

Segundo Proyecto

Objetivos

General

- Permitir conocer a profundidad cómo se comportan los lenguajes a bajo nivel.

Específicos

- Practicar las fases de síntesis de un compilador con el uso de lenguajes de alto nivel.
- Diseñar y construir un compilador que, a partir de un lenguaje de alto nivel, genere código de 3 direcciones.

Descripción general

El proyecto consistirá en un juego en el cual pelearán dos o más (hasta un máximo de ocho) robots. El juego tendrá un mapa en 2 dimensiones, con un tamaño definido por el usuario al momento de iniciar el juego, el diseño del tablero queda a discreción del estudiante. Los robots iniciarán en posiciones aleatorias del mapa. La batalla se desarrollará por turnos, cada turno se moverá y atacará solamente un robot, de manera que cuando se mueve el robot 1 es un turno, cuando se mueve el robot 2 es otro turno y así sucesivamente.

Los robots se moverán y dispararán contra los otros robots; cada robot contará con una barra de vida, al terminarse su vida el robot estará fuera del juego. Los robots tendrán también una barra de energía, cada vez que disparen deberán decidir cuanta energía gastarán, mientras más energía utilicen, más daño le ocasionarán al robot enemigo. La barra de energía se recuperará un poco cada turno. El diseño gráfico de los robots queda a criterio del estudiante con la salvedad de que se debe entender que es un robot. Cuando solamente quede un robot con vida el juego terminará y este se declarará el ganador, mostrando un mensaje al usuario indicando que robot fue el ganador.

Durante las batallas no se ingresará ningún comando por parte del usuario, sino que los robots tendrán definido un conjunto de reglas para responder a lo que vaya pasando en la batalla en un archivo de control. Esto incluirá, por ejemplo, cuando el robot choque con una pared, o cuando detecte un robot enemigo. Este archivo de control será definido por el usuario antes de la batalla en sí en un lenguaje de programación que será traducido a código de tres direcciones y optimizado para mejorar el rendimiento del juego.

Para ejecutar el juego se siguen los siguientes pasos:

1. El usuario selecciona los parámetros del juego (número de robots, tamaño del mapa, etc.).
2. El usuario elige que robots van a pelear, escogiendo sus archivos de control en la interfaz.
3. El usuario escoge la opción para iniciar la batalla.
4. El programa muestra gráficamente el combate.
5. El programa se detiene cuando solo queda un robot con vida y este es declarado el ganador.

El proyecto consiste entonces de dos partes: el juego y el editor.

El funcionamiento general de la aplicación será el siguiente:

Entradas: Archivos con código (especificado más adelante).

Procesos: Se seguirán los siguientes pasos:

1. El usuario ingresará cierto código.
2. El programa compilará el código.
3. El usuario seleccionará varios archivos, uno para cada robot, ya compilados para sus robots.
4. El juego tomará estos archivos y ejecutará lo compilado de uno en uno. En este paso es donde se da la batalla en sí.
5. Se repetirá el paso cuatro hasta que el sistema determine que un robot es el ganador.

Salida: Un robot como ganador.

Pueden existir dos tipos de usuarios, aquellos que solamente utilizan robots precompilados y aquellos que escriben sus propios robots.

Estos pasos aplican solamente a quienes escriben sus propios robots:

1. El usuario escribe cierto código Java y presiona compilar.
2. El programa compila el código y retorna un archivo con extensión 3d.

1. Motor del juego

Consta de: un editor de texto, la plataforma del juego en sí, y un menú de creación del juego

