Compilador de HULK

Daniel Abad Fundora C411
Anabel Benítez González C411
Enzo Rojas D'Toste C411

8 de abril de 2024

Índice

1. Uso del compilador **2**

2. Introducción **2**

3. Generador de lexer **2**
3.1. Grámatica de expresiones regulares usada 3

4. Generador de parser **3**

5. Análisis Léxico **4**

6. Análisis Sintáctico **4**
6.1. Palabras reservadas 4
6.2. Punto y coma 4
6.3. Precedencia y asociatividad 5
6.4. Anotación de tipos 5

7. Análisis Semántico **6**
7.1. Recolector de tipos 6
7.2. Constructor de tipos 7
7.3. Tipos especiales 7
7.3.1. VectorType 7
7.3.2. AutoType 8
7.3.3. ErrorType 8
7.3.4. SelfType 8
7.4. Chequeo de tipos 8
7.4.1. Conformidad de tipos y protocolos 9
7.4.2. Recolector de variables 9
7.4.3. Inferidor de tipos 9
7.4.4. Verificador de tipos 11

8. Generación de código **12**

1. Uso del compilador

Puede encontrarlo en este repositorio de github. Para ejecutar el proyecto necesita tener instalada la versión 3.10 de python o superior. Para instalar los paquetes necesarios ejecute el comando:

```
pip install -r requirements.txt
```

Para compilar y ejecutar un fichero con extensión '.hulk' se debe ejecutar el siguiente comando:

```
python3 src/main.py <archivo.hulk>
```

El fichero de C generado por el compilador tendrá el mismo nombre que el fichero de entrada y estará ubicado en la misma carpeta.

2. Introducción

En este informe se presenta una descripción deL proceso de desarrollo y funcionamiento de un compilador para el lenguaje HULK. El objetivo principal de este compilador es convertir programas escritos en dicho lenguaje en código ejecutable.

El proceso de compilación de un programa de HULK en nuestro compilador se desarrolla en 4 partes:

1. Análisis Léxico: Convierte el código fuente en una secuencia de tokens.
2. Análisis Sintáctico: Analiza si esta secuencia de tokens tiene sentido sintácticamente, el devuelve el árbol de sintaxis abstracta que representa al programa.
3. Análisis Semántico: Realiza varios recorridos sobre el AST para verificar que se cumplan las reglas semánticas del lenguaje.
4. Generación de código: Recorre el AST de HULK generando el código equivalente en C.

Puede encontrar más información sobre el lenguaje HULK siguiendo este enlace.

3. Generador de lexer

El lexer se construye a partir de la tabla de expresiones regulares (una lista de tuplas con la forma: `(<token_type>, <regex_str>)`). La prioridad/relevancia de cada tipo de token está marcada por el índice que ocupa en la tabla. Los tipos de tokens cuyas expresiones regulares estén registradas más cerca del inicio de la tabla tiene más prioridad.

Las principales características del lexer generado son:

- Resulta de transformar el autómata unión entre todas las expresiones regulares del lenguaje y convertirlo en determinista.
- Cada estado final almacena los tipos de tokens que se reconocen al alcanzarlo. Se establece una prioridad entre ellos para poder desambiguar.
- Para tokenizar, la cadena de entrada viaja repetidas veces por el autómata.

- Cada vez, se comienza por el estado inicial, pero se continúa a partir de la última sección de la cadena que fue reconocida.
- Cuando el autómata se "traba"durante el reconocimiento de una cadena, se reporta la ocurrencia de un token. Su lexema está formado por la concatenación de los símbolos que fueron consumidos desde el inicio y hasta pasar por el último estado final antes de trabarse. Su tipo de token es el de mayor relevancia entre los anotados en el estado final.
- Al finalizar de consumir toda la cadena, se reporta el token de fin de cadena.

Para el manejo de errores, hemos definido el token `UnknownToken`, el cual se reporta cuando ninguno de los tokens disponibles es reconocido. Además, para evitar que el proceso se detenga cuando se encuentra este token, hemos definido una tabla llamada `synchronizing_tokens`, la cual contiene los tokens que el lexer busca para recuperarse de un error y continuar el análisis.

Cada token guarda la fila y la columna en la que fue encontrado en el código fuente. Esta información es útil para la detección y reporte de errores durante la compilación. Cuando se produce un error, la inclusión de la fila y la columna del token permite identificar rápidamente la ubicación exacta donde ocurrió el problema, lo que facilita su corrección por parte del programador.

