Capítulo 2

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 (fuente: wikipedia).

2.1. Sintaxis 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 naturales, como el castellano, el inglés o las lenguas de signos, las palabras son la forma, y el contenido proposicional de las oraciones es el significado.

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 programas 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).

2.2. Alcance de los lenguajes de programación

Mitchell 2.

Para "controlar el comportamiento físico y lógico de una máquina" usamos algoritmos, un conjunto de instrucciones bien definidas, ordenadas y finitas que permite realizar algo de forma inambigua mediante pasos.

Las funciones computables son la formalización de la noción intuitiva de algoritmo, en el sentido de que una función es computable si existe un algoritmo que puede hacer el trabajo de la función, es decir, dada una entrada del dominio de la función puede devolver la salida correspondiente.

El concepto de función computable es intuitivo, se usa para hablar de computabilidad sin hacer referencia a ningún modelo concreto de computación. Cualquier definición, sin embargo, debe hacer referencia a algún modelo específico de computación, como por ejemplo máquina de Turings, las funciones μ recursivas, el lambda cálculo o las máquinas de registro.

La tesis de Church-Turing[1] dice que las funciones computables son exactamente las funciones que se pueden calcular utilizando un dispositivo de cálculo mecánico dada una cantidad ilimitada de tiempo y espacio de almacenamiento. De manera equivalente, esta tesis establece que cualquier función que tiene un algoritmo es computable.

Algunas función no computable famosas son *Halting problem* o calcular la Complejidad de Kolmogorov.

2.3. Sintaxis a través de gramáticas

Nuestro objetivo con respecto a la sintaxis de los lenguajes de programación es describir de forma compacta e inambigua el conjunto de los programas válidos en ese lenguaje. El instrumento formal básico para describir la sintaxis de los lenguajes de programación son las gramáticas independientes de contexto. El estándar de facto para gramáticas independientes de contexto de lenguajes de programación es EBNF.

Sin embargo, las gramáticas independientes de contexto no son suficientemente expresivas para describir adecuadamente la mayor parte de lenguajes de programación. Por esa razón, en la práctica el análisis de la forma de los programas suele incorporar diferentes mecanismos, entre ellos, el análisis de un programa mediante una gramática independiente de contexto, que se complementa con otros mecanismos para alcanzar la expresividad propia de una máquina de Turing. El proceso completo de análisis lo realiza el compilador del lenguaje, como se describe en la sección 3.1.

No obstante, las gramáticas independientes de contexto describen el grueso de la forma de los lenguajes de programación.

Una gramática de ejemplo clásica es la que genera los strings que representan expresiones aritméticas con los cuatro operadores $+, -, *, /$ y los números como operandos:

```
<expresion> --> numero
<expresion> --> ( <expresion> )
<expresion> --> <expresion> + <expresion>
<expresion> --> <expresion> - <expresion>
```

[1] La historia de la colaboración entre Church y Turing es un ejemplo muy interesante de como dos individuos geniales quieren y pueden trabajar juntos para potenciarse mútuamente y llegar a alcanzar logros mucho mayores de los que habrían alcanzado cada uno por su lado. Fíjense también en los estudiantes a los que dirigió Church.

```
<expresion> --> <expresion> * <expresion>
<expresion> --> <expresion> / <expresion>
```

El único símbolo no terminal en esta gramática es `expresion`, que también es el símbolo inicial. Los terminales son `{+,-,*,/,(,),numero}`, donde `numero` representa cualquier número válido.

La primera regla (o producción) dice que una `<expresion>` se puede reescribir como (o ser reemplazada por) un número. Por lo tanto, un número es una expresión válida. La segunda regla dice que una expresión entre paréntesis es una expresión válida, usando una definición recursiva de expresión. El resto de reglas dicen que la suma, resta, producto o división de dos expresiones también son expresiones válidas.

Pueden encontrar algunos ejemplos simples de gramáticas independientes de contexto en `https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html`

Algunos fenómenos de los lenguajes de programación no se pueden expresar con naturalidad mediante gramáticas independientes de contexto. Por ejemplo es difícil expresar la obligación de que una variable sea declarada antes de ser usada, o bien describir la asignación múltiple de variables, como podemos hacer por ejemplo en Python: `(foo, bar, baz) = 1, 2, 3`.

