Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Físico Químicas y Naturales

Departamento de Computación

Taller de Diseño de Software

Proyecto
Quinta entrega

Campregher Bruno
Machuca Franco
Natali Valentino

Optimizador

1. Introducción

En esta etapa del compilador se realizan una serie de transformaciones sobre diferentes representaciones del programa con el objetivo de mejorar la eficiencia del código generado sin alterar su comportamiento semántico. El optimizador toma decisiones estratégicas en distintos momentos del proceso de compilación para producir código más compacto y eficiente.

Las optimizaciones implementadas actúan en múltiples niveles de la compilación. Algunas transformaciones se aplican sobre el árbol de sintaxis abstracta (AST) simplificando la estructura del programa antes de la generación de código intermedio, mientras que otras operan directamente durante la generación del código intermedio optimizando el uso de recursos como variables temporales y direccionamiento de memoria.

El resultado de estas optimizaciones es un código intermedio mejorado que reduce operaciones redundantes, elimina código innecesario y utiliza de manera más eficiente los recursos disponibles. Esto se traduce posteriormente en un código objeto más compacto y con mejor rendimiento en tiempo de ejecución.

Repositorio del proyecto

El **código fuente completo** del proyecto está disponible en nuestro **repositorio de GitHub**.

La rama `main` contiene la versión correspondiente a la **entrega actual**.

2. Optimizaciones implementadas

2.1 Propagación de constantes

La propagación de constantes es una técnica de optimización que evalúa expresiones en tiempo de compilación cuando todos sus operandos son valores constantes conocidos. Esta optimización se implementa reduciendo operaciones aritméticas y lógicas a sus valores resultantes antes de generar código intermedio.

Funcionamiento

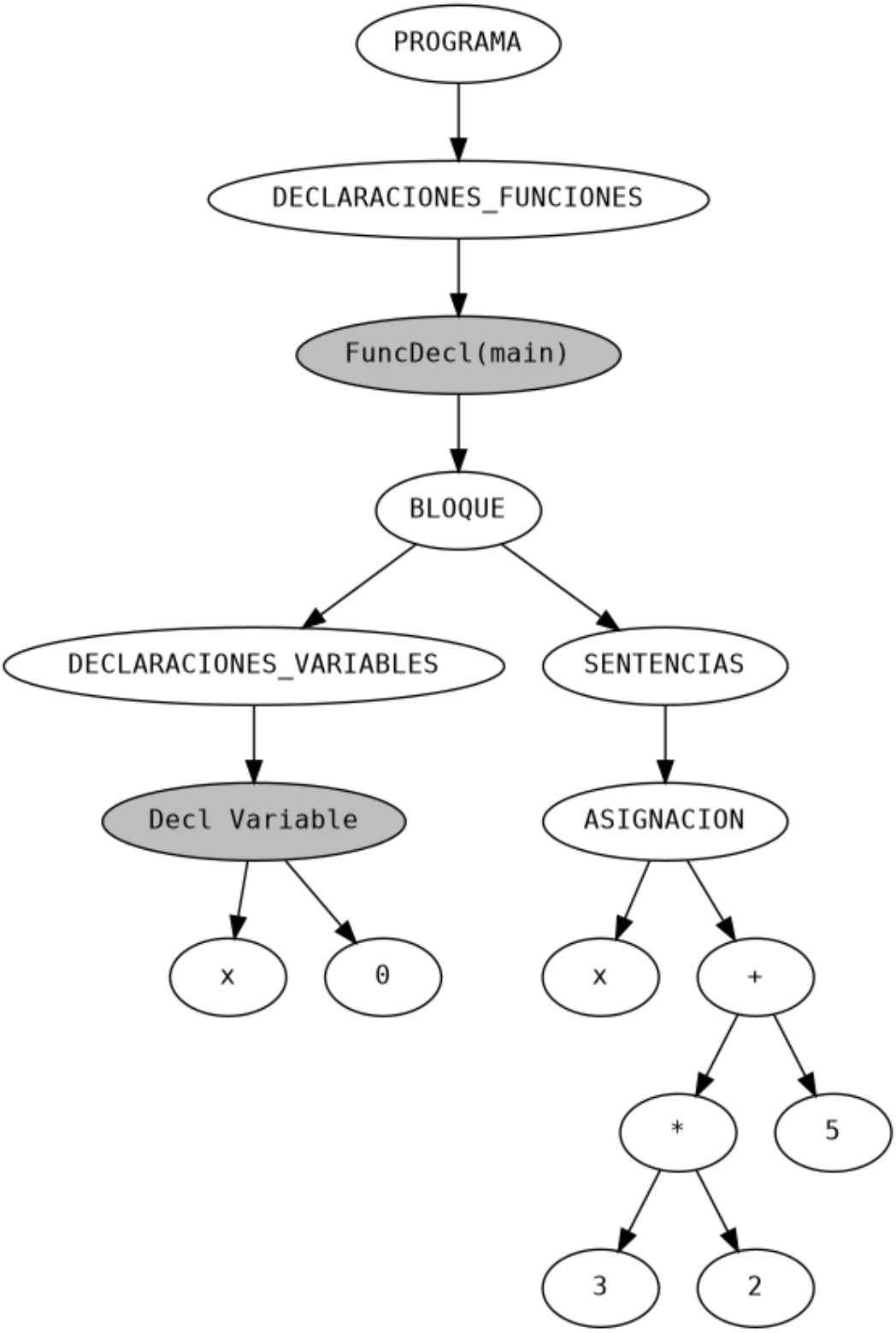
La optimización se aplica sobre el árbol de sintaxis abstracta (AST) mediante un recorrido recursivo que identifica nodos de operadores (binarios y unarios) cuyos operandos son literales. Cuando se detecta esta situación, la operación se evalúa en tiempo de compilación y el subárbol del operador se reemplaza por un único nodo literal con el resultado.