3.1. Grámatica de expresiones regulares usada

No Terminales: `<origin>`, `<disjunction>`, `<concat>`, `<kleene>`, `<factor>`

Terminales: `*`, `|`, `(`, `)`, `char`

Símbolo distinguido: `<origin>`

Producciones:

```
<origin> -> <disjunction>
<disjunction> -> <disjunction> | <concat>
<disjunction> -> <concat>
<concat> -> <concat> <kleene>
<concat> -> <kleene>
<kleene> -> <kleene> *
<kleene> -> <factor>
<factor> -> ( <disjunction> )
<factor> -> char
```

4. Generador de parser

Implementamos un generador de parsers LR(1), pues este parser es capaz de reconocer un amplio espectro de gramáticas libres de contexto.

Cuando se produce un error durante el proceso de parsing al intentar analizar una cadena de entrada, es importante informar al usuario sobre el error ocurrido, definimos un ParserError que contiene el índice del token que no fue reconocido, para que el programador que user nuestro generador pueda reportar un mensaje de error con más información.

5. Análisis Léxico

El análisis léxico marca el inicio del proceso de compilación al escanear el código fuente y descomponerlo en unidades léxicas más pequeñas, conocidas como tokens. En nuestro lexer para HULK, estos tokens abarcan desde palabras reservadas y operadores hasta literales (booleanos, cadenas y números) e identificadores.

Adicionalmente, hemos incorporado tokens especiales como spaces (espacios en blanco), escaped char (caracteres de escape) y unterminated string (cadenas no terminadas), este último con el propósito de reportar errores más específicos. Cabe destacar que estos tokens especiales se eliminan antes de avanzar a la siguiente fase, ya que no poseen valor sintáctico.

6. Análisis Sintáctico

La fase de análisis sintáctico desempeña un papel fundamental al validar la estructura gramatical del código fuente. Este análisis se encarga de interpretar la secuencia de tokens generada por el análisis léxico y verificar si cumple con las reglas de la gramática.

Para nuestro compilador para el lenguaje HULK, hemos optado por utilizar un parser LR(1) para llevar a cabo el análisis sintáctico.

En las subsecciones siguientes abordaremos las principales decisiones tomadas a la hora de diseñar la grámatica.

6.1. Palabras reservadas

Las palabras reservadas son términos específicos dentro del lenguaje que tienen un significado predefinido. Estas no pueden ser utilizadas como nombres de variables, funciones o tipos personalizados dentro del código, pues constituiría un error en el parser.

A continuación se listan las palabras reservadas de HULK:

let, in Palabras clave para la definición de variables y alcance.

if, else, elif Estructuras de control condicionales.

function Palabra clave para definir funciones.

while, for Estructuras de control de bucles.

new, is, as

protocol, extends Palabras clave relacionadas con protocolos y su herencia.

type, inherits, base Palabras clave relacionadas con tipos y su herencia.

true, false Valores booleanos.

6.2. Punto y coma

Evitamos incluir ; al final de las producciones correspondientes a las expresiones para que no ocurra este tipo de comportamientos:

```
if (let a = True in a;) ...
```

Para manejar los ; usamos la producción:

```
eol_expr %= expr + semicolon
eol_expr %= expr_block
```

Donde expr es tanto una expresión simple como un bloque, así aseguramos que cuando es un bloque es opcional usar el ;.

6.3. Precedencia y asociatividad

HULK es un lenguaje de programación centrado en expresiones, donde la mayoría de las construcciones sintácticas, como funciones, bucles y bloques de código, son expresiones. Los programas en HULK siempre terminan con una única expresión global que sirve como punto de entrada del programa. A continuación una tabla que muestra la precedencia y asociatividad de los operadores y funciones.

