b) Un comentario empieza con `/*` e incluye a todos los caracteres hasta la siguiente ocurrencia de la secuencia de caracteres `*/`.

Ejercicio 2.6.2: Extienda el analizador léxico en la sección 2.6.5 para que reconozca los operadores relacionales `<`, `<=`, `==`, `!=`, `>=`, `>`.

Ejercicio 2.6.3: Extienda el analizador léxico en la sección 2.6.5 para que reconozca los valores de punto flotante como `2.`, `3.14` y `.5`.

2.7 Tablas de símbolos

Las *tablas de símbolos* son estructuras de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente. La información se recolecta en forma incremental mediante las fases de análisis de un compilador, y las fases de síntesis la utilizan para generar el código destino. Las entradas en la tabla de símbolos contienen información acerca de un identificador, como su cadena de caracteres (o lexema), su tipo, su posición en el espacio de almacenamiento, y cualquier otra información relevante. Por lo general, las tablas de símbolos necesitan soportar varias declaraciones del mismo identificador dentro de un programa.

En la sección 1.6.1 vimos que el alcance de una declaración es la parte de un programa a la cual se aplica esa declaración. Vamos a implementar los alcances mediante el establecimiento de una tabla de símbolos separada para cada alcance. Un bloque de programa con declaraciones[8] tendrá su propia tabla de símbolos, con una entrada para cada declaración en el bloque. Este método también funciona para otras construcciones que establecen alcances; por ejemplo, una clase tendría su propia tabla, con una entrada para cada campo y cada método.

Esta sección contiene un módulo de tabla de símbolos, adecuado para usarlo con los fragmentos del traductor en Java de este capítulo. El módulo se utilizará como está, cuando ensamblemos el traductor completo en el apéndice A. Mientras tanto, por cuestión de simplicidad, el ejemplo principal de esta sección es un lenguaje simplificado, que sólo contiene las construcciones clave que tocan las tablas de símbolos; en especial, los bloques, las declaraciones y los factores. Omitiremos todas las demás construcciones de instrucciones y expresiones, para poder enfocarnos en las operaciones con la tabla de símbolos. Un programa consiste en bloques con declaraciones opcionales e "instrucciones" que consisten en identificadores individuales. Cada instrucción de este tipo representa un uso del identificador. He aquí un programa de ejemplo en este lenguaje:

```
{ int x; char y; { bool y; x; y; } x; y; }
```
(2.7)

Los ejemplos de la estructura de bloques en la sección 1.6.3 manejaron las definiciones y usos de nombres; la entrada (2.7) consiste únicamente de definiciones y usos de nombres.

La tarea que vamos a realizar es imprimir un programa revisado, en el cual se han eliminado las declaraciones y cada "instrucción" tiene su identificador, seguido de un signo de punto y coma y de su tipo.

[8] En C, por ejemplo, los bloques de programas son funciones o secciones de funciones que se separan mediante llaves, y que tienen una o más declaraciones en su interior.

¿Quién crea las entradas en la tabla de símbolos?

El analizador léxico, el analizador sintáctico y el analizador semántico son los que crean y utilizan las entradas en la tabla de símbolos durante la fase de análisis. En este capítulo, haremos que el analizador sintáctico cree las entradas. Con su conocimiento de la estructura sintáctica de un programa, por lo general, un analizador sintáctico está en una mejor posición que el analizador léxico para diferenciar entre las distintas declaraciones de un identificador.

En algunos casos, un analizador léxico puede crear una entrada en la tabla de símbolos, tan pronto como ve los caracteres que conforman un lexema. Más a menudo, el analizador léxico sólo puede devolver un token al analizador sintáctico, por decir **id**, junto con un apuntador al lexema. Sin embargo, sólo el analizador sintáctico puede decidir si debe utilizar una entrada en la tabla de símbolos que se haya creado antes, o si debe crear una entrada nueva para el identificador.

Ejemplo 2.14: En la entrada anterior (2.7), el objetivo es producir lo siguiente:

```
{  {  x:int; y:bool; } x:int; y:char; }
```

Las primeras `x` y `y` son del bloque interno de la entrada (2.7). Como este uso de `x` se refiere a la declaración de `x` en el bloque externo, va seguido de `int`, el tipo de esa declaración. El uso de `y` en el bloque interno se refiere a la declaración de `y` en ese mismo bloque y, por lo tanto, tiene el tipo booleano. También vemos los usos de `x` y `y` en el bloque externo, con sus tipos, según los proporcionan las declaraciones del bloque externo: entero y carácter, respectivamente. □

2.7.1 Tabla de símbolos por alcance

El término "alcance del identificador x" en realidad se refiere al alcance de una declaración específica de x. El término *alcance* por sí solo se refiere a una parte del programa que es el alcance de una o más declaraciones.