Ejemplo:

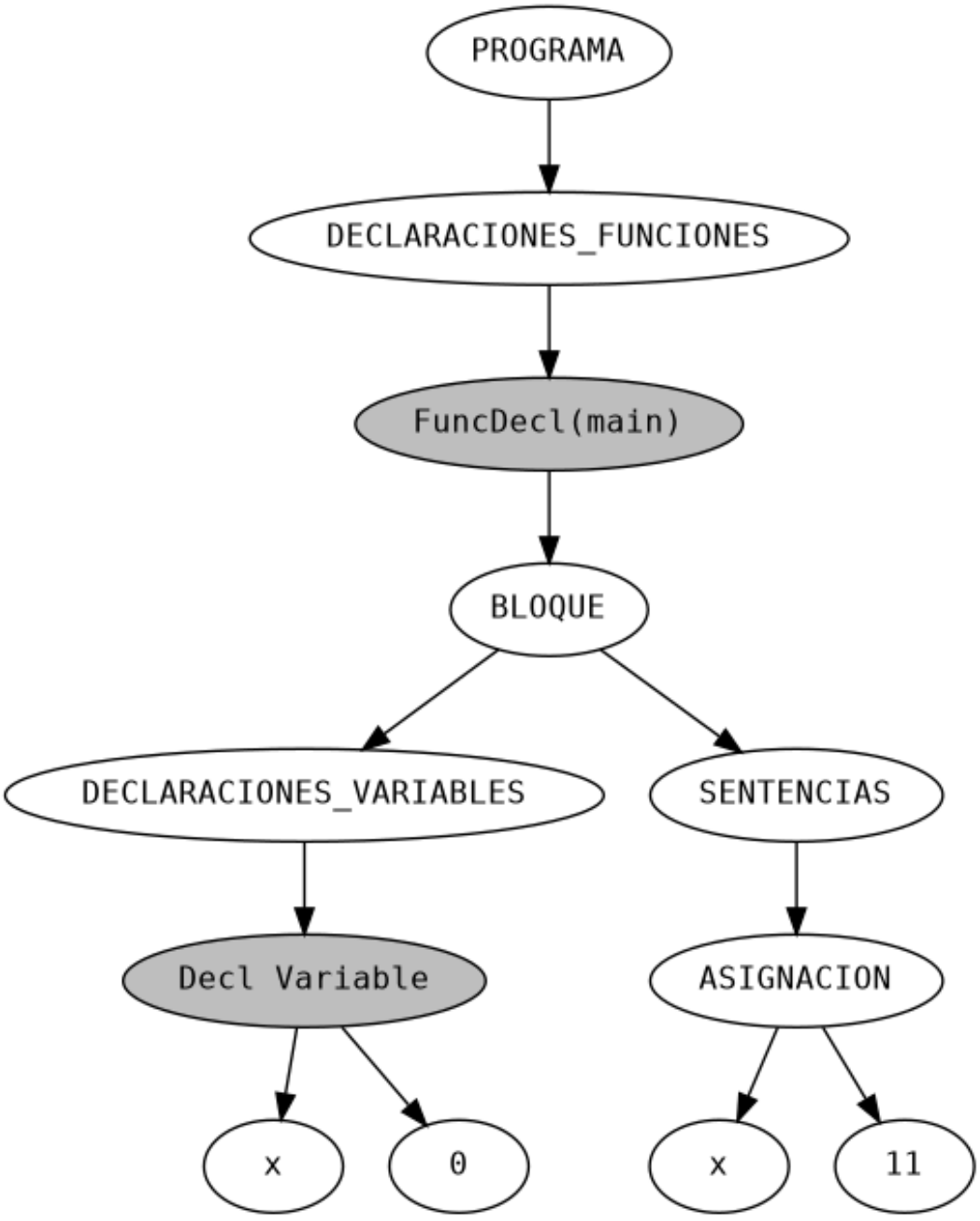
```
program
{
	void main()
	{
		integer x = 0;
		x = 3 * 2 + 5;
	}
}
```

A continuación se muestra la estructura del AST antes y después de aplicar la propagación de constantes para una expresión aritmética simple:

AST sin optimizar:



AST optimizado:



Como se observa en las figuras, el subárbol que contenía múltiples nodos de operadores y literales se reduce a un único nodo literal con el valor calculado (11).

Impacto en el código intermedio generado:

Código intermedio **sin optimización**:

```
main:
START_FUN main
MOV 0 x
MULT 3 2 t0
ADD t0 5 t0
MOV t0 x
END_FUN
```

Código intermedio **con optimización**:

```
main:
START_FUN main
MOV 0 x
MOV 11 x
END_FUN
```

2.2 Eliminación de Código Muerto

La eliminación de código muerto es una optimización que identifica y remueve del AST aquellas porciones de código que nunca serán ejecutadas o cuyos resultados nunca serán utilizados. Esta técnica reduce el tamaño del árbol y, consecuentemente, la cantidad de código intermedio generado, mejorando tanto el tiempo de compilación como la eficiencia del programa final.

Funcionamiento

La optimización se divide en tres estrategias principales que se aplican sobre el AST:

Eliminación de declaraciones no utilizadas

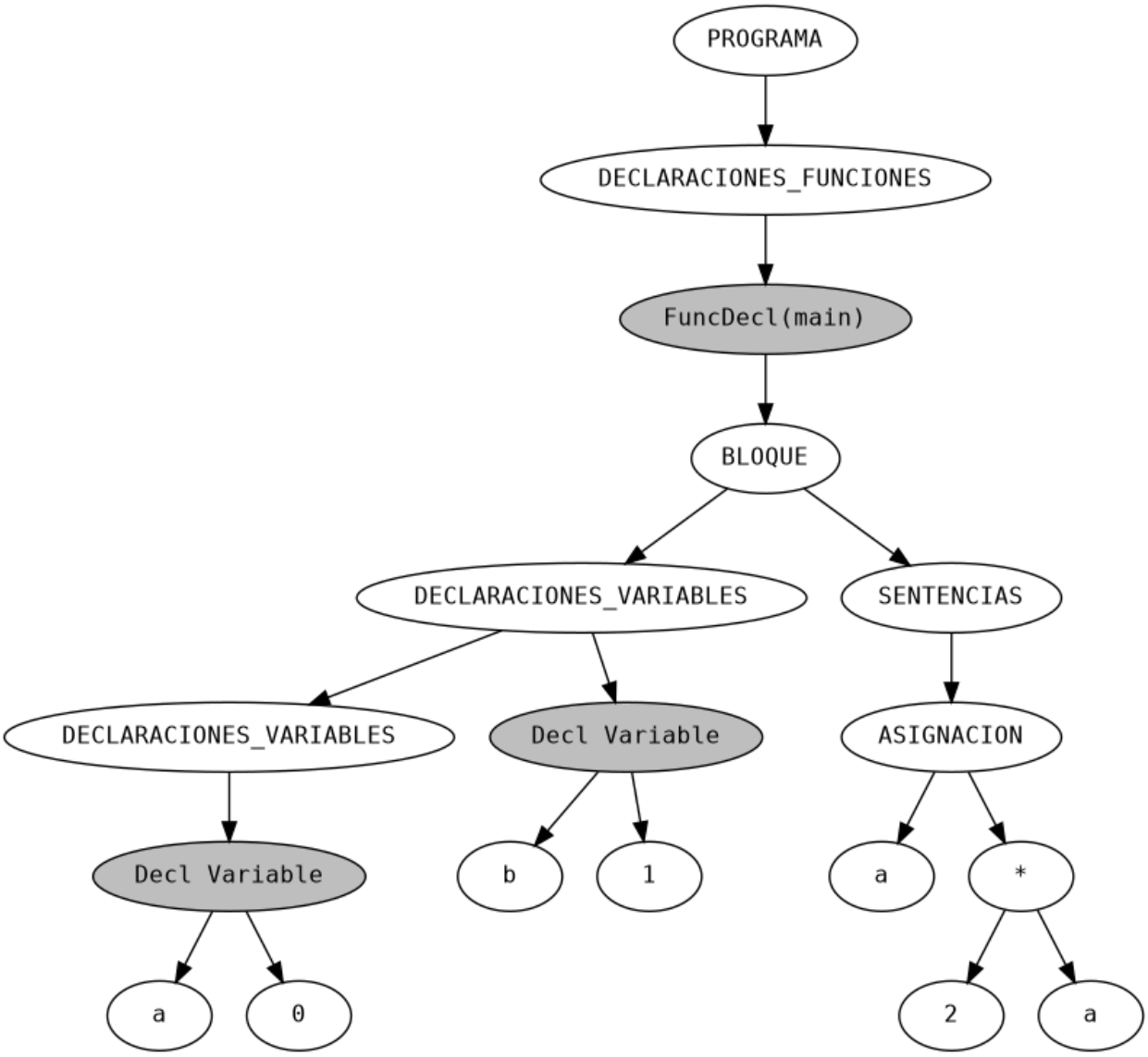
Se analizan las declaraciones de variables y funciones identificando aquellas que no tienen usos en el programa. Durante las etapas previas de análisis semántico, se contabiliza el número de referencias a cada símbolo. Si una declaración presenta contador de usos en cero, se elimina del árbol y se emite un warning informativo.

Ejemplo:

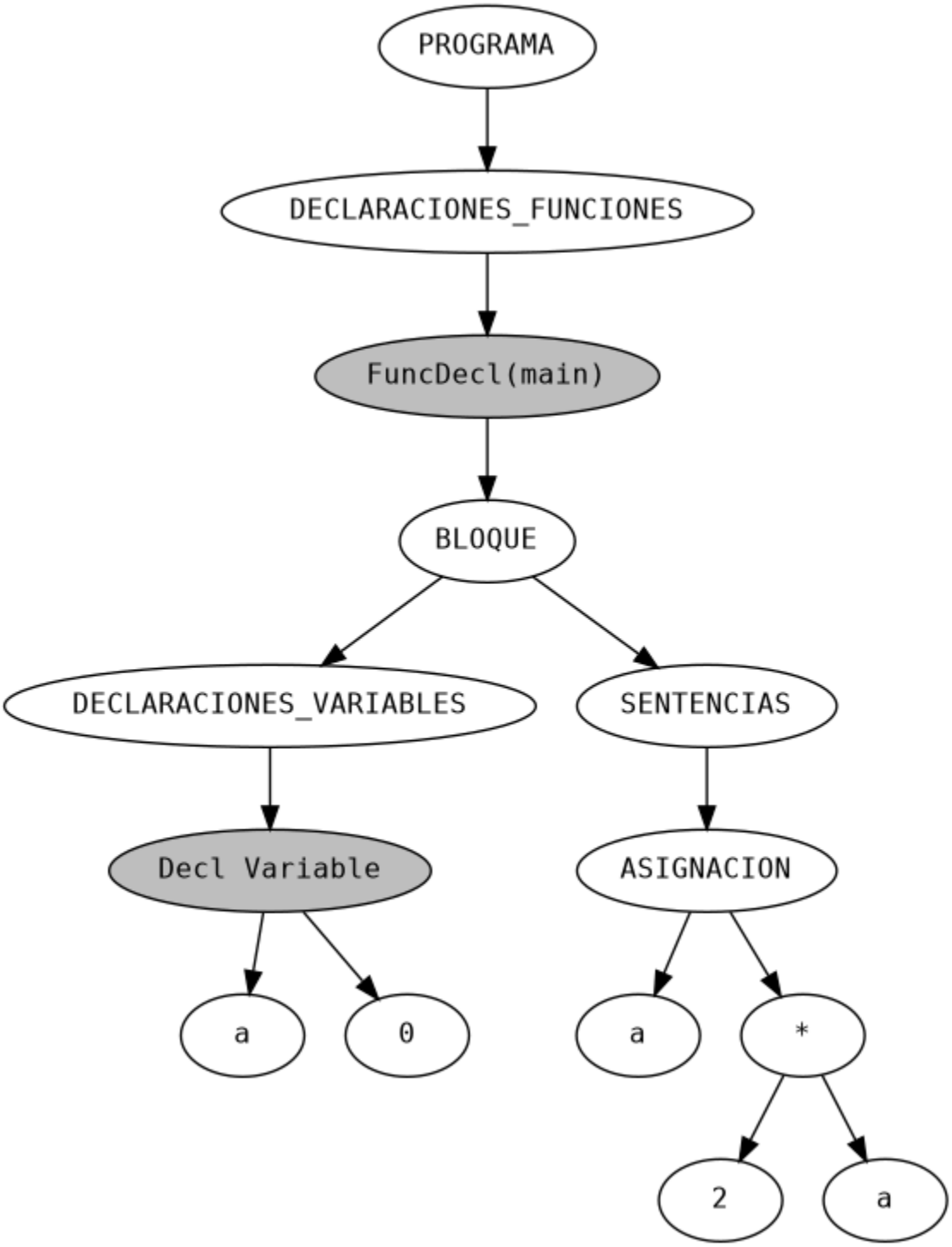
```
program
{
	void main()
	{
		integer a = 0;
		integer b = 1;

		a = 2 * a;
	}
}
```