Categoría	Expresiones y Operadores	Precedencia	Asociatividad

	if(<expr>) <expr>else <expr>
	let x = <expr>in <expr>
	while (<expr>) <expr>
	for (x in <expr>) <expr>
		1	-

Asignación	:=	2	Derecha
Or	|	3	Izquierda
And	&	4	Izquierda
Igualdad	==, !=	5	Izquierda
Comparación	>=, <=, >, <	6	Izquierda
Type-testing	is, as	7	Izquierda
Concatención	@, @@	8	Izquierda
Aditivo	+, -	9	Izquierda
Multiplicativo	*, /, %	10	Izquierda
Signo	+x, -x	11	-
Potencia	**, ^	12	Derecha
Instanciación	new A(<expr>,..., <expr>)	13	-
Not	!	14	-
Acceso a miembro	x.y, x.f(<expr>,..., <expr>)	15	Izquierda
Expresión parentizada	(<expr>)	16	-
Átomo	<num_literal>, <str_literal>, <identifier>, <bool_literal>, f(<expr>,..., <expr>), [<expr>|| x in <exp>], [<expr>,..., <exp>])	17	-

Cuadro 1: Tabla de precedencia y asociatividad

6.4. Anotación de tipos

En HULK, las anotaciones de tipo pueden agregarse a variables, argumentos de funciones, atributos de tipos y argumentos de tipos usando : <type>, lo que permite una verificación estática durante la compilación para garantizar la consistencia y compatibilidad de los tipos utilizados en el código.

Para esto creamos las siguientes producciones:

```
optional_type_annotation %= G.Epsilon
optional_type_annotation %= colon + type_annotation

type_annotation %= idx
type_annotation %= type_annotation + o_square_bracket + c_square_bracket
```

Como su nombre indica optional_type_annotation es usada en todos los lugares mencionados anteriormente para permitir al programador especificar o no el tipo.

Por su parte type_annotation permite especificar tipos simples como `String` y vectores como `Number[]` o `Boolean[][]`.

En el caso de los protocolos, la firma de los métodos siempre debe especificar los tipos, pues no tiene cuerpo donde puedan inferirse, esto es resuelto en la gramática pues en el cuerpo de la producción se espera que tanto parámetros como retorno tenga la anotación de tipo.

7. Análisis Semántico

El análisis semántico en el proceso de compilación es una fase crítica que se encarga de verificar las reglas y restricciones del lenguaje, asegurando la coherencia, consistencia y el significado del código. Esta fase examina la semántica del programa en relación con su estructura sintáctica, identificando posibles errores como tipos incorrectos, variables no declaradas o mal uso de operadores.

Para esta fase se recorre varias veces el árbol de sintaxis abstracta(AST), utilizando el patrón visitor visto en clase. A continuación explicamos el objetivo y algunos detalles de cada recorrido.

Son importantes en esta fase el Context y el Scope.

Context: Conjunto de datos y entidades relevantes para la ejecución del programa a nivel global.

- Tipos de datos disponibles.
- Protocolos definidos.
- Funciones globales disponibles.

Scope: Se refiere a la región del código donde una variable es válida y accesible. Por ejemplo, una variable declarada dentro de una función solo es accesible dentro de esa función y no fuera de ella. El Scope ayuda a evitar conflictos de nombres y a mantener la integridad del programa al limitar la visibilidad de las variables a partes específicas del código.

7.1. Recolector de tipos

Este recorrido tiene como objetivo crear el Context y recolectar en él todos los tipos y protocolos definidos en nuestro programa, por tanto solo se visita los nodos TypeDeclarationNode y ProtocolDeclarationNode, y por supuesto ProgramNode. En este recorrido también se añaden al contexto los tipos, protocolos y funciones built-in de HULK.

Types: Boolean, Number, String, Object, Range

Protocols: Iterable

Functions: print(x: Object): String,
sqrt(x: Number): Number,
sin(x: Number): Number,
cos(x: Number): Number,
exp(x: Number): Number,
log(x: Number): Number,
rand(): Number,
range(min: Number, max: Number): Range

El tipo `Vector` no está incluido en el cotexto, y por defecto cada tipo, tanto built-in como creado por el programador tiene un tipo Vector asociado, más adelante explicaremos cómo manejamos este tipo especial.