> **Pregunta 1:**
> *Piense cómo sería una gramática independiente de contexto para expresar la obligación de que una variable sea declarada antes de ser usada, o la asignación múltiple de variables. ¿Cómo es la gramática resultante? ¿Resulta fácil de pensar, fácil de entender?*

Las gramáticas que describen los lenguajes de programación se usan para verificar la correctitud de un programa escrito en ese lenguaje, mediante un compilador. En un compilador, estas limitaciones se solucionan en parte porque hay módulos de procesamiento del programa posteriores a la gramática que tratan algunos de los fenómenos que las gramáticas no pueden capturar.

Sin embargo, también se usan las gramáticas para describir los lenguajes para **consumo humano**. En ese uso, uno desea que la gramática pueda expresar de forma compacta todas las propiedades relevantes del lenguaje. Para conseguirlo, se suelen usar mecanismos ad-hoc para aumentar su poder expresivo, como por ejemplo predicados asociados a reglas. Por ejemplo, si queremos establecer una restricción de tipos en una determinada regla, podemos hacerlo de la siguiente forma:

```
<expresion> --> numero
<expresion> --> ( <expresion> )
<expresion> --> <expresion> + <expresion>
<expresion> --> <expresion> - <expresion>
<expresion> --> <expresion> * <expresion>
<expresion> --> <expresion> / <expresion> ^ <expresion>
   es_de_tipo Float
```

```
<expresion> --> <expresion> div <expresion> ^ <expresion>
   es_de_tipo Int
```

En la práctica, las restricciones de tipos las realiza el análisis semántico del compilador.
Para ejercitar este mecanismo de expresividad, resuelvan el ejercicio 2.1.

2.4. Semántica operacional vs. lambda cálculo o denotacional

Hay diferentes maneras de describir y manipular formalmente la semántica de un programa. Algunas de ellas son el lambda cálculo de Church (Mitchell 4.2), la semántica denotacional de Strachey & Scott (Mitchell 4.3) y diferentes tipos de semántica operacional. En nuestro curso vamos a estar usando un tipo de semántica operacional.

La semántica operacional (de pequeños pasos) describe formalmente cómo se llevan a cabo cada uno de los pasos de un cálculo en un sistema computacional. Para eso se suele trabajar sobre un modelo simplificado de la máquina.

Cuando describimos la semántica de un programa mediante semántica operacional, describimos cómo un programa válido se interpreta como secuencias de pasos computacionales. Estas secuencias son el significado del programa.

Tal vez la primera formalización de semántica operacional fue el uso del cálculo lambda para definir la semántica de LISP que hizo [John McCarthy. "Funciones recursivas de expresiones simbólicas y su cómputo por máquina, Parte I". Consultado el 2006-10-13.].

Pregunta 2:
¿Por qué puede haber más de un tipo de semántica operacional, mientras que el lambda cálculo y la semántica denotacional son sólo uno?

2.5. Para saber más

[John McCarthy. "Funciones recursivas de expresiones simbólicas y su cómputo por máquina, Parte I". Consultado el 2006-10-13.]

Capítulo 3

Cómo funcionan los lenguajes de programación

3.1. Estructura de un compilador

Mitchell 4.1.1.

3.2. Estructuras de datos de bajo nivel

3.2.1. Variables

Una de las estructuras de datos básicas de los lenguajes de programación son las variables. Una variable está formada por una ubicación en la memoria y un identificador asociado a esa ubicación. Esa ubicación contiene un valor, que es una cantidad o información conocida o desconocida. El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa.

Los compiladores reemplazan los nombres simbólicos de las variables con la real ubicación de los datos. Mientras que el nombre, tipo y ubicación de una variable permanecen fijos, los datos almacenados en la ubicación pueden ser cambiados durante la ejecución del programa.