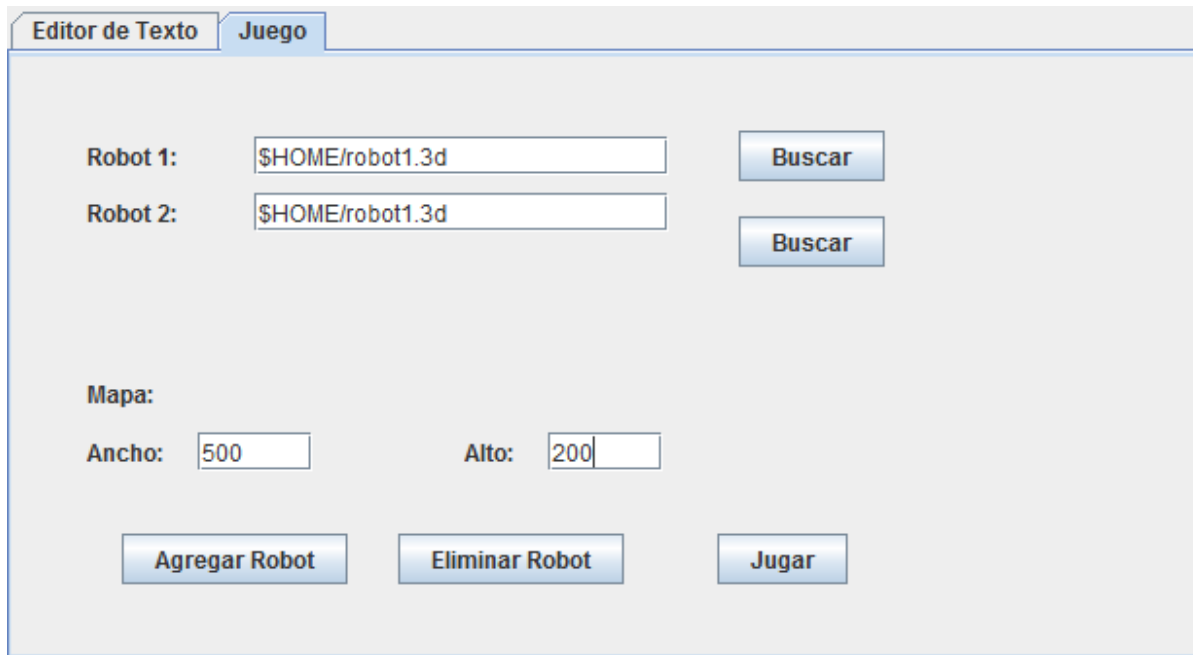
Editor de Texto

Se debe contar con un editor de código, el cual será capaz de mostrar los errores detectados en el código en tiempo real y contará con las siguientes opciones:

- Abrir: abre un archivo de código ya creado para su modificación.
- Guardar: guarda el archivo actual.
- Guardar como: guarda el archivo actual con el nombre especificado.
- Compilar: crea un archivo con el código de tres direcciones correspondiente al archivo actual. NO CORRE NADA. El archivo de tres direcciones se corre sobre el motor del juego. Si al momento de compilar se encuentran errores también se debe crear un archivo de errores como el especificado en la sección “Manejo de errores”

Menú de Creación de juego

Debe proporcionar una interfaz amigable al usuario, en donde se pueden escoger los robots a utilizarse en el juego, dichos robots pueden estar en cualquier lugar y deben ser archivos con extensión .3d. El diseño de la interfaz queda al criterio del estudiante, pero debe tener la opción de agregar varios robots al combate. Ver la figura para un ejemplo de interfaz simple.



The image shows a software interface for creating a game. It has two tabs at the top: 'Editor de Texto' and 'Juego', with 'Juego' being the active tab. The interface is divided into several sections. The first section is for adding robots, with labels 'Robot 1:' and 'Robot 2:'. Each label is followed by a text input field containing the path '\$HOME/robot1.3d'. To the right of each input field is a blue button labeled 'Buscar'. Below this is a section for map settings, labeled 'Mapa:'. It contains two input fields: 'Ancho:' with the value '500' and 'Alto:' with the value '200'. At the bottom of the interface are three blue buttons: 'Agregar Robot', 'Eliminar Robot', and 'Jugar'.

Figura: ejemplo de creación de juego.

Nota: Se puede escoger varias veces un mismo archivo fuente. En este caso se deben crear varias instancias del robot, cada una con sus propios atributos y métodos.

Plataforma de juego

El sistema deberá de proveer una plataforma que ejecutará un juego con vista top Down en el cual se enfrentarán distintos robots entre sí. El comportamiento de estos robots estará dado por archivos de código fuente el cuál será parseado a código de tres direcciones; este código de tres direcciones será posteriormente ejecutado por la plataforma cuando se ejecute un combate. Durante el juego en sí no se aceptará ningún comando del usuario hacia los robots, estos se deben mover de manera autónoma en base a las instrucciones indicadas en su archivo fuente.



Figura 1. Ejemplo de un combate

Mapa

El juego se desarrollará en un mapa rectangular, con el tamaño especificado en la creación del juego. Se debe tomar en cuenta que los robots no deben pasar del área delimitada como área de juego, por lo que se deben crear variables estáticas accesibles a todos los robots en base al tamaño del mapa.

Al iniciar el juego se creará cada robot en lugares aleatorios del mapa, sin embargo no se pueden traslapar los robots; en otras palabras varios robots no pueden tener la misma posición en el mapa. Esto se debe cumplir tanto para la inicialización del juego como para los movimientos posteriores de los robots.

El juego manejará un sistema interno de turnos, de manera que cada turno se ejecutará solamente las acciones que corresponden a un robot, al que le toque el turno. Cada turno también se ejecutará las acciones de las balas que existan en ese momento. La secuencia de los turnos puede ser configurada tanto automática, de manera que los turnos pasen de uno en uno automáticamente, o manual, de forma que se ejecute un turno y el sistema espere el comando del usuario para ejecutar el siguiente turno.

Robots

Cada robot deberá tener su propio archivo fuente, aunque pueden utilizarse varias instancias de un mismo robot en un combate. En una partida existirá un máximo de 8 robots y un mínimo de 2. El diseño gráfico de los robots queda a criterio del estudiante. Los robots tienen las siguientes propiedades intrínsecas:

Energía: Es la cantidad de energía con la que el robot cuenta para disparar, comienza con un valor de 100 unidades. Si cuando se intenta disparar se tiene una cantidad de energía menor a la necesaria, no se debe permitir el disparo. Se recupera una cantidad adecuada de energía cada tick del juego.

Vida: La vida restante del robot, esta vida se decrementará cada vez que un robot enemigo dispare sobre nuestro robot. El valor inicial de la vida es de 100 unidades.

Movimiento

Los robots pueden moverse cualquier cantidad de pixeles que deseen, tomando en consideración que no pueden salirse del mapa y no pueden terminar un turno encima de otro robot. Si el sistema detecta que un robot trata de terminar su turno encima de otro robot, debe mover al robot que está terminando su turno de manera aleatoria hasta termine en un lugar válido.

