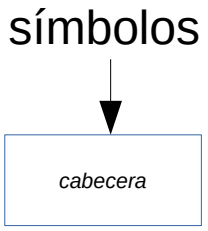
Análisis semántico

- Verificación de la parte **sensible al contexto** de miniC
  - Un programa sintácticamente correcto puede contener errores semánticos
  - Hay que verificar la declaración y uso correcto de los identificadores
- Nos apoyamos en la gramática (parte libre del contexto) y una lista de símbolos, distinguiendo las *declaraciones* y los *usos* de los identificadores.
- Se realiza en miniC.y
- La lista de símbolos almacena información sobre los **identificadores**.
- Posible implementación: listaSimbolos.c y listaSimbolos.h (Aula Virtual)
- La lista de símbolos *también* nos ayuda a generar parte de la salida en ensamblador (.data)

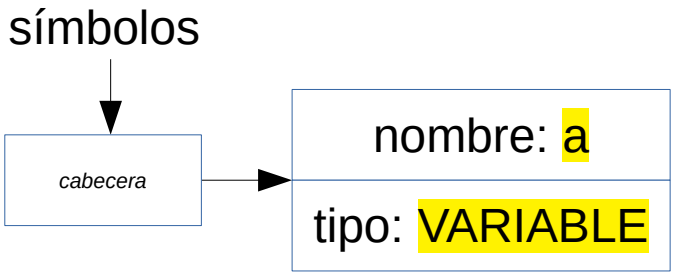
Análisis semántico (1)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```



Análisis semántico (2)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```

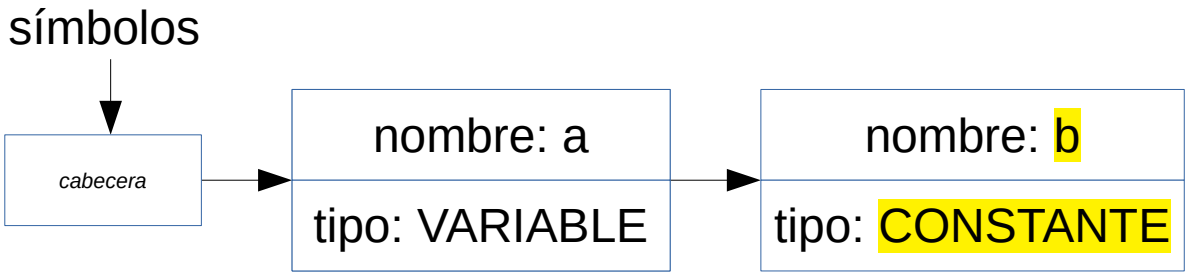


declarations → declarations **var** tipo **var_list ;**

var_list → id { /* lexema en $1, tipo var */ }

Análisis semántico (3)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```



declarations → declarations **const** tipo **const_list ;**

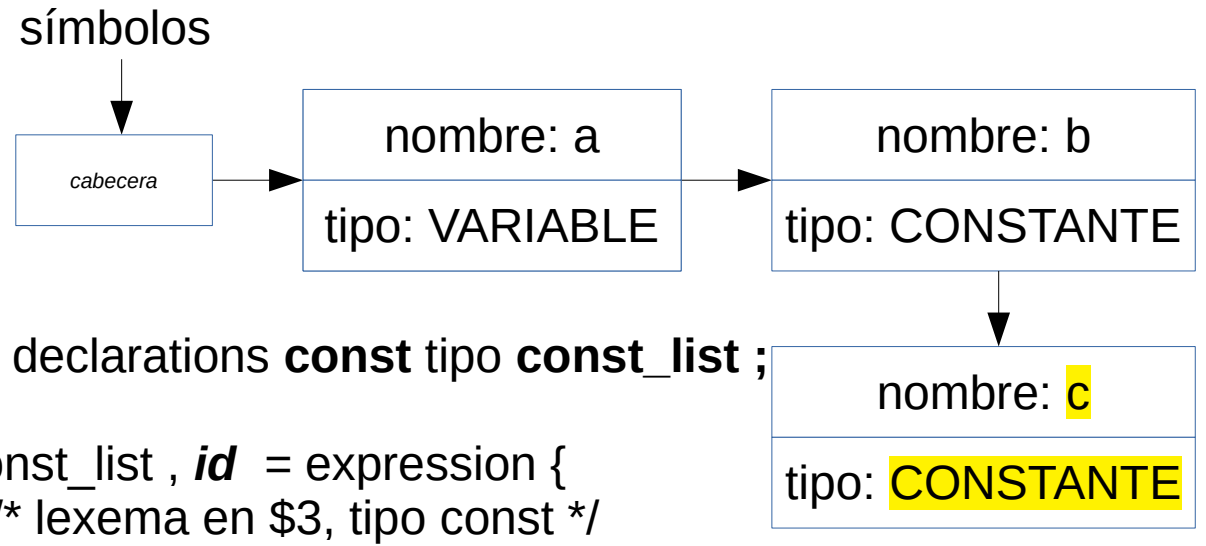
const_list → ***id*** = expression { /* lexema en $1, tipo const */ }