7.2. Constructor de tipos

Una vez que ya tenemos todos los tipos definidos en nuestro programa, pasamos a su construcción, con esto nos referimos a:

- Definir su padre: Si se especifica, asignamos el tipo especificado como el padre del tipo en cuestión. En caso de existir un ciclo en la jerarquía de tipos, asignamos el tipo especial `ErrorType` como padre temporalmente, del cual hablaremos más adelante. Si no se especifica un padre, asignamos el tipo `Object` como padre por defecto a los tipos.
- Definir los atributos: Para cada tipo, definimos los atributos que le corresponden. Estos pueden incluir el tipo (venían anotados) o no.
- Definir la firma de los métodos: Establecemos los métodos asociados a cada tipo y protocolo. Esto implica definir las firmas de los métodos, incluyendo sus nombres, parámetros y tipos de parámetros y de retorno si se especificaban.
- Definir la firma de las funciones globales: Esto incluye especificar los nombres de las funciones, sus parámetros y tipos de prámetros y de retorno si se especificaban.

En esta etapa le asignamos tipos a todos los métodos, atributos o funciones , cuando nos encontramos con uno sin anotación de tipo, utilizamos un tipo especial `AutoType` para asignar un tipo de forma automática. Este tipo especial, AutoType, actúa como un marcador temporal que permite al compilador continuar el proceso de análisis semántico, incluso cuando no se han proporcionado todas las anotaciones de tipo necesarias. En etapas posteriores del proceso, el compilador puede deducir o inferir tipos para estos basándose en su contexto y uso en el programa.

7.3. Tipos especiales

7.3.1. VectorType

El tipo Vector representa una secuencia ordenada de elementos del mismo tipo. Para manejar que cada tipo existente en el contexto tenga un tipo vector asociado, no introducimos estos `X[]` (Vector de X) en el contexto, sino que creamos (a demanda) un objeto de tipo VectorType que lo representa. Esta clase tiene un método para obtener el tipo de los

elementos del vector. También redefine la relación conformodidad, pues si other es de tipo vector retorna el conforms_to de sus elementos, en otro caso llama al confors_to de Type.

Además, define métodos estándar como `size` para obtener el tamaño del vector, y `next` y `current` pues implementa el protocolo Iterable.

Como todos los tipos en HULK, el tipo `X[]` hereda de Object.

7.3.2. AutoType

Este tipo se emplea para representar tipos que deben ser inferidos por el sistema de inferencia de tipos en lugar de ser declarados explícitamente por el usuario en el código fuente.

7.3.3. ErrorType

El tipo ErrorType es utilizado para representar errores en el sistema de tipos durante el proceso de análisis semántico. Su propósito principal es indicar que ha ocurrido un error durante el análisis semántico.

Una de las características clave de ErrorType es que siempre conforma a cualquier otro tipo. Esto significa que en cualquier situación donde se espera un tipo, ErrorType puede ser utilizado sin causar conflictos en la verificación de tipos. Del mismo modo, todo tipo en el sistema de tipos también conforma a ErrorType. Esto permite que ErrorType actúe como una especie de comodín que puede aceptar cualquier tipo de valor, lo que resulta útil para evitar la propagación de errores y permitir que el análisis del código continúe, incluso cuando se han encontrado problemas.

7.3.4. SelfType

Cuando se necesita hacer referencia al tipo en el que se está definiendo un atributo o método, se utiliza el símbolo 'self'. A este símbolo se le asigna el tipo SelfType.

Este tipo se utiliza para referirse al tipo del propio objeto dentro de la definición de un tipo. SelfType contiene una referencia al tipo real al que se refiere, permitiendo así el acceso a los atributos y métodos del tipo referido.

7.4. Chequeo de tipos

Luego de tener construido todos los tipos y funciones (sus firmas), nos quedaría verificar la correctitud de sus cuerpos, y la consistencia de los tipos. Esto implica analizar y evaluar la validez de las expresiones, y asegurarse de que cumpla con los requisitos y restricciones del lenguaje. Para garantizar la consistencia de los tipos utilizados en estas implementaciones, es necesario conocer el tipo estático de cada expresión presente en el código.

Aunque las anotaciones de tipo son opcionales, HULK infiere tipos para la mayoría de los símbolos. El proceso de inferencia de tipos se lleva a cabo antes del proceso de verificación de tipos. El inferidor de tipos asigna anotaciones de tipo a todos los símbolos y expresiones que no están explícitamente anotados. Luego, el verificador de tipos verifica que todas las reglas semánticas sean válidas.

Para realizar este proceso implementamos 3 recorredores del AST: el recolector de variables, el inferidor de tipos y el verificador de tipos.