Un robot puede moverse, girar y atacar en un turno tantas veces como quiera, siempre y cuando se tome en cuenta que no puede disparar si no tiene la energía requerida por el disparo.

Balas

La función disparar creará un objeto bala que se moverá una cantidad predeterminada de pixeles cada turno (a discreción del estudiante, pero que no sea de manera instantáneo, es decir, no puede recorrer toda la pantalla en un turno).

Las balas se deben destruir si salen del área determinada como mapa.

Ambiente

El juego debe proporcionar ciertos métodos propios del mismo, los cuales podrán ser incluidos en los archivos de los robots. Estos métodos son:

VoidDisparar(int intensidad): crea un disparo de la intensidad indicada. Esta intensidad será decrementada de la energía del robot que dispara, y si el disparo impacta con cualquier enemigo, se le restará la intensidad del disparo a su vida.

VoidGirar(int grados): girará el robot la cantidad indicada de grados, puede ser un número positivo o negativo, girando en el sentido de las manecillas del reloj con un número positivo y en sentido contrario con un número negativo.

VoidRobotDetectado(): este es un método especial que será llamado si el robot tiene en su rango de ataque a otro robot, el rango de ataque será una línea recta entre el frente robot y el final del mapa (ver figura 2). Esta función será comprobada solamente al inicio del turno del robot.

VoidRobotGolpeado(): este método se llamará si el robot ha sido golpeado desde su turno anterior por una bala. Esta función será comprobada solamente al inicio del turno del robot.



Figura 2. La línea representa el rango de ataque, el método RobotDetectado no será llamado en este caso.

VoidChocar(): método especial que será llamado cuando el robot choque con alguna pared. Esta función será comprobada solamente al inicio del turno.

Esta función se comprobará cuando el robot este en movimiento, si se detecta el choque con una pared cuando se mueve, se interrumpirá la ejecución actual, y el programa deberá saltar a esta función y realizar su contenido, luego terminará su turno.

IntgetEnergia(): regresa el valor actual de la energía del robot.

Voidadelante(int x): mueve el robot x unidades hacia adelante.

Voidatras(int x): mueve el robot x unidades hacia atrás.

Voidfin(): termina el turno del robot sin ejecutar más comandos.

Voidimprimir(string s): imprime el string indicado en un espacio destinado para ello. Se sugiere tener un label de mensajes en la interfaz, que sea donde se escriban estos mensajes.

Cada turno el motor deberá evaluar si las condiciones de los métodos Chocar y RobotDetectado se cumplen, de ser así se debe ejecutar el código correspondiente.

El juego funcionará por turnos, de manera que cada turno se ejecutará el script de un robot, verificando si se deben ejecutar las funciones especiales y se moverán las balas.

El sistema también debe verificar cada turno si alguna bala impacta con un robot, de ser así es necesario substraer de la vida del robot la energía de la bala y eliminar la bala.

Si al final de un turno solamente queda un robot, éste será declarado el ganador.

Modos de ejecución

Existirán tres botones para controlar el juego: run, pause y step.

- Run ejecutará un turno tras otro, es decir, en el ejemplo anterior se ejecutará el turno 1, luego el turno 2 y luego el turno 3, sin ninguna intervención del usuario.
- Step solamente ejecutará el turno 1 (si se vuelve a presionar el botón step, se ejecutará el turno 2 y así sucesivamente).

- El botón pause detendrá la ejecución del run, la cual podrá volver a reanudarse desde el turno en que se detuvo con presionar nuevamente el botón run, o podrá utilizarse step para continuar turno por turno.

Ejemplo:

Robot 1

```
Main(){  
    Disparar();  
}
```

Robot2

```
Main(){  
    adelante(5);  
}
```

El resultado será:

Turno 1:

Robot 1 dispara.

Turno 2:

Robot 2 se mueve 5 unidades hacia enfrente.

Bala1 Se mueve.

Turno 3:

Robot 1 dispara.

Bala1 se mueve.

Nota: El cuerpo principal del robot debe ir en una función llamada Main(), esta función se considerará como ambiente global al momento de generar el código de tres direcciones.

2. Definición del lenguaje

Definición de funciones

Son un conjunto de instrucciones que pueden o no recibir parámetros, desde el exterior, y pueden retornar valores. Estas funciones utilizan variables locales donde su ámbito se limita al espacio de la función. Las funciones tendrán por default visibilidad public, a menos que el programador especifique otra visibilidad. Los parámetros deben tener definido un tipo.

La sintaxis general de una función va como sigue:

```
Visibilidad function nombre_funcion(parametros){  
<Instrucciones o variables locales>  
}
```

Importante: Los parámetros pueden ser pasados por valor o referencia. Si se van a pasar parámetros por referencia, se le antepone el símbolo "&". Dos o más funciones pueden llamarse igual siempre y cuando sus firmas sean diferentes (tipo de retorno, parámetros y nombre) dentro de su mismo ámbito.