Los alcances son importantes, ya que el mismo identificador puede declararse para distintos fines en distintas partes de un programa. A menudo, los nombres comunes como `i` y `x` tienen varios usos. Como otro ejemplo, las subclases pueden volver a declarar el nombre de un método para redefinirlo de una superclase.

Si los bloques pueden anidarse, varias declaraciones del mismo identificador pueden aparecer dentro de un solo bloque. La siguiente sintaxis produce bloques anidados cuando *instrs* puede generar un bloque:

$$bloque \rightarrow \ '\{'\ decls\ instrs\ '\}'$$

Colocamos las llaves entre comillas simples en la sintaxis para diferenciarlas de las llaves para las acciones semánticas. Con la gramática en la figura 2.38, *decls* genera una secuencia opcional de declaraciones e *instrs* genera una secuencia opcional de instrucciones.

Optimización de las tablas de símbolos para los bloques

Las implementaciones de las tablas de símbolos para los bloques pueden aprovechar la regla del bloque anidado más cercano. El anidamiento asegura que la cadena de tablas de símbolos aplicables forme una pila. En la parte superior de la pila se encuentra la tabla para el bloque actual. Debajo de ella en la pila están las tablas para los bloques circundantes. Por ende, las tablas de símbolos pueden asignarse y desasignarse en forma parecida a una pila.

Algunos compiladores mantienen una sola hash table de entradas accesibles; es decir, de entradas que no se ocultan mediante una declaración en un bloque anidado. Dicha hash table soporta búsquedas esenciales en tiempos constantes, a expensas de insertar y eliminar entradas al entrar y salir de los bloques. Al salir de un bloque B, el compilador debe deshacer cualquier modificación a la hash table debido a las declaraciones en el bloque B. Para ello puede utilizar una pila auxiliar, para llevar el rastro de las modificaciones a la tabla hash mientras se procesa el bloque B.

Inclusive, una instrucción puede ser un bloque, por lo que nuestro lenguaje permite bloques anidados, en donde puede volver a declararse un identificador.

La regla del *bloque anidado más cercano* nos indica que un identificador x se encuentra en el alcance de la declaración anidada más cercana de x; es decir, la declaración de x que se encuentra al examinar los bloques desde adentro hacia fuera, empezando con el bloque en el que aparece x.

