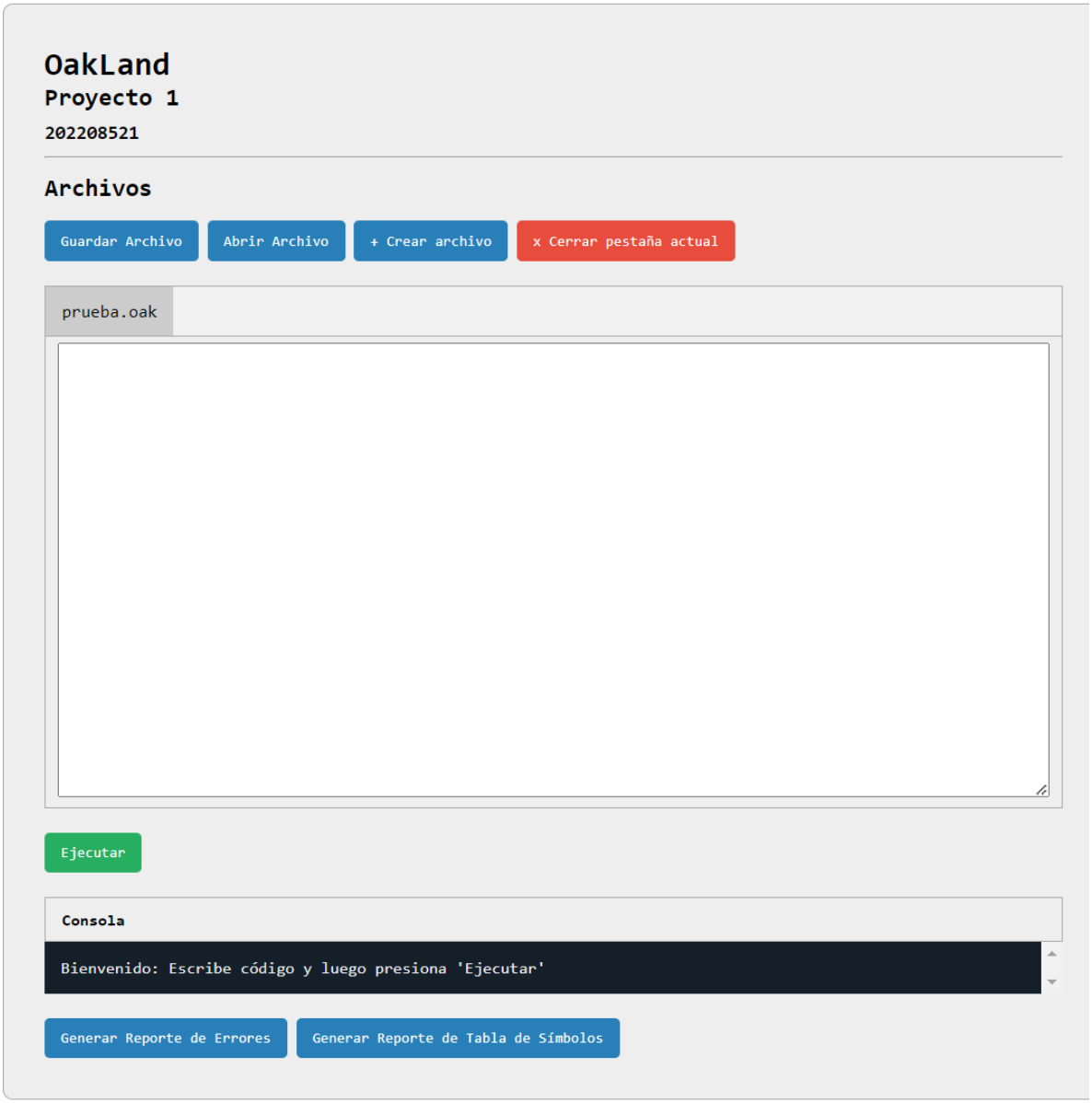
Manual de Usuario

Interfaz de Usuario



Esta es con la que el usuario se encontrará. Podemos ver distintos componentes, los cuales se describen a continuación:

- Guardar Archivo: Sirve para guardar el archivo que está abierto
- Abrir Archivo: Abre un archivo en el computador del usuario
- Crear un archivo: Crea una pestaña con nombre definido por el usuario

