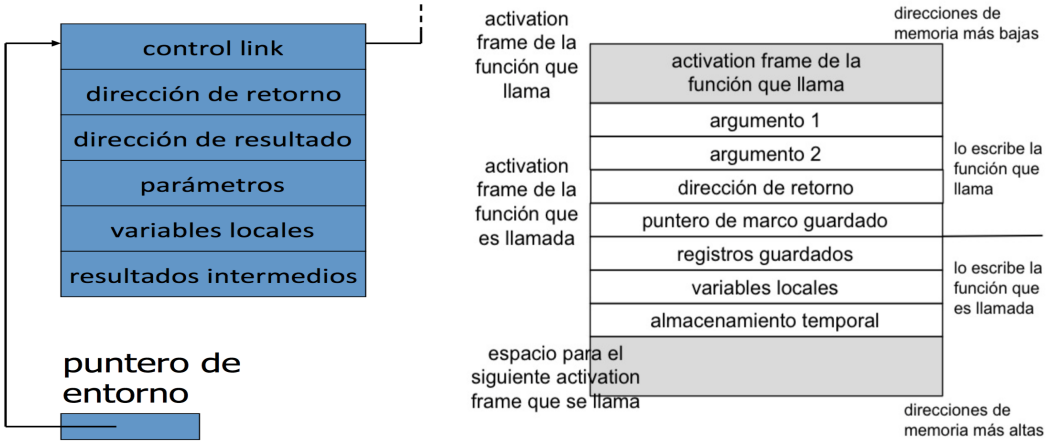


Vemos que este modelo de máquina separa la memoria en la que se guarda el código (**stack**), de la memoria en la que se guardan los datos (**heap**). Usamos dos variables para saber a qué parte de la memoria necesitamos acceder en cada momento de la ejecución del programa: el **contador de programa**, que es una dirección en la parte de la memoria donde se guarda el código, en concreto, la dirección donde se encuentra la instrucción de programa que se está ejecuntando actualmente y y el **puntero de entorno**, el cual nos sirve para saber cuáles son los valores que se asignan a las variables que se están usando en una parte determinada del código.

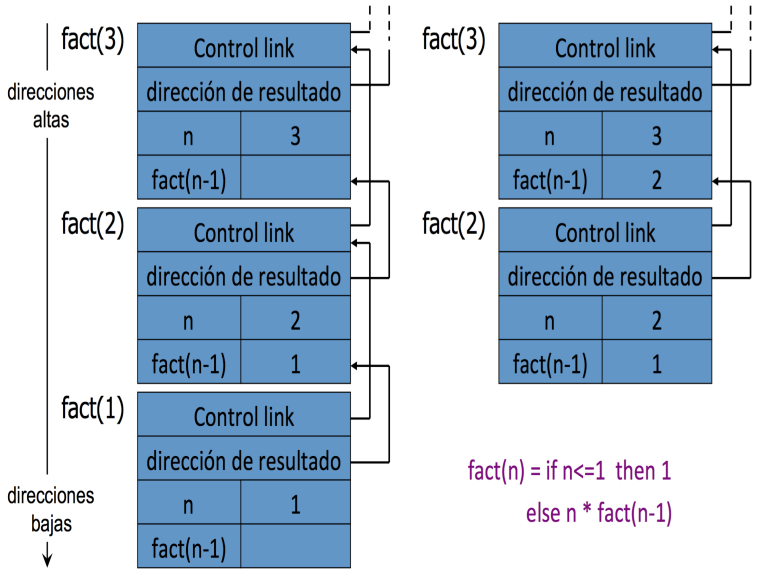
Cuando el programa entra en un nuevo bloque, el stack se encarga de agregar una estructura de datos que se llama ***activation record***, que contiene el espacio para las variables locales declaradas en el bloque, normalmente, por la parte de arriba de la pila. Entonces, el puntero de entorno apunta al nuevo activation record.

Cuando el programa sale del bloque, se retira el activation record de la pila y el puntero de entorno se restablece a su ubicación anterior, es decir, al puntero de entorno correspondiente a la función que llamaba a la función que ha sido desapilada. El activation record que se apila más recientemente es el primero en ser desapilado, a esto se le llama disciplina de pila.