El identificador en el código fuente puede estar ligado a un valor durante el tiempo de ejecución y el valor de la variable puede por lo tanto cambiar durante el curso de la ejecución del programa. De esta forma, es muy sencillo usar la variable en un proceso repetitivo: puede asignársele un valor en un sitio, ser luego utilizada en otro, más adelante reasignársele un nuevo valor para más tarde utilizarla de la misma manera, por ejemplo, en procedimientos iterativos.

Diferentes identificadores del código pueden referirse a una misma ubicación en memoria, lo cual se conoce como aliasing. En esta configuración, si asignamos un valor a una variable utilizando uno de los identificadores también cambiará el valor al que se puede

acceder a través de los otros identificadores.

Veamos un ejemplo:

```
x: int;
y: int;
x: = y + 3;
```

En la asignación, el valor almacenado en la variable y se añade a 3 y el resultado se almacena en la ubicación para x. Notemos que las dos variables se utilizan de manera diferente: usamos el valor almacenado en y, independientemente de su ubicación, pero en cambio usamos la ubicación de x, independientemente del valor almacenado en x antes de que ocurra la asignación.

La ubicación de una variable se llama su L-valor y el valor almacenado en esta ubicación se llama el R-valor de la variable.

En ML, los L-valores y los R-valores tienen diferentes tipos. En otras palabras, una región de la memoria asignable tiene un tipo diferente de un valor que no se puede cambiar. En ML, un L-valor o región asignable de la memoria se llama celda de referencia. El tipo de una celda de referencia indica que es una celda de referencia y especifica el tipo de valor que contiene. Por ejemplo, una celda de referencia que contiene un número entero tiene tipo `int ref`.

Cuando se crea una celda de referencia, se debe inicializar a un valor del tipo correcto. ML no tiene variables no inicializadas o punteros colgantes. Cuando una asignación cambia el valor almacenado en una celda de referencia, la asignación debe ser coherente con el tipo de la celda de referencia: una celda de referencia de tipo entero siempre contendrá un entero, una celda de referencia de tipo lista contendrá siempre (o hará referencia a) una lista, y así.

ML tiene operaciones para crear celdas de referencia, para acceder a su contenido y para cambiar su contenido: `ref`, `!` y `:=`, que se comportan como sigue:

- `ref v` – crea una celda de referencia que contiene el valor v
- `! r` – devuelve el valor contenido en la celda r
- `r: = v` – ubica el valor v en la celda de referencia r

Capítulo 5

Estructura en bloques

5.1. Código estructurado vs. código *spaghetti*

Mitchell 8.1

En lenguajes como ensamblador o Fortran es muy fácil escribir programas que resulten incomprensibles porque al que lee le cuesta entender la estructura de control del programa. A este tipo de código se le llama código *spaghetti*, y aquí mostramos unos ejemplos:

```
10 IF (X .GT. 0.000001) GO TO 20
   X = -X
11 Y = X*X - SIN(Y)/(X+1)
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X=A
   Y = B-A + C*C
   GO TO 11
   ...
```

En este ejemplo de Fortran de Mitchell (8:204) se ven algunos efectos del código *spaghetti*.

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

Este fragmento de código *spaghetti* podría traducirse a un lenguaje con bloques de la siguiente forma:

```
10 FOR i = 1 TO 10
20     PRINT i; " squared = "; i * i
30 NEXT i
40 PRINT "Program Completed."
50 END
```

Para evitar este problema, la mayoría de los lenguajes de programación modernos proporcionan alguna forma de bloque. Un bloque es una región del texto del programa con inicio y fin explícitos e inambiguos. Esta región del texto permite organizar de forma explícita la lógica del programa. Pero además, posibilita sucesivas abstracciones sobre el flujo de ejecución.

En primer lugar, los bloques nos permiten hacer declaraciones de variables locales. Por ejemplo:

```
{ int x = 2;
  { int y = 3;
    x = y+2;
  }
}
```

En esta sección del código hay dos bloques. Cada bloque comienza con una llave izquierda, `{`, y termina con una llave derecha, `}`[1]. El bloque exterior comienza con la primera llave izquierda y termina con la última llave derecha. El bloque interno se anida en el interior del bloque exterior, comienza con la segunda llave izquierda y termina con la primera llave derecha.