7.4.1. Conformidad de tipos y protocolos

En HULK, la relación básica entre tipos se llama conformidad ($\leq$). $T1 \leq T2$ (T1 se conforma a T2) si una variable de tipo T2 puede contener un valor de tipo T1 de tal manera que todas las operaciones semánticamente válidas con T2 también lo sean con T1.

A continuación listamos las reglas para la conformidad en HULK:

- Todo tipo se conforma a Object.
- Todo tipo se conforma a sí mismo.
- Si T1 hereda de T2, entonces T1 se conforma a T2.
- Si T1 se conforma a T2 y T2 se conforma a T3, entonces T1 se conforma a T3.
- Los únicos tipos que se conforman a Number, String y Boolean, respectivamente, son esos mismos tipos.

Extendemos esta definición a los protocolos de la siguiente manera:

- Si P1 extiende a P2, entonces P1 se conforma a P2.
- Si P1 se conforma a P2 y P2 se conforma a P3, entonces P1 se conforma a P3.
- Un protocolo nunca se conforma a un tipo.
- Un protocolo P1 se conforma a otro P2 si todos los métodos definidos en P2 pueden ser sustituidos por métodos definidos en P2. Esto quiere decir que los parámetros sean covariantes, y los valores de retorno contravariantes. Esto significa que los argumentos pueden ser del mismo tipo o más genéricos, y los valores de retorno pueden ser del mismo tipo o más específicos que los definidos en el P2. Por supuesto ambos métodos deben tener igual nombre y cantidad de parámetros.
- Un tipo T se conforma a un protocolo P si todos los métodos definidos en P pueden ser sustituidos con métodos definidos en T.

7.4.2. Recolector de variables

Este recorrido se encarga de construir los scopes y asignarle a cada nodo del AST el scope que le corresponde. Además define en estos scopes sus variables. Cada variable cuenta con una propiedad `is_parameter`, pues aunque por ejemplo dentro de un método los parámetros son tratados como variables, nos vimos en la necesidad de hacer esta distinción para el reporte de errores.

En el scope que corresponde a los métodos se define la variable especial `self` de tipo `SelfType` que envuelve al tipo actual, pero se hizo necesario diferenciarlo para el reporte de errores.

7.4.3. Inferidor de tipos

El objetivo de este recorrido es inferir el tipo de todas las variables, atributos, parámetros y retorno de métodos/funciones definidad en nuestro programa. A continuación se describe cómo se infiere el tipo de cada expresión:

Literales: Su tipo proviene directamente del parser.

Operaciones aritméticas: Tanto en las binarias (`+`, `-`, `*`, `/`, `%`, `^`, `**`) como en las unarias (`-x`) se infiere que el tipo de los operandos es Number y el tipo de retorno es Number.

Operaciones boolenas: Tanto en las binarias (`|`, `&`) como en las unarias (`!`) se infiere que el tipo de los operandos es Boolean y el tipo de retorno es Boolean.

Operaciones de comparación: (`>=`, `<=`, `>`, `<`) se infiere que el tipo de los operandos es Number y el tipo de retorno es Boolean.

Operaciones de igualdad: (`==`, `!=`) se infiere que el tipo de retorno es Boolean. Si tenemos $x == y$, donde sabemos que tipo de x es A y tipo de y es B, solo devolveremos Boolean si $A \leq B$ o $B \leq A$, en caso contrario inferiremos tipo ErrorType, aunque aún no lo reportaremos.

Concatenación: (`@`, `@@`) se infiere que el tipo de los operandos es Object y el tipo de retorno es String.

Is: Su tipo de retorno es Boolean.

As: Su tipo de retorno es el tipo especificado a la derecha.

Instanciación de tipo: Su tipo de retorno es el tipo instanciado.

Indexación: (`x[y]`) Se infiere que y es un número.

Llamados a funciones y métodos: Se infiere el tipo de los argumentos como el tipo del parámetro correspondiente y el tipo de retorno es el correspondiente al método o función.

Variables en expresiones let-in: Cuando una variable no está anotada con tipo, el inferidor de tipos le asigna el tipo inferido para su expresión de inicialización.

Atributos: En una declaración de atributo que no está anotada con tipo, el inferidor de tipos asigna un tipo que sea equivalente al tipo inferido para su expresión de inicialización.