Análisis semántico (4)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```

declarations → declarations **const** tipo **const_list ;**

const_list → const_list , ***id*** = expression {
/* lexema en $3, tipo const */
}

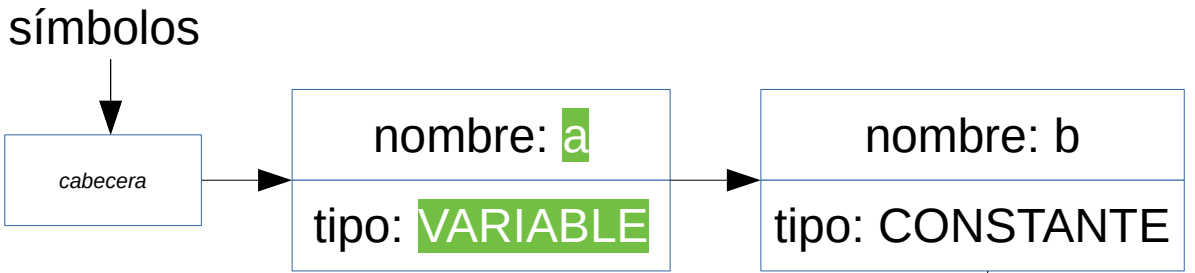


Análisis semántico (5)

- Revisión de la lista de símbolos (Aula Virtual)
- Algunas ideas para usar la lista de símbolos en miniC.y
  - ¿Dónde se declara?
  - ¿Dónde se inicializa?
  - ¿Dónde se destruye?
- Cambios necesarios en makefile

Análisis semántico (6)

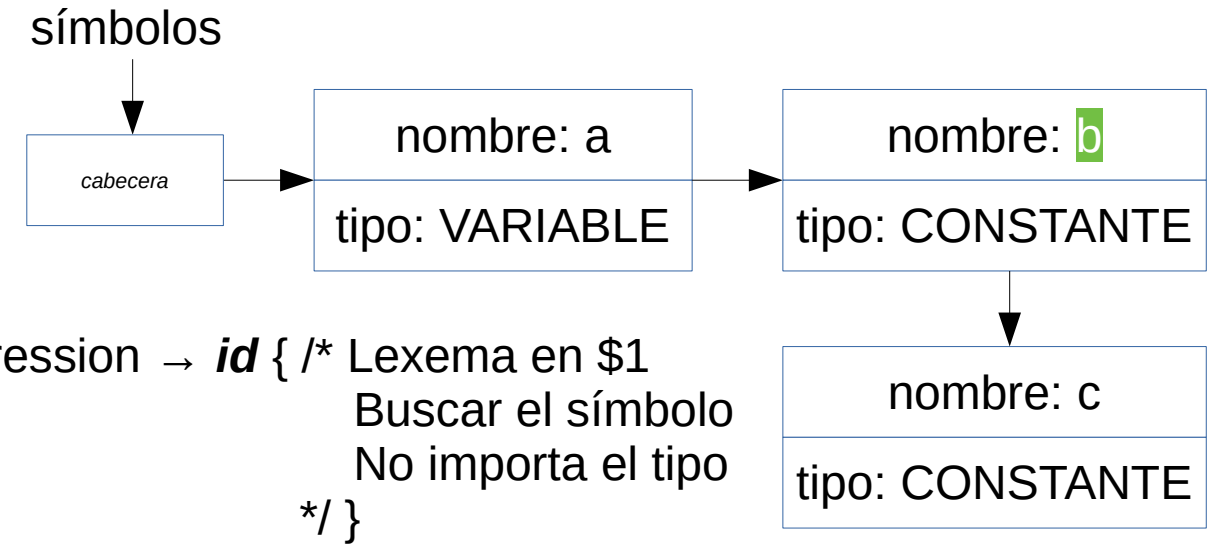
```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```