A continuación se muestra la estructura del AST antes y después de aplicar la eliminación de variables no utilizadas:

AST sin optimizar:



AST optimizado:



La declaración de ‘b’ es eliminada del árbol y se genera el siguiente warning informativo:

```
Warning: Linea 5 Col 27
└── Warning: variable 'b' declarada pero no utilizada
```

El funcionamiento para funciones declaradas pero nunca invocadas es análogo: se detectan aquellas funciones con contador de usos en cero, se eliminan completamente del AST y se emite un warning similar indicando que la función fue declarada pero no utilizada.

Eliminación de código inalcanzable después de return

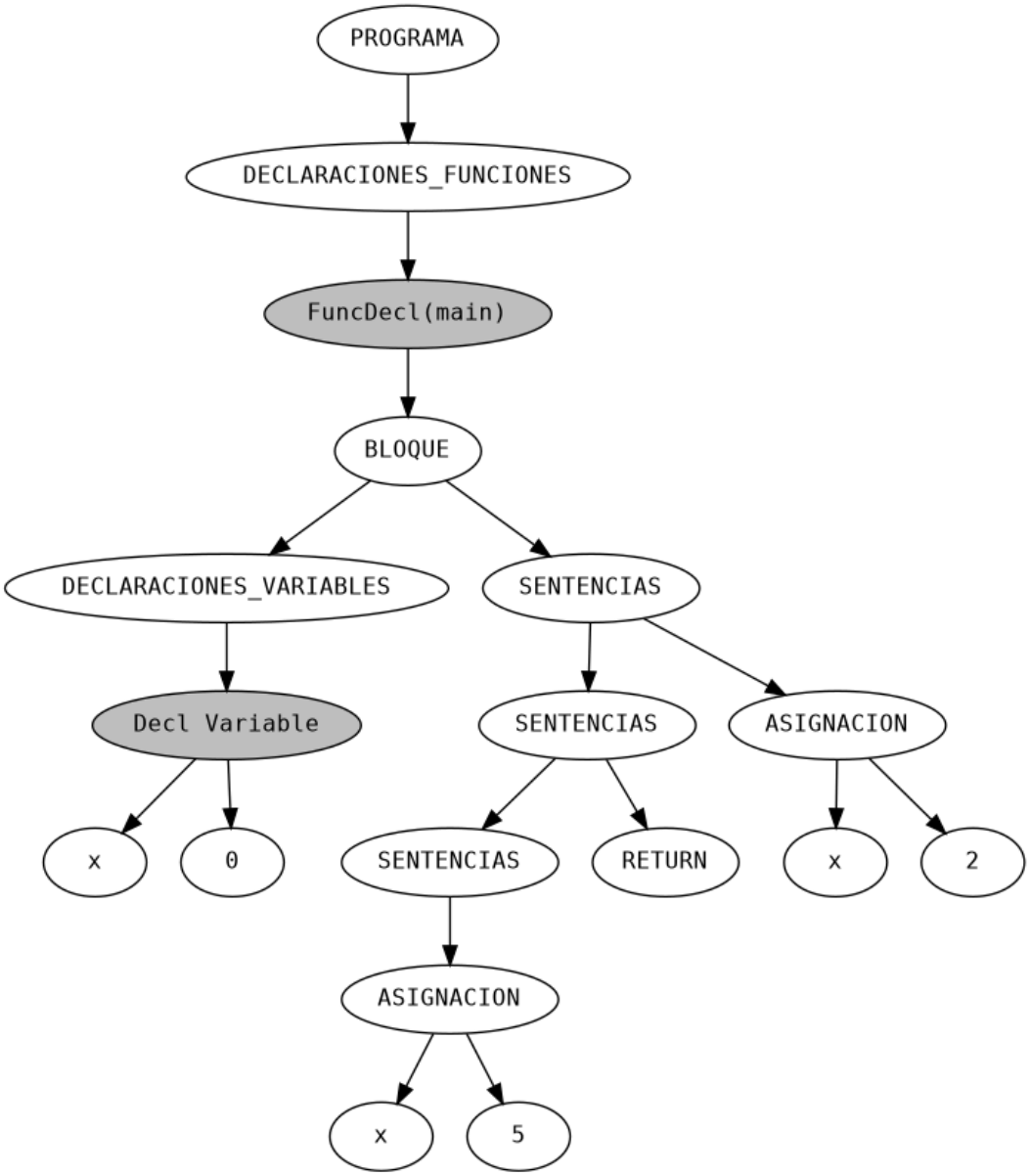
Se detectan sentencias que aparecen después de una instrucción return dentro del mismo bloque. Dado que el return finaliza la ejecución de la función, cualquier código posterior es inalcanzable y puede ser eliminado de forma segura. La optimización recorre el árbol identificando nodos de tipo SENTENCIAS y verifica si el hijo izquierdo contiene un return, en tal caso, elimina completamente el subárbol derecho.