Ejemplo:

Para una entrada de parámetros por valor:

```
Public function sumar(int x,int y){  
    return x+y;  
}
```

Para una entrada de parámetros por referencia:

```
Public function sumar(int x,int y, &z){  
    z=x+y;  
}
```

Las funciones pueden ser un miembro de una clase o bien pueden ser declarados a nivel global, ósea que puede no ser una parte de una clase.

Llamadas a funciones

Se debe detallar el nombre de la función y sus parámetros. Ejemplos de llamadas:

```
/*llamada de una función que devuelve valores*/  
var1=obtenerSuma(valor1,valor2);  
/*llamada a un procedimiento*/  
Calcular_disparo(10,2);
```

Instrucciones if, if-else:

La sintaxis general debe ir como sigue:

```
/*esta es una instrucción if*/
if(condición){
    [instrucciones o variables]
}

if(condicion)
    <una instruccion>
/*estas son instrucciones if-else*/
if(condicion){
    [instrucciones o variables]
}
else{
    [instrucciones o variables]
}

if(condicion)
    <una instruccion>
else
    <una instrucción>

/*esta es una instrucción de muchos if-else*/

if(condición_1){
    [instrucciones o variables]
}
else if(condición_2){
    [instruccines o variables1]
}
else if(condición_3){
    [instrucciones o variables2]
}
else if(condición_4){[
    instrucciones o variables3]
}
else{
    [instrucciones o variables4]
}
}
```

Importante: Las condiciones deben dar valores 1 o true para tomarse como verdadero, cualquier otro caso es falso. La instrucción no necesariamente puede venir dentro de una función, puede venir a nivel global. Pero tampoco puede ser tomado como miembro de una clase, por lo tanto si se encuentra dentro de una clase solo puede ir dentro de una función.

Instrucciones while:

La sintaxis general deberá ser como sigue:

```
while(condición){  
    [instrucciones o variables]  
}  
/*o bien puede venir como sigue*/  
while(condición)  
    <instrucción>
```

Se debe ejecutar el cuerpo del while mientras sea verdadera la condición, primero se evalúa la condición y luego se ejecuta el cuerpo.

Importante: Las condiciones deben dar valores 1 o true para tomarse como verdadero, cualquier otro caso es falso. La instrucción no necesariamente puede venir dentro de una función, puede venir a nivel global. Pero tampoco puede ser tomado como miembro de una clase, por lo tanto si se encuentra dentro de una clase solo puede ir dentro de una función.

Instrucciones do-while:

La sintaxis general deberá ser como sigue:

```
do{  
    [instrucciones o variables]  
}while(condición);
```

Se debe ejecutar el cuerpo del do-while mientras sea verdadera la condición. Primero se ejecuta el cuerpo y luego se evalúa la condición.

Importante: Las condiciones deben dar valores 1 o true para tomarse como verdadero, cualquier otro caso es falso. La instrucción no necesariamente puede venir dentro de una función, puede venir a nivel global. Pero tampoco puede ser tomado como miembro de una clase, por lo tanto si se encuentra dentro de una clase solo puede ir dentro de una función.

Instrucciones for:

La sintaxis general deberá ser como sigue:

```
for (asignaciones; condición; expresiones){  
    [instrucciones]  
}  
/*o bien puede venir como sigue*/  
for (asignaciones; condición; expresiones)  
    <instruccion>
```

El orden de ejecución irá como sigue: Se establecen las asignaciones codificadas, luego se evalúa la condición, a continuación se ejecuta el cuerpo del for y por último se ejecutan las expresiones que se codificaron en el for. El proceso que se acaba de detallar se debe repetir mientras la condición sea verdadera.

Importante: Las condiciones deben dar valores 1 o true para tomarse como verdadero, cualquier otro caso es falso. La instrucción no necesariamente puede venir dentro de una función, puede venir a nivel global. Pero tampoco puede ser tomado como miembro de una clase, por lo tanto si se encuentra dentro de una clase solo puede ir dentro de una función.

Instrucciones switch

La sintaxis general deberá ser como sigue:

```
switch (variable){  
    case literal_1:  
        [instrucciones]  
        break;  
    case literal_2:  
        [otras_instrucciones]  
        break;  
    .  
    .  
    .  
    case literal_n:  
        [otras_instrucciones]  
        break;  
    default:  
        [otras_instrucciones]  
}  
  
/*o bien*/  
switch (variable){  
    case literal_1:  
        [instrucciones]  
        break;
```

```
case literal_2:  
    [otras_instrucciones]  
    break;  
.  
.  
.  
Case literal_n:  
    [otras_instrucciones]  
    break;  
  
}
```

Se debe ir comparando la variable con cada literal de los case. Si coincide con alguno ejecuta el cuerpo del case y luego termina la instrucción.

Importante: Es obligatorio que tengan break al final. La instrucción no necesariamente puede venir dentro de una función, puede venir a nivel global. Pero tampoco puede ser tomado como miembro de una clase, por lo tanto si se encuentra dentro de una clase solo puede ir dentro de una función.

Instrucción return:

Debe llevar la siguiente sintaxis:

```
return [valor];
```

Esta instrucción permite terminar una función y asignarle como valor de retorno a la misma.

Importante: Es obligatorio que la instrucción se encuentre dentro de una función.