Ademas el activation record posee espacio para guardar resultados intermedios, en el caso de que sea necesario. Podemos observar, en la figura siguiente, que hay dos direcciones de memoria donde se guardan datos importantes: el control link, contiene el que será el puntero de entorno cuando se desapile el activation record actual y la dirección de memoria distinguida es la llamada dirección de retorno, que es donde se va a guardar el resultado de la ejecución de la función, si es que lo hay.



Ejemplo de activation record



Capítulo 4

Control de la ejecución

4.1. Pasaje de parámetros

1. Pasaje por valor:
   - La función que llama pasa el r-valor del argumento a la función. Es necesario computar el valor del argumento en el momento de la llamada. Reduce el *aliasing* (dos identificadores para una sola ubicación en memoria)
   - La función no puede cambiar el valor de la variable de la función que llama
   - Ejemplos: C, Java, Scheme. Se pueden pasar punteros si queremos que se pueda modificar el valor de la variable de la función que llama
2. Pasaje por referencia:
   - La función que llama pasa el l-valor del argumento a la función. Se asigna la dirección de memoria del argumento al parámetro. Aumenta el *aliasing*
   - La función puede modificar la variable de la función que llama
   - Ejemplos: C++, PHP
3. Pasaje por valor-resultado:
   - Intenta tener los beneficios de llamada por referencia (efectos secundarios en los argumentos) sin los problemas de *aliasing*
   - Hace una copia en los argumentos al principio, copia las variables locales a los argumentos actuales al final del procedimiento. Asi los argumentos son modificados.
   - Se comporta como llamada por referencia sin la presencia de *aliasing*
   - Cuidado: el comportamiento depende del orden en que las variables locales se copian.
   - Ejemplos: BBC BASIC V
4. Pasaje por nombre:
   - En el cuerpo de la función se sustituye textualmente el argumento para cada instancia de su parámetro. Se implementó para Algol 60 pero sus sucesores no lo incorporaron
   - Es un ejemplo de ligado tardío. La evaluación del argumento se posterga hasta que efectivamente se ejecuta en el cuerpo de la función. Asociado a evaluación perezosa en lenguajes funcionales (ej: Haskell)
5. Pasaje por necesidad:
   - Variación de call-by-name donde se guarda la evaluación del parámetro después del primer uso
   - Idéntico resultado a call-by-name (y más eficiente) si no hay efectos secundarios
   - El mismo concepto que lazy evaluation

método	qué se pasa	lenguajes	comentarios
por valor (by value)	valor	C, C++	simple, los parámetros que se pasan no cambian, pero puede ser costoso
por referencia (by reference)	dirección	FORTRAN, C++	económico, pero los parámetros pueden cambiar!
por valor-resultado (by value-result)	valor + dirección	FORTRAN, Ada	más seguro que por referencia, pero más costoso
por nombre (by name)	texto	Algol	complicado, ya no se usa

4.2. Alcance

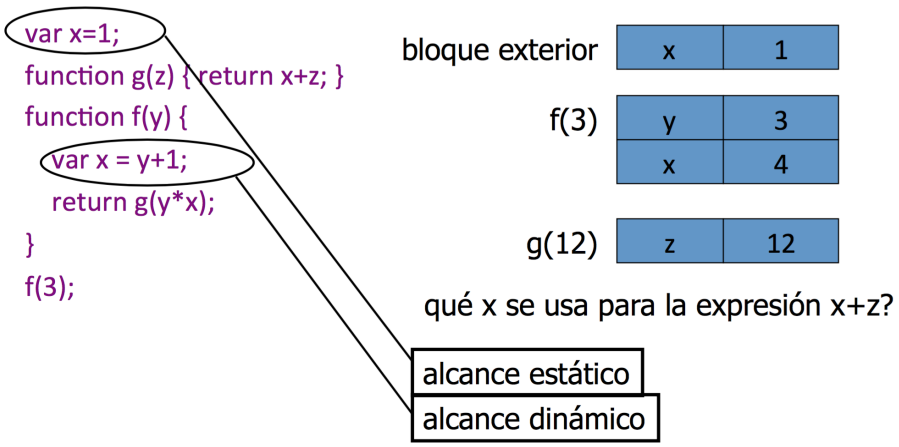
Si un identificador `x` aparece en el cuerpo de una función, pero `x` no se declara dentro de la función, entonces el valor de `x` depende de alguna declaración fuera de la función. En esta situación, la ubicación de `x` está fuera del registro de activación para la función y es una variable global a esa función. Debido a que `x` ha sido declarada en otro bloque, el acceso a una `x` libre o global consiste en encontrar el registro de activación pertinente en la pila.