```
statement → id  = expression {
                /* Lexema en $1
                   Buscar el símbolo
                   y comprobar que
                   es VARIABLE */
            }
```

Análisis semántico (7)

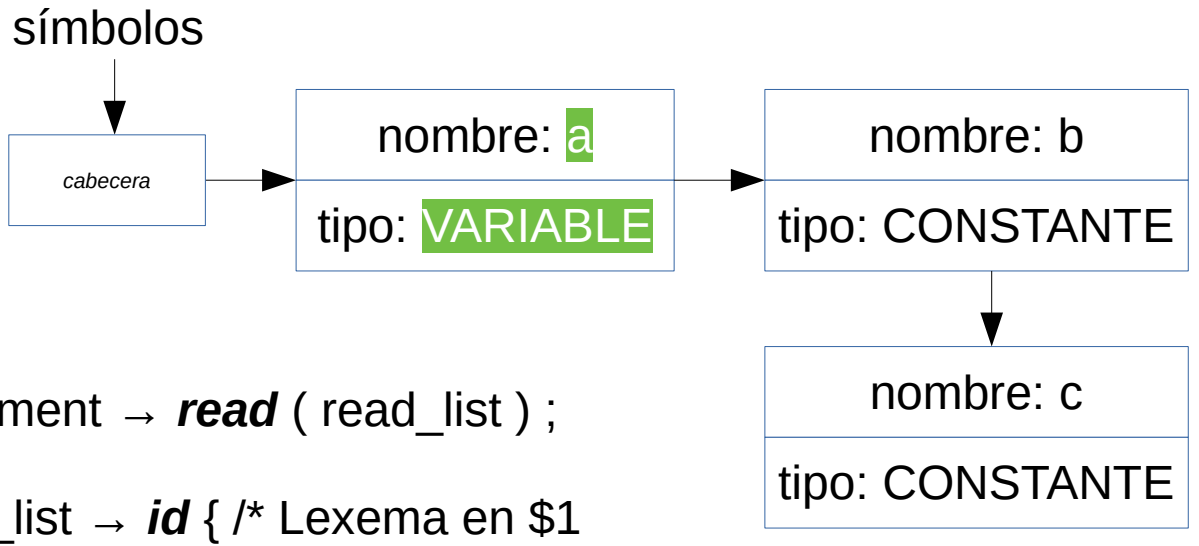
```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```



expression → ***id*** { /* Lexema en $1
Buscar el símbolo
No importa el tipo
*/ }