Ejemplo 2.15: El siguiente seudocódigo utiliza subíndices para diferenciar entre las distintas declaraciones del mismo identificador:

```
1)  {    int x1; int y1;
2)       {  int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)         ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

El subíndice no forma parte de un identificador; es, de hecho, el número de línea de la declaración que se aplica al identificador. Por ende, todas las ocurrencias de x están dentro del alcance de la declaración en la línea 1. La ocurrencia de y en la línea 3 está en el alcance de la declaración de y en la línea 2, ya que y se volvió a declarar dentro del bloque interno. Sin embargo, la ocurrencia de y en la línea 5 está dentro del alcance de la declaración de y en la línea 1.

La ocurrencia de w en la línea 5 se encuentra supuestamente dentro del alcance de una declaración de w fuera de este fragmento del programa; su subíndice 0 denota una declaración que es global o externa para este bloque.

Por último, z se declara y se utiliza dentro del bloque anidado, pero no puede usarse en la línea 5, ya que la declaración anidada se aplica sólo al bloque anidado. □

La regla del bloque anidado más cercano puede implementarse mediante el encadenamiento de las tablas de símbolos. Es decir, la tabla para un bloque anidado apunta a la tabla para el bloque circundante.

Ejemplo 2.16: La figura 2.36 muestra tablas de símbolos para el seudocódigo del ejemplo 2.15. B_1 es para el bloque que empieza en la línea 1 y B_2 es para el bloque que empieza en la línea 2. En la parte superior de la figura hay una tabla de símbolos adicional B_0 para cualquier declaración global o predeterminada que proporcione el lenguaje. Durante el tiempo que analizamos las líneas 2 a 4, el entorno se representa mediante una referencia a la tabla de símbolos inferior (la de B_2). Cuando avanzamos a la línea 5, la tabla de símbolos para B_2 se vuelve inaccesible, y el entorno se refiere en su lugar a la tabla de símbolos para B_1, a partir de la cual podemos llegar a la tabla de símbolos global, pero no a la tabla para B_2. □

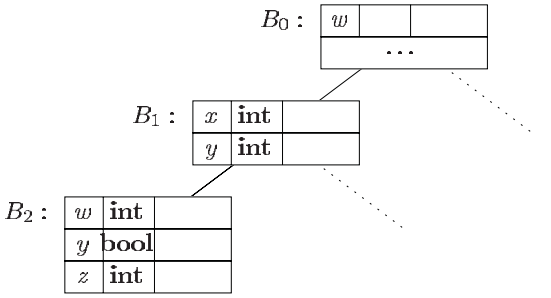

Figura 2.36: Tablas de símbolos encadenadas para el ejemplo 2.15

La implementación en Java de las tablas de símbolos encadenadas en la figura 2.37 define a una clase `Ent`, abreviación de *entorno*.[9] La clase `Ent` soporta tres operaciones:

- *Crear una nueva tabla de símbolos.* El constructor `Ent(p)` en las líneas 6 a 8 de la figura 2.37 crea un objeto `Ent` con una hash table llamada `tabla`. El objeto se encadena al parámetro con valor de entorno `p`, estableciendo el campo `siguiente` a `p`. Aunque los objetos `Ent` son los que forman una cadena, es conveniente hablar de las tablas que se van a encadenar.
- *put* una nueva entrada en la tabla actual. La hash table contiene pares clave-valor, en donde:
  - La *clave* es una cadena, o más bien una referencia a una cadena. Podríamos usar de manera alternativa referencias a objetos token para identificadores como las claves.
  - El *valor* es una entrada de la clase `Simbolo`. El código en las líneas 9 a 11 no necesita conocer la estructura de una entrada; es decir, el código es independiente de los campos y los métodos en la clase `Simbolo`.

[9] "Entorno" es otro término para la colección de tablas de símbolos que son relevantes en un punto dado en el programa.

```
1) package símbolos;                    // Archivo Ent.java
2) import java.util.*;
3) public class Ent {
4)     private Hashtable tabla;
5)     protected Ent ant;

6)     public Ent(Ent p) {
7)        tabla = new Hashtable(); ant = p;
8)     }

9)     public void put(String s, Simbolo sim) {
10)       tabla.put(s, sim);
11)    }

12)    public Simbolo get(String s) {
13)       for( Ent e = this; e != null; e = e.ant ) {
14)          Simbolo encontro = (Simbolo)(e.tabla.get(s));
15)          if( encontro != null ) return encontro;
16)       }
17)       return null;
18)    }
19) }
```

Figura 2.37: La clase *Ent* implementa a las tablas de símbolos encadenadas

- *get* una entrada para un identificador buscando en la cadena de tablas, empezando con la tabla para el bloque actual. El código para esta operación en las líneas 12 a 18 devuelve una entrada en la tabla de símbolos o `null`.

El encadenamiento de las tablas de símbolos produce una estructura tipo árbol, ya que puede anidarse más de un bloque dentro de un bloque circundante. Las líneas punteadas en la figura 2.36 son un recordatorio de que las tablas de símbolos encadenadas pueden formar un árbol.

2.7.2 El uso de las tablas de símbolos

En efecto, la función de una tabla de símbolos es pasar información de las declaraciones a los usos. Una acción semántica "coloca" (put) información acerca de un identificador x en la tabla de símbolos, cuando se analiza la declaración de x. Posteriormente, una acción semántica asociada con una producción como $factor \rightarrow \textbf{id}$ "obtiene" (get) información acerca del identificador, de la tabla de símbolos. Como la traducción de una expresión E_1 **op** E_2, para un operador **op** ordinario, depende sólo de las traducciones de E_1 y E_2, y no directamente de la tabla de símbolos, podemos agregar cualquier número de operadores sin necesidad de cambiar el flujo básico de información de las declaraciones a los usos, a través de la tabla de símbolos.

Ejemplo 2.17: El esquema de traducción en la figura 2.38 ilustra cómo puede usarse la clase *Ent*. El esquema de traducción se concentra en los alcances, las declaraciones y los usos. Implementa la traducción descrita en el ejemplo 2.14. Como dijimos antes, en la entrada

programa	→		{ *sup* = **null**; }
		bloque	
bloque	→	'{'	{ *guardado* = *sup;* *sup* = **new** *Ent*(*sup*); print("{ "); }
		decls instrs '}'	{ *sup* = *guardado*; print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**tipo id ;**	{ *s* = **new** *Símbolo*; *s.type* = **tipo**.*lexema* *sup.put*(**id**.*lexema*, *s*); }
instrs	→	*instrs instr*	
	|	ϵ	
instr	→	*bloque*	
	|	*factor* ;	{ print("; "); }

factor	→	**id**	{ *s* = *sup.get*(**id**.*lexema*);
			print(**id**.*lexema*);
			print(":"); }
			print(*s.tipo*);

Figura 2.38: El uso de las tablas de símbolos para traducir un lenguaje con bloques