Definición de una clase

El lenguaje será totalmente orientado a objetos; por lo tanto hay que implementar el concepto de clases. La sintaxis general de una clase va como el siguiente ejemplo:

```
class nombre{
<declaraciones>
}
/*ejemplo*/
class nombre{
    public Int edad;
    String nombre;
    Float cantidad_dinero;
}
/*acceso a
estructuras*/
nombre variable1 = new nombre(); //al declarar de una vez crearle su espacio de
direcciones
variable1.nombre= "Juan Perez";
variable1.edad=30;
variable1.
cantidad_dinero=1500.0;
```

Herencia de una clase

Para permitir la reutilización de código, el lenguaje permitirá la herencia de una clase a otra. La sintaxis será la siguiente:

```
class A{
<declaraciones>
}
class B extends A{
<declaraciones>
}
```

En este caso la clase B tendrá acceso a todos los atributos y métodos de la clase A, con excepción de aquellos marcados como privados (ver tabla en la sección Modificadores de visibilidad).

Importación de una clase

Para poder utilizar una clase ya creada, se utilizará la instrucción import. Se debe limitar dónde se buscarán las clases para incluir, esto queda a criterio del estudiante, aunque la recomendación es que sea o en el directorio actual, o en un subdirectorío designado específicamente para las importaciones. La sintaxis es:

```
Import A;
```

Modificadores de visibilidad

La siguiente tabla muestra los accesos dependiendo que tipo de modificador de visibilidad tenga la función o variable.

Visibilidad	Interior de la clase	Herencia	Exterior de la clase
Public	X	X	X
private	X		
protected	X	X	

En caso de que no se declare la visibilidad de una variable, se asumirá que es public.

Variables y Tipos de datos

La sintaxis para la declaración de una variable será la siguiente:

<visibilidad> Int a <asignación>

O bien en el caso de una matriz

<visibilidad> int a[expresión] <asignación>

Tomar en cuenta que las matrices pueden ser de varias dimensiones, y pueden venir anidadas. Es decir que las matrices pueden definirse como sigue: int mat[4,[2,2]], int mat1[12,4,5], string mat2[a,4,[3,b,5], [2,3]].

Los tipos de datos válidos serán los utilizados como los maneja java conforme a su sintaxis.

- String
- Float
- Char
- Boolean
- Int

Comentarios:

Una línea: //

Varias líneas: /* comentario de varias

Líneas */

Carácter de escape

Para los símbolos que pueden confundir al compilador se utiliza el carácter“\” (código ASCII #134)

Tabla de operadores

Operador	Asociatividad	Precedencia
=,+=, -=, *=, %=	Derecha	1
	Izquierda	2
&&	Izquierda	3
==,!=	Izquierda	4
<,<=, >, >=	Izquierda	5
+, -	Izquierda	6
*, /, %	Izquierda	7
b++,c--	-	8
++b,--c	-	9
(casteo)variable	-	10
Literales, (expresion), variables,\$ this,null	-	11

Entre mayor sea su precedencia se ejecuta con más prioridad.

3. Estándar para código de tres direcciones

No se permite que se esté haciendo referencia a más de tres direcciones de memoria a la vez.

Operadores válidos

Nombre	Símbolo
suma	+
resta	-
multiplicación	*
División	/
modulo	%

Operadores que deben ir en una condición

Operación	Operador
If (var==var) goto Lx goto Ly	Comparación
If (var!=var) goto Lx goto Ly	Diferencias
If(var>var) goto Lx goto Ly	Mayor
If(var>=var) goto Lx goto Ly	Mayor o igual
If(var<var) goto Lx goto Ly	Menor
If(var<=var) goto Lx goto Ly	Menor o igual
If (1==1) goto Lx	True
If (1==0) goto Lx	False

Para los operadores || y &&, se debe hacer por medio de cortocircuito.

Temporales y variables

Para los temporales se nombrarán de la siguiente manera:

```
tnum  
ejemplo  
:  
t1,t2,t3.
```

Las variables globales (fuera de todo ámbito) pueden ser llamadas con el nombre que se definió en el lenguaje de alto nivel.

Importante: Para diferentes funciones siempre empezar desde “t1”.

Instrucciones

Para instrucciones sencillas, pueden venir los siguientes casos:

```
t1=t2 operador t3;  
p=p+1;  
m=m+t4;  
t1=4;  
t2=p+1;
```

Para instrucciones que deben acceder a un array:

```
t1=arr[t2];  
arr[t1]=t2;
```

Para el acceso a la pila donde se alojan las variables locales, temporales importantes y algunas referencias a memoria:

```
t1=stack[t2];  
stack[t1]=t2;
```

Para el acceso a la pila de memoria donde se alojan los objetos:

```
t1=heap[t2];  
heap[t1]=t2;
```

Uso de etiquetas

Para hacer un salto con etiquetas:

```
goto Lnum;
```

Para marcar donde inicia una etiqueta:

L_{num}:

Para marcar el final de un método se puede usar

L_{fin}

Declaración y uso de funciones

Para crear una función será de la siguiente manera:

```
public void funcion(){  
    .... Instrucciones 3 direcciones...  
}
```

Para hacer una llamada a una función:

función();

Pila y heap

Los nombres de la pila y el heap, así como de los punteros quedan a discreción del estudiante, teniendo en cuenta que se debe evitar que estos ocasionen conflictos con variables del lenguaje de alto nivel.