Ejemplo:

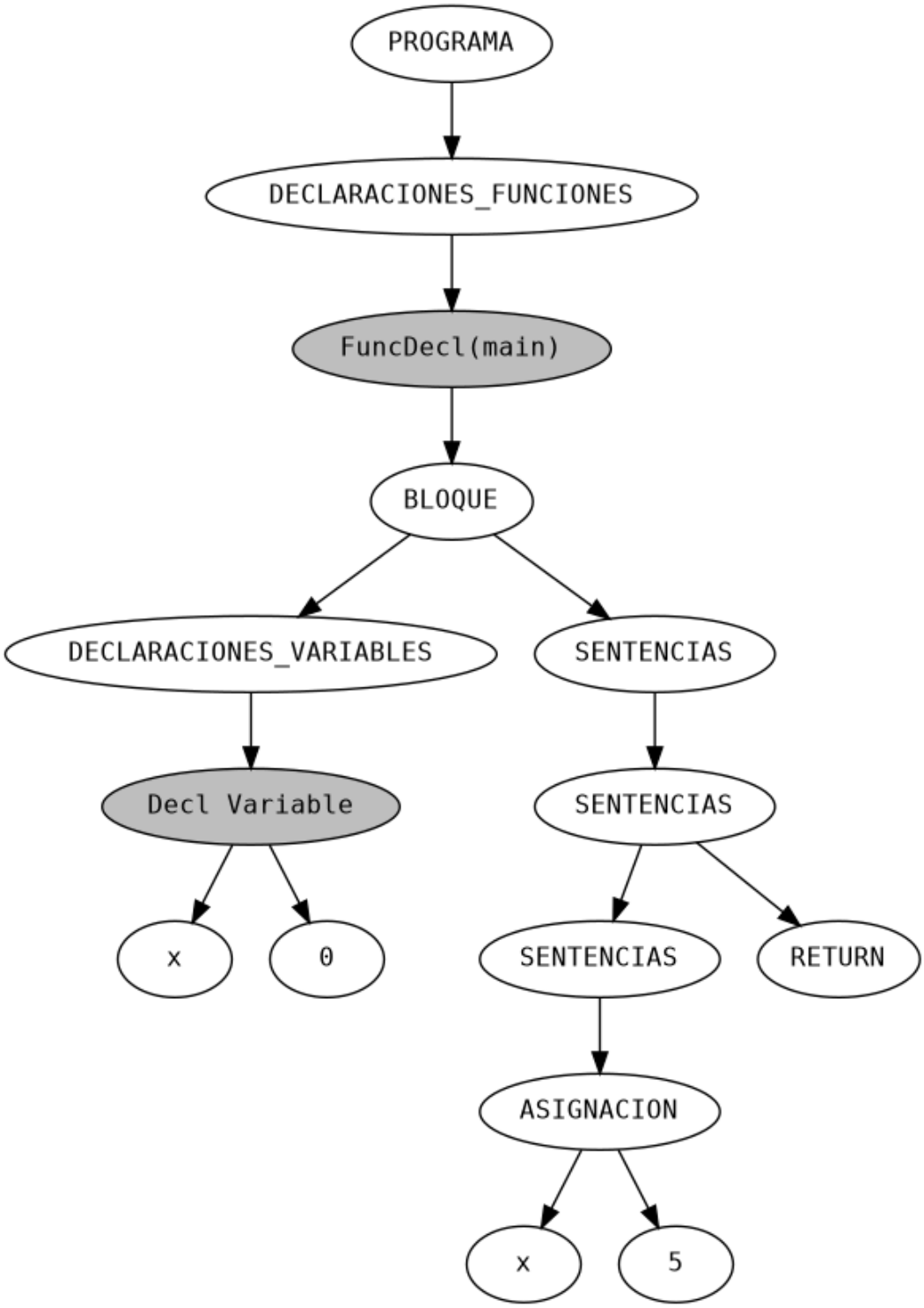
```
program
{
	void main()
	{
		integer x = 0;
		x = 5;
		return;
		x = 2;
	}
}
```

A continuación se muestra la estructura del AST antes y después de aplicar la eliminación de código posterior a un return:

AST sin optimizar:



AST optimizado:

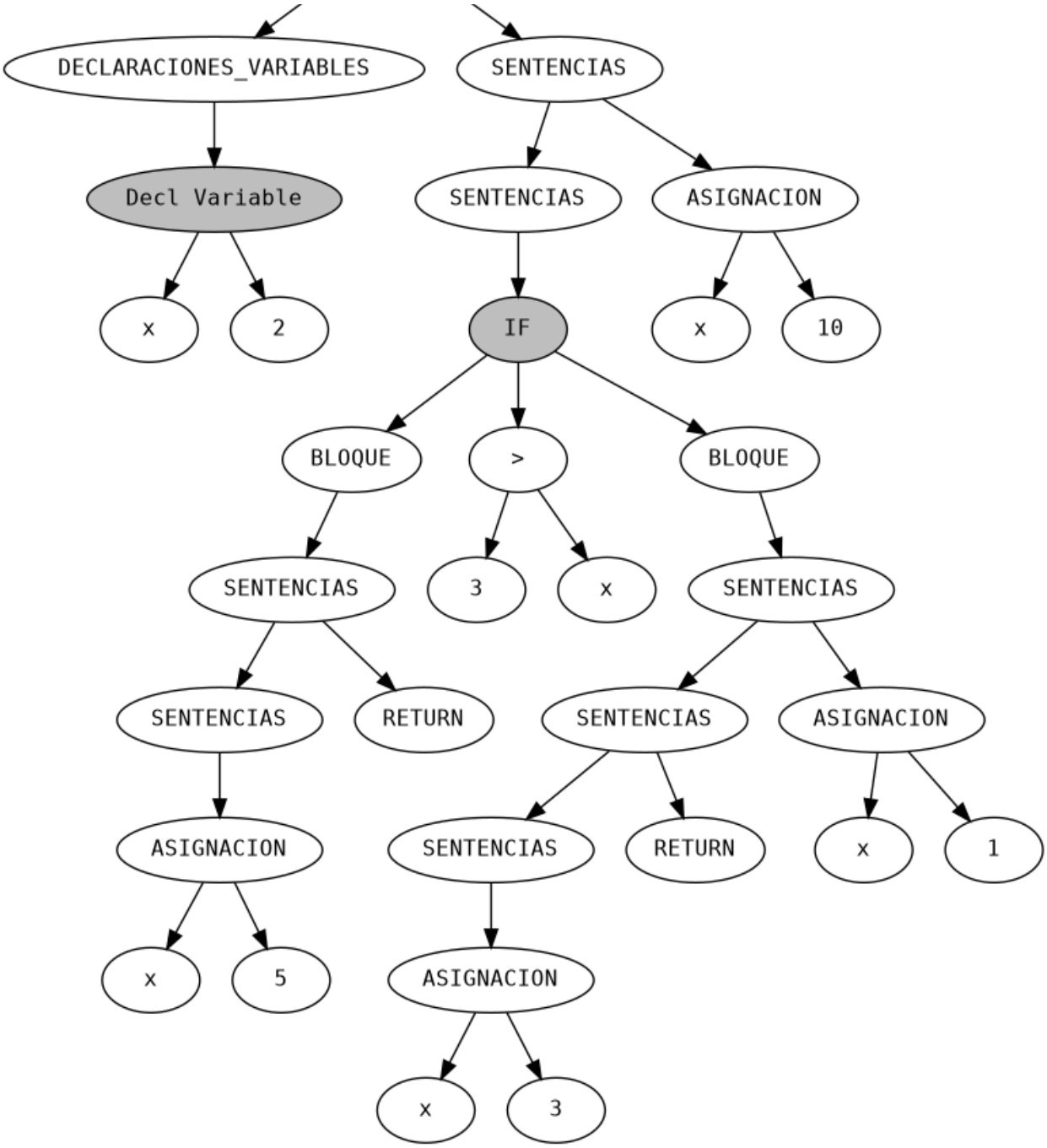


Ejemplo:

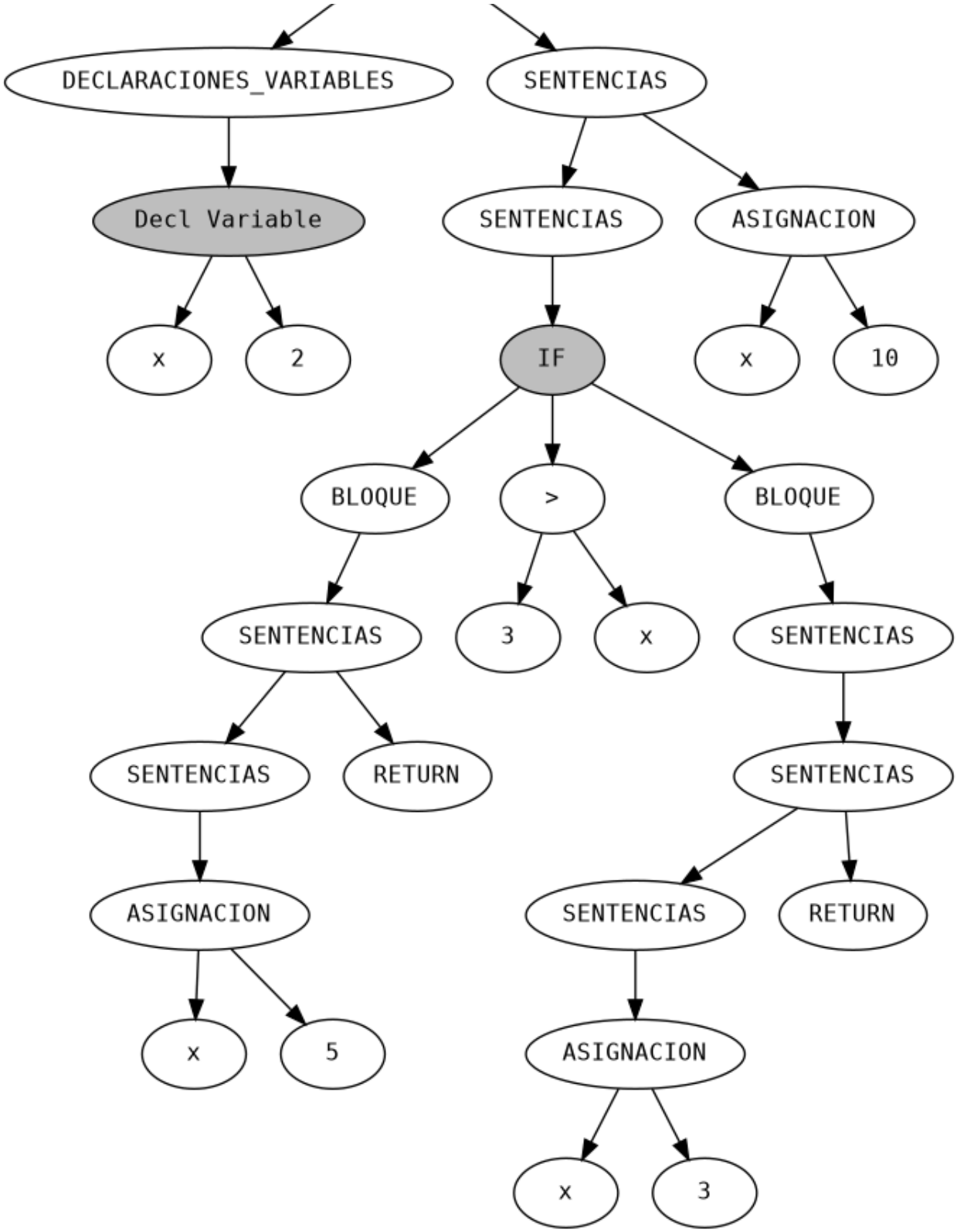
```
program
{
	void main()
	{
		integer x = 2;
		if(3 > x) then
		{
			x = 5;
			return;
		}
		else
		{
			x = 3;
			return;
			x = 1;
		}

		x = 10;
	}
}
```

A continuación se muestra la estructura del AST antes y después de aplicar la eliminación de código posterior a un return:

AST sin optimizar:



AST optimizado:



En este ejemplo se muestra cómo el optimizador realiza la eliminación de código muerto mediante un análisis conservador del flujo de control.

Dentro de la rama `else` del `if`, la instrucción `x = 3;` es seguida por un `return`, lo que hace que la línea `x = 1;` nunca se ejecute. El optimizador detecta esta instrucción como inalcanzable y la elimina de manera segura del AST final.