Análisis semántico (8)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```

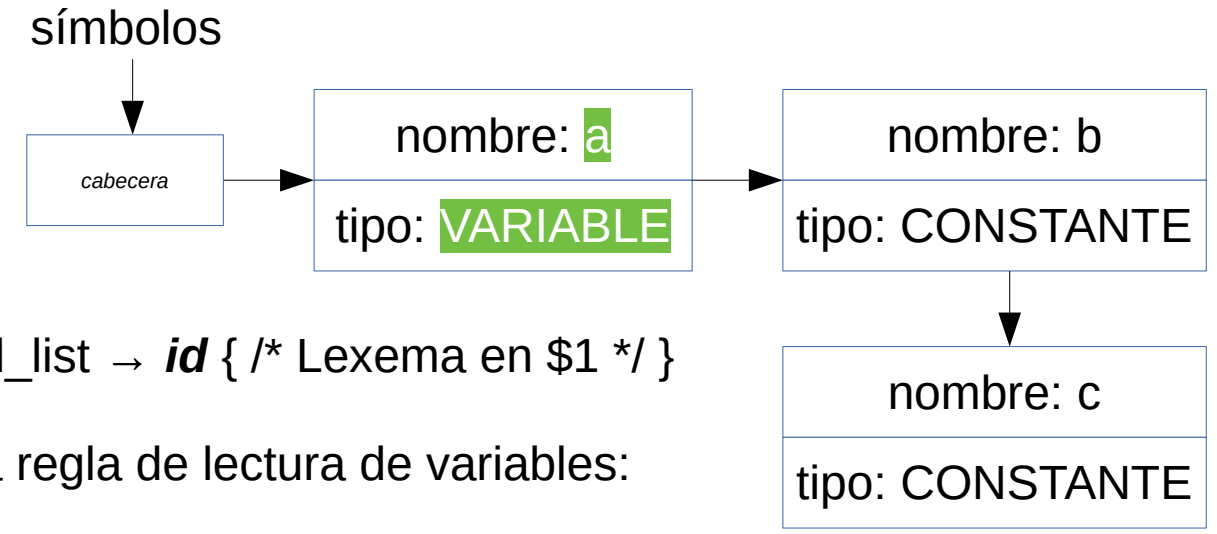


statement → ***read*** (read_list) ;

read_list → ***id*** { /* Lexema en $1
Buscar el símbolo
y comprobar que
es VARIABLE */ }

Análisis semántico (9)

```
test1() {

    var int a;
    const int b = 3, c = 0;

    a = 1 + b;
    read (a);

}
```



read_list → ***id*** { /* Lexema en $1 */ }

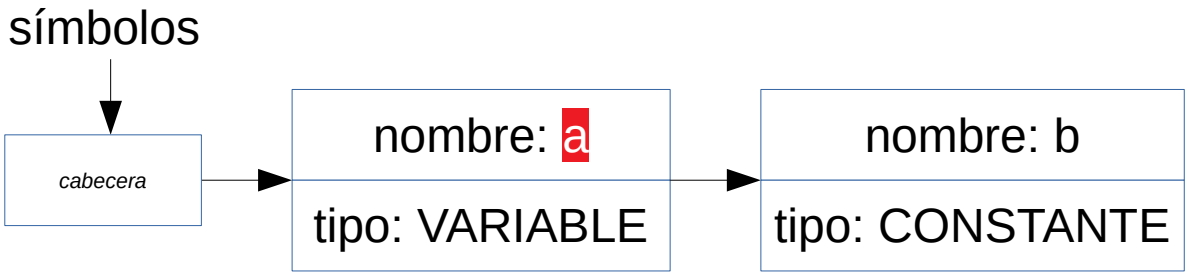
Otra regla de lectura de variables:

read (a,b);

read_list → read_list , ***id*** { /* lexema en $3 */ }

Errores semánticos (1)

```
test2() {

    var int a;
    const int b = 3;

    var int a;
    const int a = 0;

    a = 1 + b;

}
```



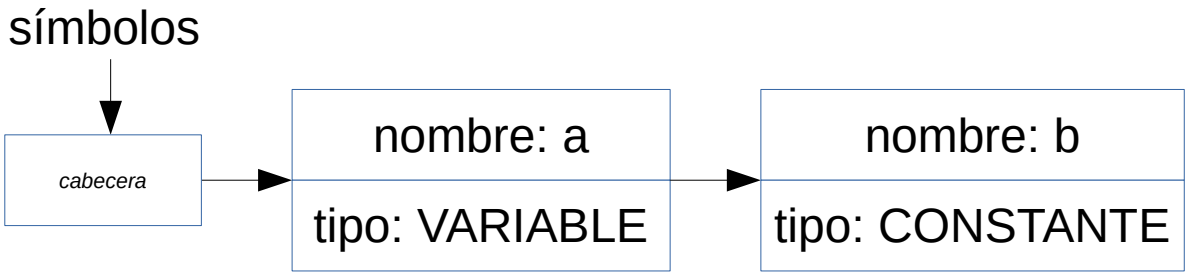
Los identificadores están redeclarados

var_list → ***id*** { /* lexema en $1 */ }
const_list → ***id*** = expression { /* lexema en $1 */ }

Hay que considerar las otras reglas de var_list y const_list

Errores semánticos (2)

```
test3() {

    var int a;
    const int b = 3;

    x = 1;
    a = 2*y;
    read (z);

}
```