Hay dos políticas principales para buscar la declaración adecuada de un identificador global:

Alcance estatico: un identificador global se refiere al identificador con ese nombre que se declara en el bloque contenedor más cercano del texto del programa.

Alcance dinamico: un identificador global se refiere al identificador asociado con el registro de activación más reciente.

Aunque la mayoría de los lenguajes actuales de programación de propósito general utilizan alcance estático para las declaraciones de variables y funciones, el alcance dinámico es un concepto importante que se utiliza en lenguajes de propósito específico y en construcciones especializadas como las excepciones. Algunos lenguajes con alcance dinámico son los dialectos antiguos de Lisp, los lenguajes de marcado Tex/LaTex, las excepciones y las macros.



4.3. Alto orden

4.3.1. Funciones de primera clase

Un lenguaje tiene funciones de primera clase si las funciones pueden ser declaradas dentro de cualquier alcance, pasadas como argumentos a otras funciones y, devueltas como resultado de funciones. En un lenguaje

con funciones de primera clase y con alcance estático, un valor de función se representa generalmente por una clausura, un par formado por un puntero al código del cuerpo de una función y otro puntero a un activation record. Veamos un ejemplo de función en ML que requiere a otra función como argumento:

1 fun map (f, nil) = nil — map(f, x::xs) = f(x) :: map(f, xs)

La función map toma una función f y una lista m como argumentos y aplica f a cada elemento de m en orden. El resultado de map(f, m) es la lista de resultados f(x) para elementos x de la lista m. Esta función es útil en muchos programas en los que se usan listas. Por ejemplo, si tenemos una lista de tiempos de vencimiento para una secuencia de eventos y queremos incrementar cada tiempo de vencimiento, podemos hacerlo pasando una función de incremento a map.

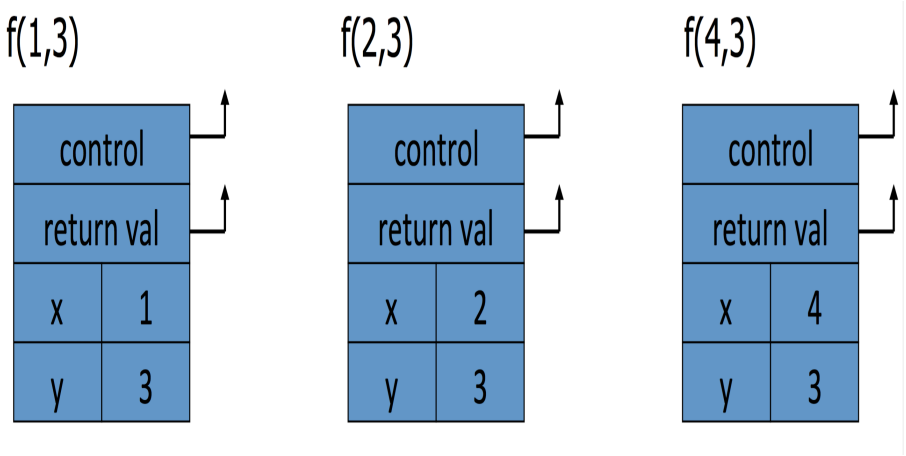
4.3.2. Pasar funciones a otras funciones

4.4. Recursión a la cola

Una optimización que realizan los compiladores es la llamada eliminación de la recursion a la cola. Para las funciones recursivas a la cola, que describimos a continuación, se puede reutilizar un activation record para una llamada recursiva a la función. Esto reduce la cantidad de espacio de la pila que usa una función recursiva, y evita llegar a problemas por límites de hardware como el llamado *stack overflow*, en el que la ejecución de un programa requiere más espacio del que hay disponible en la pila. Una llamada a `f` en el cuerpo de `g` es una llamada de la cola si `g` devuelve el resultado de la llamada `f` sin ningún cálculo adicional. Una función `f` es recursiva de cola si todas las llamadas recursivas en el cuerpo de `f` son llamadas a la cola a `f`.

Veamos como ejemplo la función recursiva a la cola que calcula el factorial:

fun factcola(n,a) = if n ¡= 1 then a else factcola(n-1, n * a);



4.5. Excepciones

Capítulo 5

Orientación a objetos