La variable `x` se declara en el bloque exterior y la variable `y` se declara en el bloque interior. Una variable declarada dentro de un bloque se dice que es una variable local para ese bloque. Las variables que no están declaradas en un bloque, sino en algún otro bloque que lo contiene, se dice que es una variable global para el bloque más interno. En este ejemplo, `x` es local en el bloque exterior, `y` es local para el bloque interior, y `x` es global para el bloque interior.

Gracias a la estructura de bloques se pueden implementar mecanismos de gestión de la memoria que permiten, por ejemplo, el llamado a funciones de forma recursiva.

Las versiones de Fortran de los años 1960 y 1970 no eran estructuradas por bloques. En Fortran histórico, todas las variables, incluyendo todos los parámetros de cada procedimiento (llamado subrutina en Fortran) tenían una ubicación fija en memoria. Esto hacía imposible llamar a un procedimiento de forma recursiva, ya fuera directa o indirectamente. Si el procedimiento Fortran P llama Q, Q llama R, R y luego intenta llamar a P, la segunda

[1]Los bloques se pueden delimitar con cualquier símbolo arbitrario, de hecho, los lenguajes de programación modernos ya no usan llaves sino espacios en blanco, que resultan más naturales para el humano. Un buen ejercicio es pensar la gramática que describiría la sintaxis de delimitadores de bloque de Python.

llamada a P no se puede realizar. Si P se llama por segunda vez en esta cadena de llamadas, la segunda llamada escribiría sobre los parámetros y la dirección de la primera llamada, por lo tanto, no se podría volver correctamente a la primera llamada.

5.2. Estructura de bloque

Mitchell 7.1

Los lenguajes con estructrua de bloque se caracterizan por las siguientes propiedades:

- Las nuevas variables se pueden declarar en varios puntos de un programa.
- Cada declaración es visible dentro de una determinada región de texto del programa, llamada bloque. Los bloques pueden ser anidados, pero no pueden superponerse parcialmente. En otras palabras, si dos bloques contienen expresiones o declaraciones en común, entonces un bloque debe estar enteramente contenida dentro del otro.

  Pregunta 1:
  Den un ejemplo de bloques superpuestos parcialmente, y argumenten qué problemas podemos encontrar si permitimos este tipo de estructuras en nuestros programas.

- Cuando un programa inicia la ejecución de las instrucciones contenidas en un bloque en tiempo de ejecución, se asigna memoria a las variables declaradas en ese bloque.
- Cuando un programa sale de un bloque, parte o toda la memoria asignada a las variables declaradas en ese bloque se libera.
- Un identificador de variable que no está declarado en el bloque actual se considera global a ese bloque, y su referencia es a la entidad con el mismo identificador nombre que se encuentra en el bloque más cercano que contiene al bloque actual.

Aunque la mayoría de los lenguajes de programación de propósito general modernos son estructurados por bloques, muchos lenguajes importantes no explotan todas las capacidades que permite la estructura por bloques. Por ejemplo, C y C ++ no permiten declaraciones de función locales dentro de los bloques anidados, y por lo tanto no se ocupan de las cuestiones de aplicación asociados con el retorno de las funciones de los bloques anidados.

En este capítulo, nos fijamos en cómo se manejan en memoria tres clases de variables:

variables locales que se almacenan en la pila de ejecución, en el activation record asociado al bloque.

parámetros de función que también se almacenan en el activation record asociada con el bloque.