Parámetros: En una función, método o tipo, cuando un parámetro no está anotado con tipo, el inferidor de tipos asigna el tipo más específico que sea consistente con el uso de ese parámetro en el cuerpo. Si se infiere más de un tipo en diferentes ramas de la jerarquía se reporta un error de uso inconsistente del parámetro.

Condicional if-elif-else: Esta expresión tiene la particularidad de contar con más de una rama, por tanto se infiere que su tipo es el ancestro común más cercano (LCA) de los tipos de cada rama, o en última instancia Object, en el caso de Iterable y Number por ejemplo.

Tipo de retorno: En una función o método, cuando el tipo de retorno no está anotado, el inferidor de tipos asigna el tipo del cuerpo de la función o método.

Inicialización explícita de vector: (`[<expr>,..., <expr>]`) Su tipo de retorno es un vector del LCA de todos los elementos.

Inicialización implícita de vector: (`[<selector_expr>|| x in <iterable_expr>]`) Su tipo de retorno es un vector del tipo inferido en la expresión del selector. Se infiere el tipo A de la expresión del iterable y en caso de que efectivamente $A \leq Iterable$ se infiere que x es del tipo de retorno del método currente de A.

Otras expresiones: El tipo de expresiones complejas que tienen un cuerpo se determina por el tipo del cuerpo. Este es el caso de las expresiones let, while y for. El tipo de un bloque de expresiones es el tipo de la última expresión del bloque.

Analizando el orden de resolución podremos darnos cuenta de que es probable que en la primera vez que recorra el AST infiriendo tipos no logre inferirlos todos, pero que si lo recorriera de nuevo podría inferir más, por ejemplo:

```
function f(x) => g(x);
function g(x) => x + x;
...
```

En este caso, por el orden en que recorremos, cuando analicemos f(x) en la primera iteración aún no conoceremos el tipo del parámetro, ni el tipo de retorno de g(x). Pero si iteráramos una segunda vez sí pudiéramos resolverlo, quedando en la primera iteración:

```
function f(x: AutoType): AutoType;
function g(x: Number): Number;
...
```

Y en la segunda iteración:

```
function f(x: Number): Number;
function g(x: Number): Number;
...
```

Por esta razón continuamos iterando mientras se cumpla que resolvimos algún AutoType en la iteración anterior. Cuando realicemos una iteración y no logremos inferir ningún tipo podemos asegurar que no podremos inferir en iteraciones posteriores tampoco. llegado este punto tenemos 2 casos posibles:

- Existen tipos que no logramos inferir: En este caso se reporta el error de que no se pudo inferir ese símbolo y que es necesario que se especifique su tipo. Además en este punto el tipo de aquellos símbolos marcados como AutoType se marcan como ErrorType.
- Logramos inferir todos los tipos: En cuyo caso se continúa a la siguiente fase sin reportar errores.

7.4.4. Verificador de tipos

En esta etapa se verifica la consistencia en el uso de los símbolos y la compatibilidad de sus tipos. Los errores que se reportan son:

- WRONG_SIGNATURE (Firma incorrecta): Este error se produce cuando se intenta definir un método en una clase que ya está definido en uno de sus ancestros, pero con una firma diferente. En otras palabras, el método tiene el mismo nombre pero parámetros diferentes en la clase actual y en una de sus superclases.

- SELF_IS_READONLY (Self es de solo lectura): Ocurre cuando se intenta modificar la variable "self", que se utiliza para referirse al objeto actual.
- INCOMPATIBLE_TYPES (Tipos incompatibles): Este error se genera cuando se intenta realizar una conversión entre dos tipos que no son compatibles entre sí. O sea que el tipo inferido para la expresión no se conforma al tipo correspondiente para esa expresión.
- VARIABLE_NOT_DEFINED (Variable no definida): Se produce cuando se intenta acceder a una variable que no ha sido definida previamente.
- INVALID_OPERATION (Operación no válida): Este error ocurre cuando se intenta realizar una operación entre tipos de datos que no admiten esa operación en particular.
- EXPECTED_ARGUMENTS (Argumentos esperados): Se produce cuando se llama a una función o método con un número incorrecto de argumentos.
- BASE_OUTSIDE_METHOD (Base fuera de un método): Este error ocurre cuando se intenta utilizar la palabra clave "base"fuera del contexto de un método. La palabra clave "base"se utiliza para acceder a los miembros de la clase base en la que se está realizando la herencia.

8. Generación de código