```
{ int x;  char y;  { bool y; x; y; } x; y; }
```

el esquema de traducción elimina las declaraciones y produce

```
{  {  x:int;  y:bool;  }  x:int;  y:char; }
```

Observe que los cuerpos de las producciones se alinearon en la figura 2.38, para que todos los símbolos de gramática aparezcan en una columna y todas las acciones en una segunda columna. Como resultado, por lo general, los componentes del cuerpo se esparcen a través de varias líneas.

Ahora, consideremos las acciones semánticas. El esquema de traducción crea y descarta las tablas de símbolos al momento de entrar y salir de los bloques, respectivamente. La variable *sup* denota la tabla superior, en el encabezado de una cadena de tablas. La primera producción de la gramática subyacente es *programa* → *bloque*. La acción semántica antes de *bloque* inicializa *sup* a **null**, sin entradas.

La segunda producción, *bloque* → *'{' decls instrs '}'*, tiene acciones al momento en que se entra y se sale del bloque. Al entrar al bloque, antes de *decls*, una acción semántica guarda una referencia a la tabla actual, usando una variable local llamada *guardado*. Cada uso de esta producción tiene su propia variable local *guardado*, distinta de la variable local para cualquier otro uso de esta producción. En un analizador sintáctico de descenso recursivo, *guardado* sería local para el *bloque* for del procedimiento. En la sección 7.2 veremos el tratamiento de las variables locales de una función recursiva. El siguiente código:

$$sup = \textbf{new}\ Ent(sup);$$

establece la variable *sup* a una tabla recién creada que está encadenada al valor anterior de *sup*, justo antes de entrar al bloque. La variable *sup* es un objeto de la clase *Ent*; el código para el constructor *Ent* aparece en la figura 2.37.

Al salir del bloque, después de *'}'*, una acción semántica restaura *sup* al valor que tenía guardado al momento de entrar al bloque. En realidad, las tablas forman una pila; al restaurar *sup* a su valor guardado, se saca el efecto de las declaraciones en el bloque.[10] Por ende, las declaraciones en el bloque no son visibles fuera del mismo.

Una declaración, *decls* → **tipo id** produce una nueva entrada para el identificador declarado. Asumimos que los tokens **tipo** e **id** tienen cada uno un atributo asociado, que es el tipo y el lexema, respectivamente, del identificador declarado. En vez de pasar por todos los campos de un objeto de símbolo *s*, asumiremos que hay un campo *tipo* que proporciona el tipo del símbolo. Creamos un nuevo objeto de símbolo *s* y asignamos su tipo de manera apropiada, mediante *s.tipo* = **tipo**.*lexema*. La entrada completa se coloca en la tabla de símbolos superior mediante *sup.put*(**id**.*lexema*, *s*).

La acción semántica en la producción *factor* → **id** utiliza la tabla de símbolos para obtener la entrada para el identificador. La operación *get* busca la primera entrada en la cadena de tablas, empezando con *sup*. La entrada que se obtiene contiene toda la información necesaria acerca del identificador, como su tipo. □

2.8 Generación de código intermedio

El front-end de un compilador construye una representación intermedia del programa fuente, a partir de la cual el back-end genera el programa destino. En esta sección consideraremos representaciones intermedias para expresiones e instrucciones, y veremos ejemplos de cómo producir dichas representaciones.

2.8.1 Dos tipos de representaciones intermedias

Como sugerimos en la sección 2.1 y especialmente en la figura 2.4, las dos representaciones intermedias más importantes son:

[10] En vez de guardar y restaurar tablas en forma explícita, una alternativa podría ser agregar las operaciones estáticas *push* y *pop* a la clase `Ent`.