Ejemplo:

```
void calculadora__Mult(){  
    t1=p+0;  
    t2=stack[t1];  
    t3=t2+0;  
    t4=heap[t3];  
    t5=p+2;  
    stack[t5]=t4;  
    t6=p+0;  
    t7=stack[t6];  
    t8=t7+1;  
    t9=heap[t8];  
    t10=p+2;  
    t11=stack[t10];  
    t12=t11*t9;  
    t13=p+1;  
    stack[t13]=t12;  
}  
void calculadora__factorialRecursivo(int){  
    t1=p+1;  
    t2=stack[t1];  
    t3=p+3;  
    stack[t3]=t2;  
    t4=p+3;
```

```
t5=stack[t4];
if(t5==0)
goto L43;
goto L44;
L43:
t7=p+2;
stack[t7]=1;
goto Lfin;
L44:
t8=p+1;
t9=stack[t8];
t10=p+4;
stack[t10]=t9;
t11=p+1;
t12=stack[t11];
t13=p+5;
stack[t13]=t12;
t14=p+5;
t15=stack[t14];
t16=t15-1;
t17=p+6;
stack[t17]=t16;
t18=p+8;
t19=p+6;
t20=stack[t19];
stack[t18]=t20;
t21=p+0;
t22=stack[t21];
p=p+7;
t23=p+0;
stack[t23]=t22;
calculadora__factorialRecursivoint();
t24=p+2;
t25=stack[t24];
p=p-7;
t26=p+4;
t27=stack[t26];
t28=t27*t25;
t29=p+2;
stack[t29]=t28;
goto Lfin;
}
```

Para los archivos que contengan código de 3 direcciones, deben llevar el siguiente nombre:

Archivo.3d

A continuación definiremos las reglas que deben tomarse en cuenta para optimizar código.

Optimización de código

Debe optimizarse el código de tres direcciones por mirilla y luego por bloques.

Mirilla

Vamos a definir una mirilla de tamaño 12 en donde se van a contemplar las siguientes reglas:

1. Simplificación algebraica:

```
/*para esta entrada*/  
t1=p+0  
t2=stack[t1]  
  
/*Se debe dar esta salida*/  
t2=stack[p]  
  
/*para esta entrada*/  
t1=p*1  
t2=t1+10  
/*Se debe dar esta salida*/  
t2=p+10
```

2. Etiquetas con goto consecutivos:

```
/*Para esta entrada*/  
If(t1==10)goto L1  
goto L2  
L2:  
<líneas de código>  
L1:  
<líneas de código>  
  
/*Se debe dar esta salida*/  
If(t1==10)goto L1  
<líneas de código>  
L1:  
<líneas de código>
```