- Cerrar pestaña actual: Cierra la pestaña que se encuentre abierta
- Espacio para escribir: Es el espacio donde el usuario escribirá el código que quiere que se ejecute
- Boton ejecutar: va a ejecutar lo que esté en el espacio para escribir que esté abierto
- Consola: Mostrará el resultado de la ejecución, ya sea exitoso o con errores
- Generar Reporte de Errores: Genera un html con los errores reportados
- Generar Reporte de tabla de símbolos: Genera un html con una tabla de los símbolos que se generó al ejecutar el código otorgado

Descripción del Lenguaje

A continuación, se describe el lenguaje Oakland:

Identificadores

Un identificador será utilizado para dar un nombre a variables y métodos. Un identificador
está compuesto básicamente por una combinación de letras, dígitos, o guión bajo.
Ejemplos de identificadores válidos:
IdValido
id_Valido
i1d_valido5
_value
Ejemplo de identificadores no válidos
8
&idNoValido
.5ID
true
Tot@l
1d
Consideraciones:
● El identificador puede iniciar con una letra o un guión bajo _
● Por simplicidad el identificador no puede contener carácteres especiales (.$,-, etc)
● El identificador no puede comenzar con un número.

3.4. Case Sensitive

El lenguaje OakLand es case sensitive, esto quiere decir que diferenciará entre mayúsculas
con minúsculas, por ejemplo, el identificador variable hace referencia a una variable
específica y el identificador Variable hace referencia a otra variable. Las palabras

reservadas también son case sensitive por ejemplo la palabra **if** no será la misma que **IF**.

3.5. Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el
análisis sintáctico de la entrada. Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con el símbolo de **//** y al
final como un carácter de finalización de línea.
- Los comentarios con múltiples líneas que empezarán con los símbolos **/*** y terminarán con los símbolos ***/**

```
// Esto es un comentario de una línea
/* Esto es un comentario multilínea
*/
```

3.6. Tipos estáticos

El lenguaje OakLand no soportará múltiples asignaciones de diferentes tipos para una
variable, por lo tanto si una variable es asignada con un tipo, solo será posible asignar
valores de ese tipo a lo largo de la ejecución, sí alguna variable se le asignase un valor que
no corresponde a su tipo declarado, el programa debe mostrar un mensaje detallando el
error.

3.7. Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos:

Tipo primitivo	Definición	Rango (teórico)	Valor por defecto
int	Acepta valores números enteros	-2,147,483,648 a +2,147,483,647	0
float	Acepta valores numéricos de punto flotante	1.2E-38 a 3.4E+38 (6 dígitos de precisión)	0.0
string	Acepta cadenas de caracteres	[0, 65535]	

caracteres (acotado
por conveniencia)
“”
boolean Acepta valores
lógicos de verdadero
y falso
true false true
char Acepta un solo
caracter ASCII
[0, 65535] caracteres ‘’
Consideraciones:
● Por conveniencia y facilidad de desarrollo, el tipo String será tomado como un tipo
primitivo.
● Cuando se haga referencia a *tipos numéricos* se estarán considerando los tipos **int**
y **float**
● Cualquier otro tipo de dato que no sea primitivo tomará el valor por defecto de **null** al
no asignarle un valor en su declaración.
● Cuando se declare una **variable** y no se defina su valor, automáticamente tomará el
valor por defecto del tipo **null** esto para evitar la lectura de basura durante la ejecución.
● El literal 0 se considera tanto de tipo **int** como **float**.
● Las literales de tipo **char** deben de ser definidas con comilla simple (‘ ‘) mientras que
las literales de **string** deben ser definidas com comilla doble (“ ”)

3.8. Tipos Compuestos

Cuando hablamos de tipos compuestos nos vamos a referir a ellos como no primitivos, en
estos tipos vamos a encontrar las estructuras básicas del lenguaje OakLand.
● Array
● Struct
Estos tipos especiales se explicarán más adelante.

3.9. Valor nulo (**null**)

En el lenguaje OakLand se utiliza la palabra reservada **null** para hacer referencia a la
nada, esto indicará la ausencia de valor, por lo tanto cualquier operación sobre **null** será
considerada un **error y dará null** como resultado.