Es importante destacar que la última instrucción, `x = 10;`, no se elimina, aunque en la práctica no se ejecutaría debido a que ambas ramas del `if` terminan con `return`. Esto se debe a que el análisis realizado es conservador: solo se eliminan instrucciones que es seguro afirmar que nunca se ejecutarán en cualquier flujo posible del programa. De esta forma se evita eliminar código que podría ser necesario en otras condiciones

Optimización de estructuras de control con condiciones constantes

Cuando las condiciones de estructuras if o while son literales booleanos conocidos en tiempo de compilación, se puede determinar estáticamente qué rama será ejecutada:

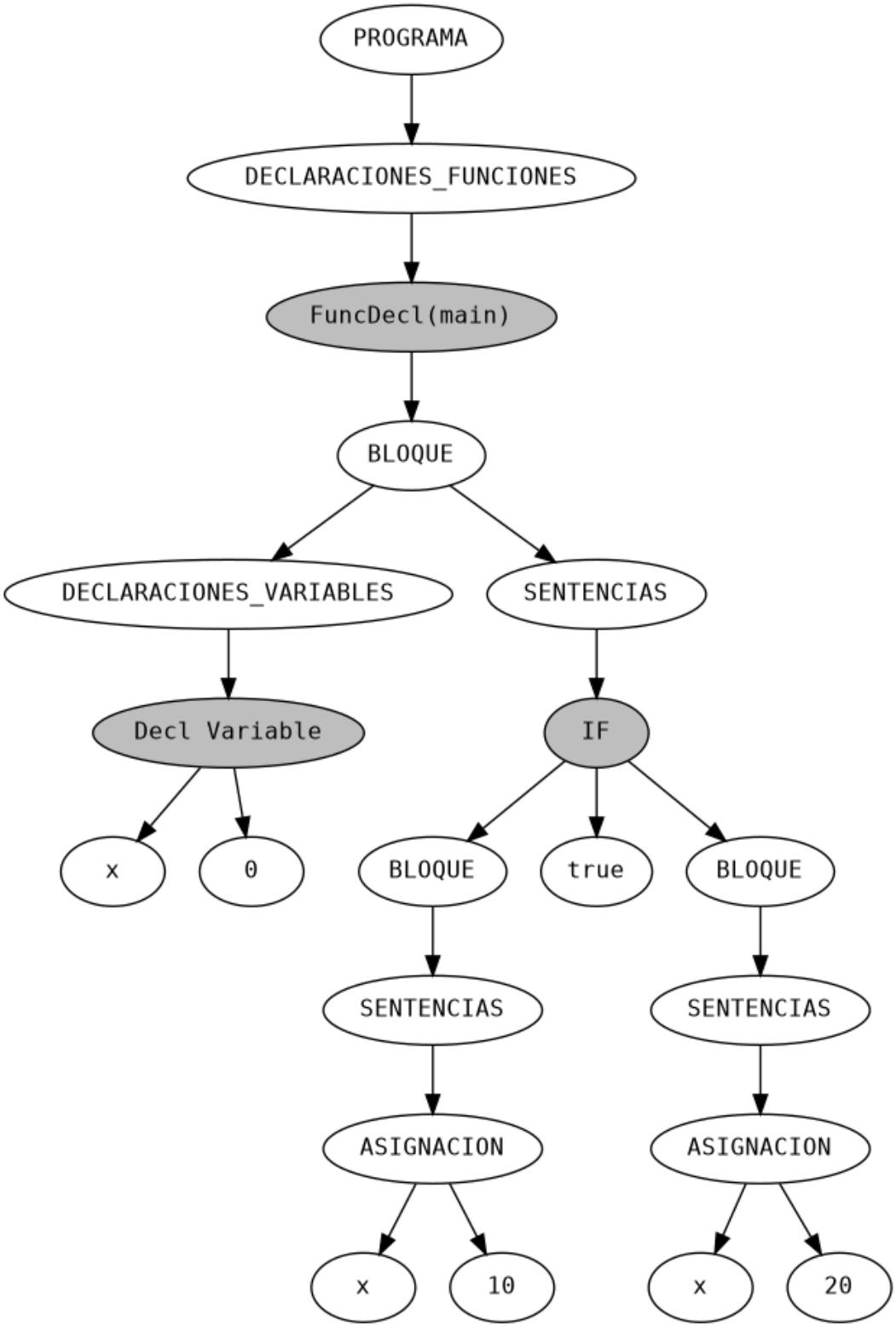
- **If con condición true**: Se reemplaza todo el nodo if por el bloque then, eliminando la condición y el bloque else
- **If con condición false**: Se reemplaza todo el nodo if por el bloque else (si existe), eliminando la condición y el bloque then
- **While con condición false**: Se elimina completamente el bucle, ya que nunca se ejecutará
- **While con condición true**: Se mantiene el bucle pero se emite un warning sobre posible bucle infinito

Ejemplo:

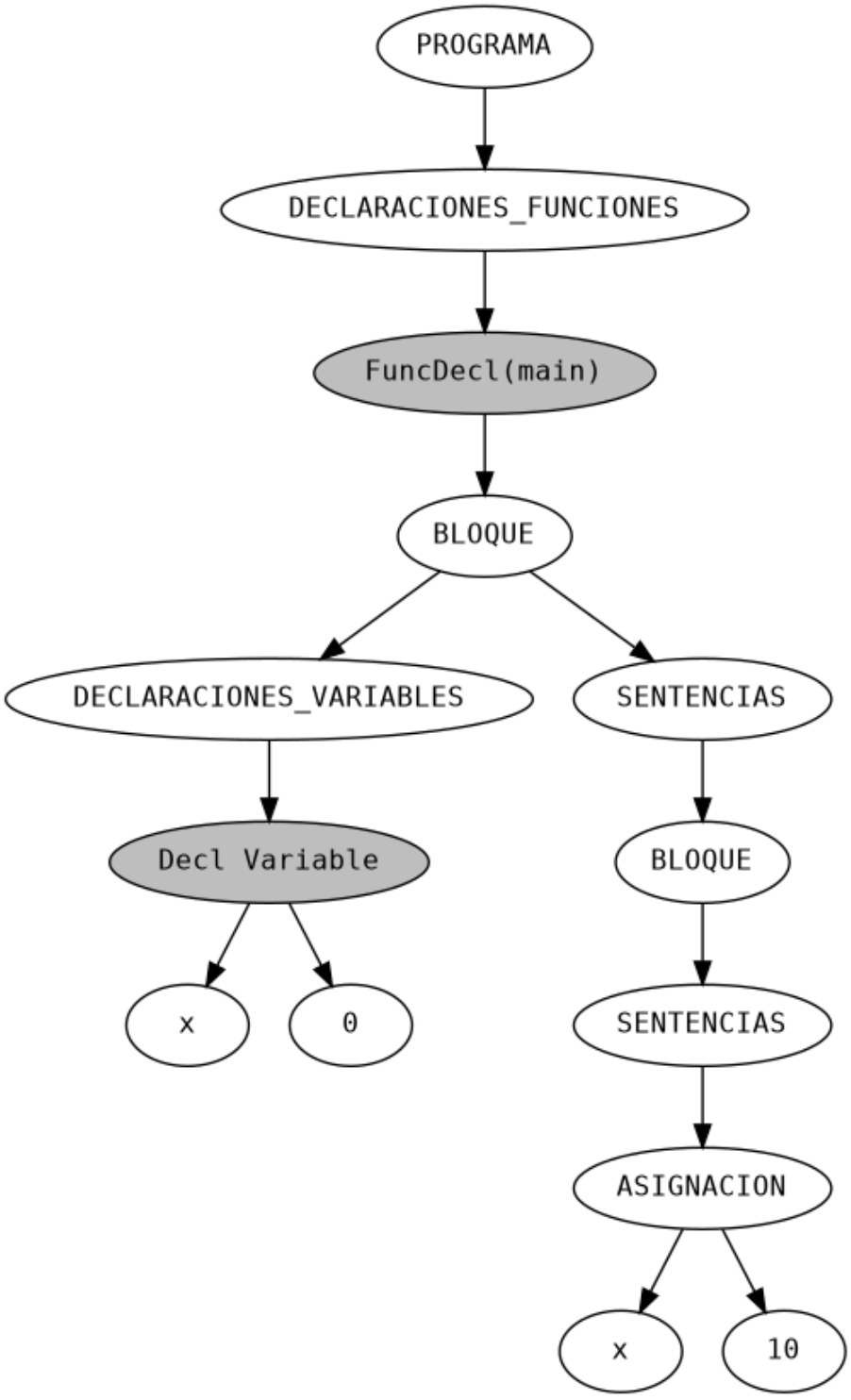
```
program
{
	void main()
	{
		integer x = 0;
		if (true) then
		{
			x = 10;
		}
		else
		{
			x = 20; // código muerto
		}
	}
}
```

En este caso, la condición del `if` es siempre verdadera, por lo que el compilador puede optimizar el código eliminando la rama `else` que nunca se ejecutará.

AST sin optimizar:



AST optimizado:



2.3 Reutilización de temporales

En la generación de código intermedio, cada operación aritmética o lógica que produce un resultado temporal se almacena en una variable temporal. Estas variables no corresponden a variables del programa original, sino a espacios de memoria temporales que existen únicamente durante la evaluación de expresiones.
Si cada operación crea un nuevo temporal sin límite, se consumiría innecesariamente más memoria.

Funcionamiento

Para optimizar esto, se implementa la reutilización de temporales libres:

1. Cuando un temporal deja de ser necesario (porque su valor ya fue usado en la siguiente operación o se asignó a una variable real), se marca como libre.

2. La próxima vez que se requiera un temporal, se recupera uno libre en lugar de crear uno nuevo.

Beneficios de esta optimización:

- Menor cantidad de memoria ocupada durante la ejecución del código intermedio.

- Menor número de registros o espacio en la pila necesarios para almacenar resultados intermedios.

Ejemplo:

```
program
{
      void main()
      {
            integer x = 1;

            integer y = 2;

            x = (x + 1) + (y + 2);
      }
}
```

El código intermedio generado sería:

```
main:
START_FUN main
MOV 1 x
MOV 2 y
ADD x 1 t0      // t0 = x + 1
ADD y 2 t1      // t1 = y + 2
ADD t0 t1 t1    // t1 se reutiliza para el resultado final
MOV t1 x
END_FUN
```

En este ejemplo, el temporal t1 se reutiliza para almacenar el resultado final de la expresión (x + 1) + (y + 2), evitando la creación de un temporal adicional y reduciendo así el uso de memoria.

3. Pruebas y verificación

Se desarrollaron nuevos conjuntos de prueba específicos para la etapa de optimización. Los tests se encuentran en la carpeta `test/optimizacion` y abarcan código muerto y propagación de constantes.

4. Ejecución del compilador

El archivo principal del proyecto (`main.c`) se reutiliza de la entrega anterior, incorporando modificaciones tanto en la estructura general del flujo de ejecución como en las funcionalidades implementadas.
En esta versión, se integra la etapa de optimización, la cual se ejecuta únicamente cuando el programa no presenta errores semánticos. De este modo, se asegura que las transformaciones aplicadas sobre el código intermedio se realicen sobre una representación válida y coherente del programa fuente.

En comparación con la entrega anterior, en la que se generaban múltiples archivos intermedios (`.lex`, `.sint`, `.sem`, `.ci`, `.s`), esta versión del compilador simplifica significativamente la gestión de salidas.
Actualmente, solo se conservan los archivos correspondientes a las etapas finales:

- `.ci`: código intermedio (generado únicamente en modo de depuración).
- `.s`: código ensamblador.

Además, se incorpora un nuevo sistema de advertencias (warnings). Estas advertencias informan al usuario sobre posibles situaciones problemáticas que no constituyen errores semánticos, como variables o funciones declaradas pero no utilizadas, entre otros casos. Las advertencias no detienen el proceso de compilación, pero proporcionan información útil para mejorar la calidad del código fuente.

Finalmente, se implementó la liberación completa del Árbol de Sintaxis Abstracta (AST) al finalizar la compilación.

Uso del programa (actualización):

- Se debe incluir el flag `-opt` para activar la ejecución de las optimizaciones.
- Se añadió el flag `-debug` para generar el AST y el código intermedio del programa.