3. Flujos de ejecución

```
if(1==1)goto L1;  
goto L2;
```

```
L1:  
<código1>
```

```
L2:  
<código2>
```

/*debe transformarse en*/

```
<código1y código2>
```

4. Código inalcanzable

```
if(1==0)goto L1;  
goto L2;
```

```
L1:  
<código1>
```

```
L2:  
<código2>
```

/*debe transformarse en*/

```
<código2>
```

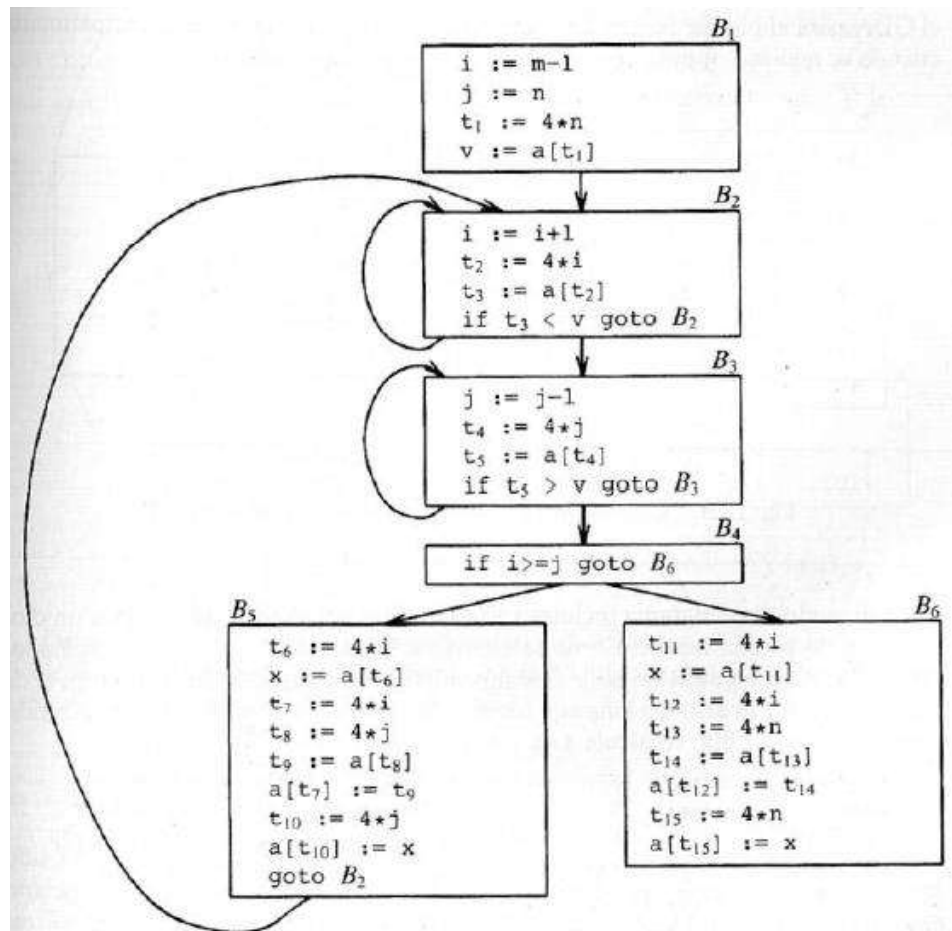
Bloques

Se contemplarán los casos que se describen a continuación, para validar que se está realizando la optimización por bloques será necesario que se realice el grafo:

Dado el ejemplo, veremos los diferentes casos de optimización por bloques.