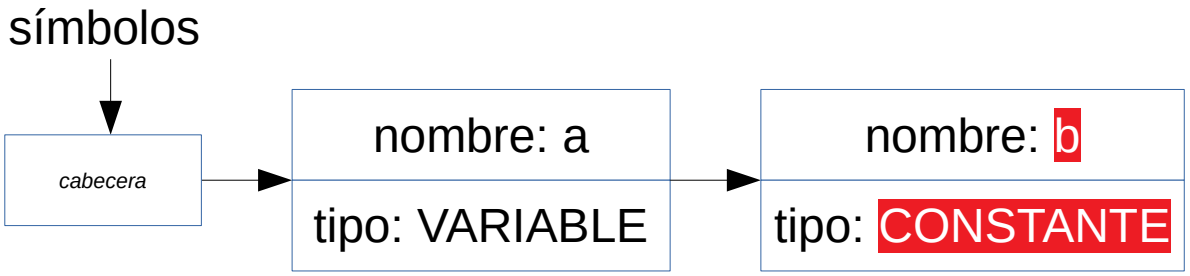


Los identificadores no están declarados

statement → ***id*** = expression { /* Lexema en $1 */ }
expression → ***id*** { /* Lexema en $1 */ }
read_list → ***id*** { /* Lexema en $1 */ }
read_list → read_list , ***id*** { /* Lexema en $3 */ }

Errores semánticos (3)

```
test4() {

    var int a;
    const int b = 3;

    b = 1;
    read (b);

}
```



Asignación a una constante

statement → ***id*** = expression { /* Lexema en $1 */ }
read_list → ***id*** { /* Lexema en $1 */ }
read_list → read_list , ***id*** { /* Lexema en $3 */ }

Resumen de análisis semántico (1)

- Declaraciones:

var_list → ***id***	{ Si $1 **sí** está en la tabla: ¡error! En caso contrario, insertar en la tabla }
var_list → var_list , ***id***	{ Si $3 **sí** está en la tabla: ¡error! En caso contrario, insertar en la tabla }
const_list → ***id*** = expression	{ Si $1 **sí** está en la tabla: ¡error! En caso contrario, insertar en la tabla }
const_list → const_list, ***id*** = expression	{ Si $3 **sí** está en la tabla: ¡error! En caso contrario, insertar en la tabla }

Resumen de análisis semántico (2)

- Usos:

statement → ***id*** = expression { Si $1 **no** está en la tabla: ¡error!
En caso contrario, si $1 es CONSTANTE: ¡error!
}

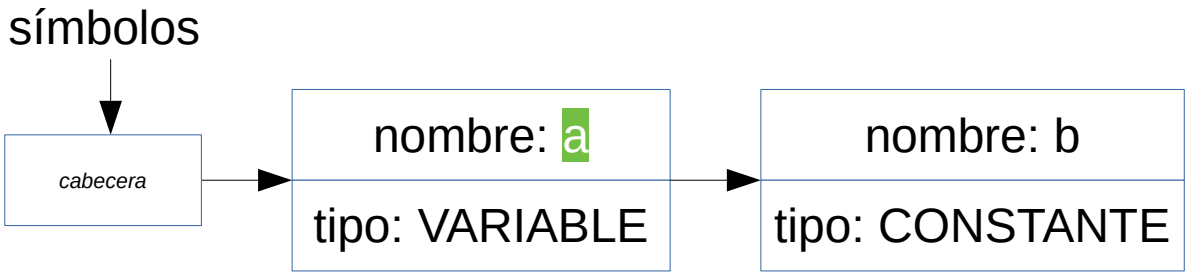
read_list → ***id*** { Si $1 **no** está en la tabla: ¡error!
En caso contrario, si $1 es CONSTANTE: ¡error!
}

read_list → read_list , ***id*** { Si $3 **no** está en la tabla: ¡error!
En caso contrario, si $3 es CONSTANTE: ¡error!
}

expression → ***id*** { Si $1 **no** está en la tabla: ¡error! }

Generación de código (1)