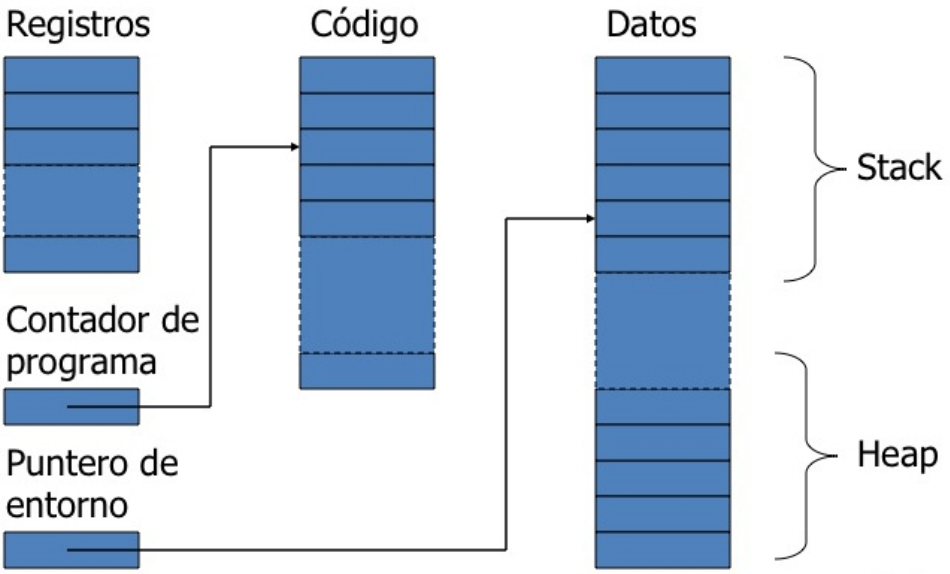

Figura 5.1: Modelo simplificado de la computadora que usamos para describir la semántica operacional.

variables globales que se declaran en algún bloque que contiene al bloque actual y por lo tanto hay que acceder a ellos desde un activation record que se colocó en la pila de ejecución antes del bloque actual.

La mayoría de complicaciones y diferencias entre lenguajes están relacionadas con el acceso a las variables globales, a consecuencia de la estructura de la pila: para dejar más a mano las variables locales, una variable global puede quedar enterrado en la pila bajo un número arbitrario de activation records.

5.3. Activation records

Para describir la semántica de los lenguajes de programación 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 5.1.

Vamos a usar esta simplificación para describir cómo funcionan en memoria los lenguajes estructurados por bloques.

Vemos que este modelo de máquina separa la memoria en la que se guarda el código de la memoria en la que se guardan los datos. 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** y el **puntero de entorno**. El contador de programa 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. Normalmente se incrementa después de ejecutar cada instrucción.

El puntero de entorno nos sirve para saber cuáles son los valores que se asignan a las variables que se están usando en una parte determinada del código. En los lenguajes no estructurados por bloques, la memoria de datos es no estructurada, todo el espacio es heap, y por lo tanto los valores que se asignan a las variables son visibles desde cualquier parte del código. En cambio, en los lenguajes estructurados por bloques, se guardan en el *heap* algunos datos para los que necesitamos persistencia (por ejemplo, variables globales), en los registros guardamos algunos datos para los que queremos rápido acceso, y en la pila de ejecución o stack se guardan los datos estructurados en forma de pila, lo cual hace posible que se instancien variables locales y argumentos distintos para cada llamada de función. Esto posibilita que las funciones puedan ser más abstractas y que se puedan hacer llamadas recursivas.

Pregunta 2:
¿Por qué no se pueden hacer llamadas recursivas sin bloques ni activation records? Explique usando Fortran como ejemplo.

El stack o pila de ejecución funciona de la siguiente forma: cuando el programa entra en un nuevo bloque, se agrega a la pila una estructura de datos que se llama activation record o marco de pila (*stack frame*), 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*.

En la figura 5.2 podemos ver la información que hay en un activation record de arquitectura x86. Vemos que en esta arquitectura hay, además de espacio para variables locales, también espacio para los argumentos que puede recibir una función, que funcionan como variables locales con respecto a la función pero que son escritos por la función que la llama. También encontramos espacio para guardar resultados intermedios, en el caso de que sea necesario. Podemos observar que hay dos direcciones de memoria donde se guardan datos importantes: una, el control link, contiene el que será el puntero de entorno cuando se desapile el activation record actual, es decir, el puntero de entorno del activation record correspondiente a la función que llamaba a la función del activation record actual. La otra 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.

Los activation records pueden tener tamaños variables, por lo tanto, las operaciones que ponen y sacan activation records de la pila guardan también en cada activation record una variable con la dirección de memoria que corresponde a la parte superior del registro de activación anterior. Esta variable o puntero (porque apunta a una dirección de memoria) se llama control link, porque es el enlace que se sigue cuando se devuelve el control a la instrucción en el bloque precedente.