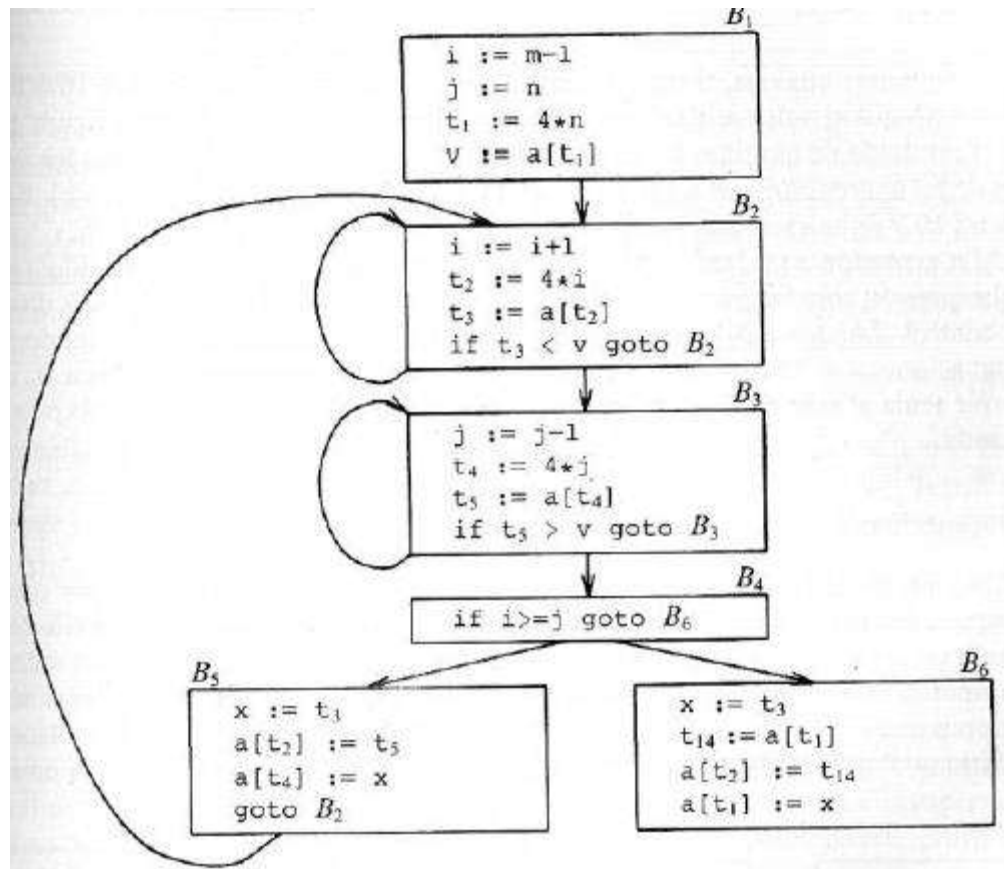
```
(1)  i := m-1
(2)  j := n
(3)  t1 := 4*n
(4)  v := a[t1]
(5)  i := i+1
(6)  t2 := 4*i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
(9)  j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

Vamos a dividir el código en bloques:



Sub expresiones comunes

Una ocurrencia de una expresión E como una sub expresión si E ha sido calculado previamente y los valores de las variables dentro de E no han cambiado desde el cálculo anterior. Se puede evitar recalcular la expresión si se puede utilizar el valor calculado anteriormente



En la figura se muestra el resultado de la eliminación de sub expresiones comunes. Más que todo afectó a los bloques 5 y 6.

1. Propagación de copias

Consiste en eliminar todas aquellas referencias que estén repetidas. Por ejemplo:

Ante instrucciones $f=a$, sustituir todos los usos de f por a

Antes	Después
$a = 3 + i$	$a = 3 + i$
$f = a$	$b = a + c$
$b = f + c$	$d = a + m$
$d = a + m$	$m = a + d$
$m = f + d$	

2. Eliminación de código inactivo

Una variable está activa en un punto si su valor puede ser utilizado posteriormente. De lo contrario se puede decir que es una variable inactiva. E no tras palabras se busca eliminar todas las proposiciones que calculan un valor y que nunca son utilizadas.

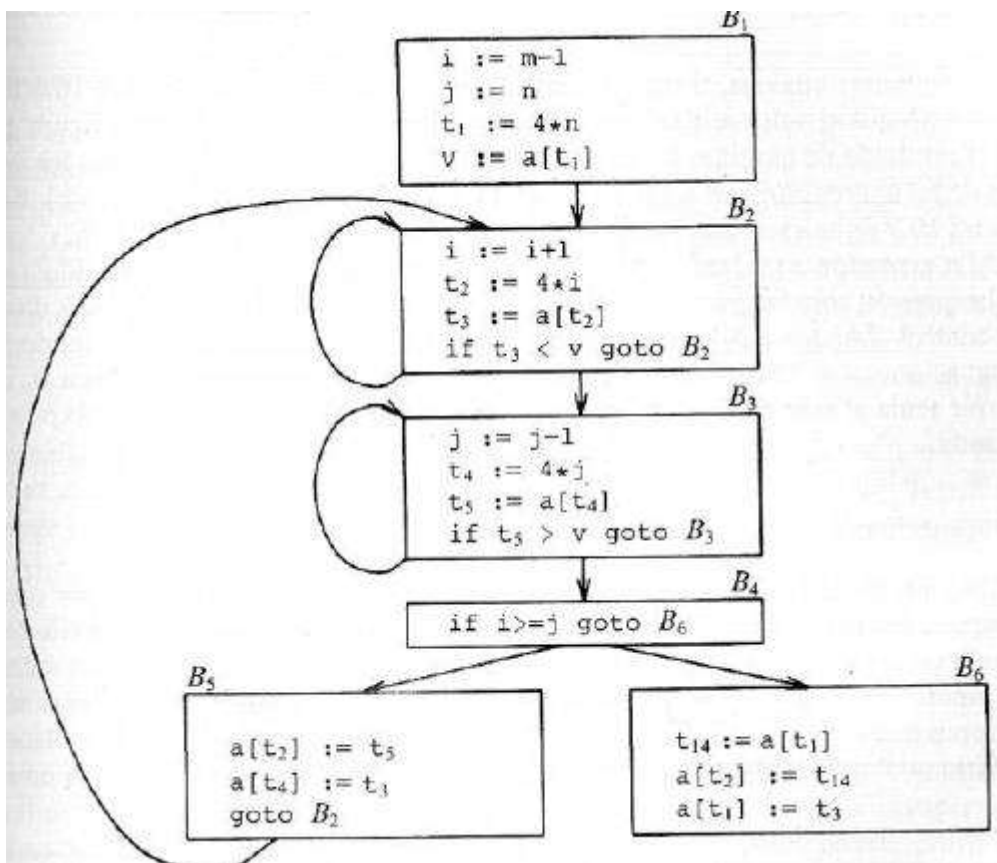
Por ejemplo:

t1=p+10;

t2=p+1;

stack[t2]=10;

Lo que se encuentra resaltado, se calculó pero nunca se utilizó.



Bloque después de eliminar propagación de copias y sub expresiones comunes.

Manejo de errores

Se debe mostrar un reporte de los errores que se encuentren durante las fases de compilación del proyecto. Los errores deben tener su respectivo detalle como por ejemplo:

Línea	Posición	Tipo	Descripción
1	10	Léxico	No se encuentra el símbolo@
45	4	Sintáctico	Se esperaba“}”
90	1	Semántico	La variable ‘a’ ya fue declarada.

Consideraciones

- El código de 3 direcciones, que se genere, debe ser almacenado en archivos para verificar que estén traduciendo y ejecutándolos.
- Cuando se dé un error en el lenguaje ya sea (léxico, sintáctico o semántico), el compilador debe detenerse y dejar de analizar.

Lenguajes a utilizar

- El lenguaje de programación será java.
- Las herramientas de generación de compiladores será JLex o JFlex y Cup.

Datos importantes

- La calificación del proyecto es personal y deberá realizarse el día acordado para su respectiva entrega y calificación.
- La aplicación de conocimientos correspondientes a cursos de semestres anteriores o el actual, es indispensable para realizar una parte o la totalidad de la práctica o los proyectos. Si existe alguna duda al respecto, se pueden dirigir a los auxiliares a quienes les pueden preguntar acerca de sus dudas.
- Copias en cualquiera de las actividades teóricas o prácticas tendrán una nota de cero puntos y serán reportados al catedrático titular de su sección y a la Escuela De Ciencias y Sistemas para su respectiva sanción.
- Se deben de utilizar los lenguajes y herramientas indicados, en caso contrario se tendrá una nota de cero puntos.
- Copias de gramáticas y código, totales o parciales tienen cero y serán reportados al catedrático de su sección y a la Escuela de Ciencias y Sistemas.
- Código bajado de internet tiene cero.
- Gramática bajada de internet tiene cero.

Entregables

1. Gramáticas con cada uno de los lenguajes explicados en este enunciado (lenguaje de los robots y gramática para correr código de tres direcciones).
2. Juego funcional.
4. Código fuente del proyecto.
5. Manual Técnico.

Fecha de entrega

19 de noviembre del 2011 **No hay prórroga**. Se entrega todo en un disco compacto y la entrega será presencial. El lugar y hora será acordado con su auxiliar.