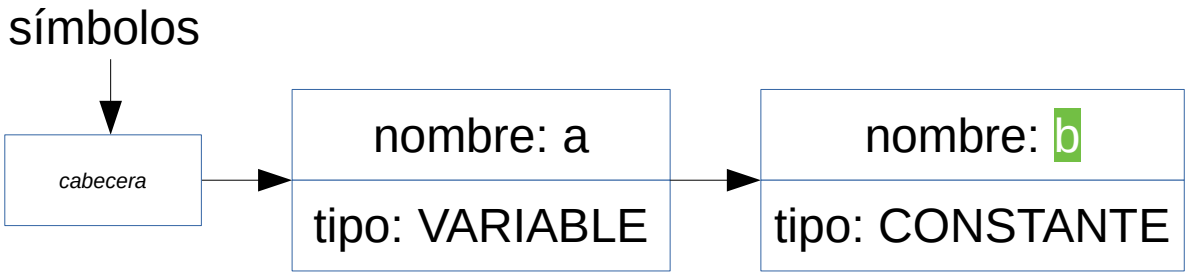
```
test5() {

    var int a;
    const int b = 3;

    a = 1 + b;
    read (a);

}
```




```
.data

# Identificadores
_a : .word 0
```

Generación de código (2)

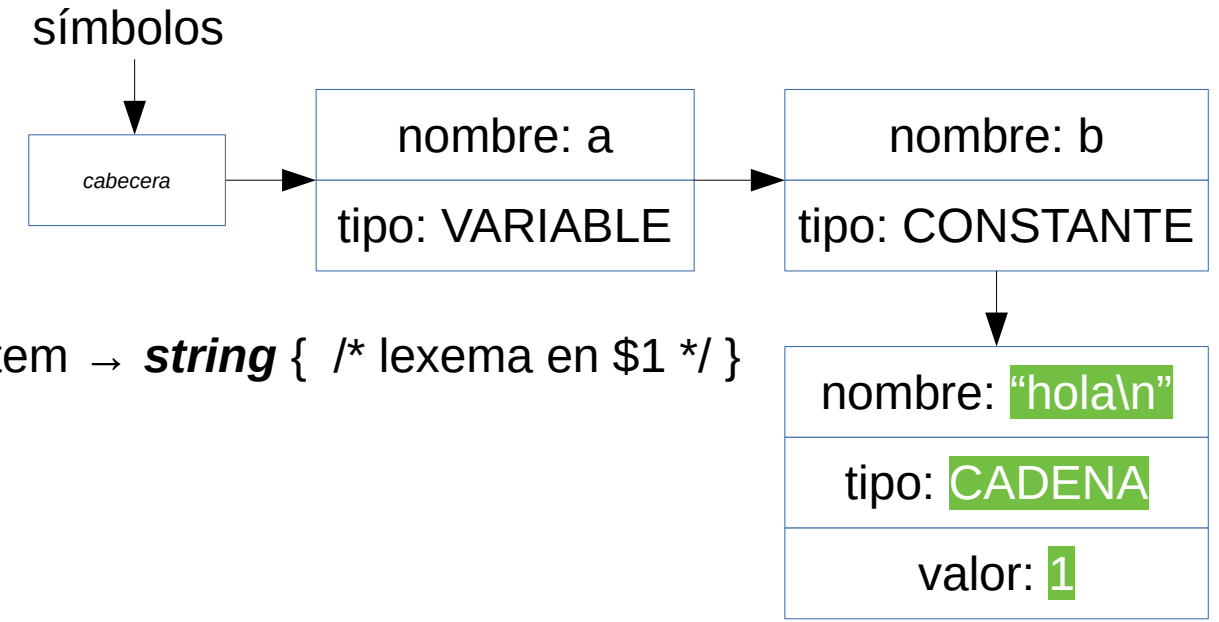
```
test5() {

    var int a;
    const int b = 3;

    a = 1 + b;
    read (a);

}
```




```
.data

# Identificadores
_a : .word 0
_b : .word 0
```

Cadenas de caracteres

```
test6() {

    var int a;
    const int b = 3;

    print (“hola\n”);

}
```



print_item → ***string*** { /* lexema en $1 */ }

Generación de código (3)

```
test6() {

    var int a;
    const int b = 3;

    print (“hola\n”);

}
```