En la siguiente sección vamos a ver cómo funcionan los activation records en ejemplos de ejecución. Para una explicación alternativa, pueden consultar la sección correspondiente de la wikipedia sobre la estructura de pila de llamadas.

5.3.1. Detalle de ejecución de un activation record

Mitchell 7.2

Describiremos el comportamiento de los bloques *in line* primero, ya que estos son más simples que los bloques asociados a las llamadas de función. Un bloque *in line* es un bloque que no es el cuerpo de una función o procedimiento.

Cuando un programa en ejecución entra en un bloque *in line*, se asigna espacio en memoria para las variables que se declaran en el bloque, un conjunto de posiciones de memoria en la pila, el activation record.

Veamos cómo funciona paso a paso, con el siguiente ejemplo de código.

```
1 { int x=0;
  int y=x+1;
3     { int z=(x+y)*(x-y);
      };
5 }
```

Si este código es parte de un programa más grande, la pila puede contener espacio para otras variables antes de ejecutar este bloque.

Este código estará escrito en código máquina, traducido por el compilador, y guardado en la parte de la memoria en la que se guarda el programa. El contador de programa recorrerá cada una de sus instrucciones, y realizará las acciones que indiquen en memoria, tal como sigue.

Cuando se introduce el bloque exterior, se pone en la parte superior de la pila un activation record que contiene espacio para las variables `x` e `y`. A continuación se ejecutan las declaraciones que establecen valores de `x` e `y`, y los valores de `x` e `y` se almacenan en las direcciones de memoria del activation record.

Cuando se introduce el bloque interior, se apila un nuevo activation record que con direcciones en memoria para `z`. Después de establecer el valor de `z`, el activation record que contiene este valor se retira de la pila y el puntero de entorno deja de referirse a este activation record y pasa a referirse al activation record que está ahora en la cabeza de la pila. Por último, cuando se sale del bloque exterior, el activation record que con `x` e `y` se retira de la pila y el puntero de entorno se refiere al activation record que esté en la cabeza de la pila o a ninguno si no hay más activation records en la pila.

Podemos visualizar esta ejecución usando una secuencia de gráficos que describen la secuencia de estados por los que pasa la pila, como se ve en la figura 5.3.

El número de direcciones en memoria que va a necesitar un activation record en tiempo de ejecución depende del número de variables declaradas en el bloque y sus tipos. Debido a que estas cantidades se conocen en tiempo de compilación, el compilador puede determinar

el formato de cada activation record y almacenar esta información como parte del código compilado.

En general, un activation record también puede contener espacio para resultados intermedios, como hemos visto en la figura 5.2. Estos son valores que no reciben un identificador de variable explícito en el código, pero que se guardan temporalmente para facilitar algún cálculo.

Por ejemplo, el registro de activación para este bloque,

```
{ int z = (x + y) * (x-y); }
```

puede tener espacio para asignar valor a la variable `z` pero también para los valores `x+y` y `x-y`, porque los valores de estas subexpresiones pueden tener que ser evaluados y almacenados en algún lugar antes de multiplicarse. Si hacemos zoom en el tercer estado de la pila de la figura 5.3, veremos lo que se muestra en la figura 5.4, con espacio para valores temporales.

En las computadoras modernas, hay suficientes registros en memoria para guardar estos resultados intermedios en los registros y no en la pila. No hablamos de ubicación de registros porque es un tema que compete a los compiladores y no al diseño de lenguajes de programación.

5.4. Ejercicios

5.1. El siguiente es un ejemplo de "*spaghetti code*". Reescríbalo de forma que NO use saltos (GOTO), y en cambio use programación estructurada.

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

5.2. Diagrame los estados de la pila de ejecución en los diferentes momentos de la ejecución del siguiente programa, mostrando cómo se apilan y desapilan los diferentes activation records a medida que se va ejecutando el programa. Puede representar los activation records con la información de variables locales y control links.

```
{int x=2 {int y=3; x=y+2;}}
```

5.3. Diagrame como en el ejercicio anterior.

```
int sum(int n){
    if(n==0)
        return n;
```