3.10. Secuencias de escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de
cadenas de texto. Las secuencias de escape disponibles son las siguientes
Secuencia Definición
¥" Comilla Doble
¥¥ Barra invertida
¥n Salto de línea
¥r Retorno de carro
¥t Tabulación

4. Sintaxis del lenguaje OakLand

A continuación, se define la sIntaxis para las sentencias del lenguaje OakLand

4.1. Bloques de Sentencias

Será un conjunto de sentencias delimitado por llaves "{ }", cuando se haga referencia a esto
querrá decir que se está definiendo un ámbito local con todo y sus posibles instrucciones,
además tendrá acceso a las variables del ámbito global.
11
Las variables declaradas en dicho ámbito únicamente podrán ser utilizadas en este ámbito o
en posibles ámbitos anidados. Cada instrucción del lenguaje OakLand deberá terminar con
el delimitador **punto y coma** (;).

```
// <sentencias de control>
{
// sentencias
}
int i = 10 // variable global es accesible desde este ámbito
if( i == 10 )
{
int j = 10 + i; // i es accesible desde este ámbito
if ( i == 10 )
{
int k = j + 1; // i y j son accesibles desde este ámbito
}
j = k; // error k ya no es accesible en este ámbito
}
```

Consideraciones:

● Estos bloques estarán asociados a alguna sentencia de control de flujo por lo tanto
no podrán ser declarados de forma independiente.

4.2. Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de
agrupación. Para los signos de agrupación se utilizarán los paréntesis ()

```
3 - (1 + 3) * 32 / 90 //1.5
```

4.3. Variables

Una variable es un elemento de datos cuyo valor puede cambiar durante el curso de la ejecución de un programa **siempre y cuando sea el mismo tipo de dato**. Una variable cuenta con un nombre y un valor, los nombres de variables no pueden coincidir con una palabra reservada.

Para poder utilizar una variable se tiene que definir previamente, la declaración nos permite crear una variable y asignarle un valor o sin valor.

Además la definición de tipo durante la declaración puede ser implícita o explícita, es explícita cuando se indica el tipo del cual será la variable e implícita cuando esta toma el tipo del valor al cual se está asignando.

Sintaxis:

```
// declaración con tipo y valor
<Tipo> <identificador> = <Expresión> ;
// declaración con tipo y sin valor
<Tipo> <identificador> ;
// declaración infiriendo el tipo
var <identificador = <Expresión> ;
```

Consideraciones:

- Las variables solo pueden tener **un tipo de dato** definido y este no podrá cambiar a lo largo de la ejecución.
- Solo se puede declarar **una** variable por sentencia.
- Si la variable ya existe se debe actualizar su valor por el nuevo, validando que el nuevo valor sea del mismo tipo del de la variable.
- El nombre de la variable **no puede ser una palabra reservada** ó del de una variable previamente definida.
- El lenguaje al ser case sensitive distinguirá a las variables declaradas como por ejemplo id y Id se toman como dos variables diferentes.
- Si la expresión tiene un tipo de dato **diferente** al definido previamente se tomará como **error** y la variable obtendrá el valor de **null** para fines prácticos.
- Cuando se asigna un valor de tipo ***int*** a una variable de tipo ***float*** el valor será considerado como ***float***, esta es la única conversión implícita que habrá.

● Al definir una variable con **var** es obligatorio inicializar el valor, de manera que pueda
inferirse correctamente el tipo, caso contrario será tomado como error.
Ejemplos:

```
// declaración de variables
// correcto, declaración sin valor
int valor;
// correcto, declaración de una variable tipo int con valor
var valor1 = 10;
```

13

```
// incorrecto, no se inicializó el valor de la variable
var noinicializada;
// Error: no se puede asignar un float a un int
int tipoincorrecto = 10.01;
float valor2 = 10.2; // correcto
float valor2_1 = 10 + 1; // correcto será 11.0
var valor3 = "esto es una variable"; //correcto variable tipo string
char caracter = 'A'; //correcto variable tipo char
bool valor4 = true; //correcto
//debe ser un error ya que los tipos no son compatibles
string valor4 = true;
// debe ser un error ya que existe otra variable valor3 definida
previamente
var valor3 = 10;
// es posible declarar nuevamente una variable siempre que esta
// declaración se haga en un nuevo bloque
{
var valor3 = "redefiniendo variable con un tipo distinto"
}
int .58 = 4; // debe ser error porque .58 no es un nombre válido
string if = "10"; // debe ser un error porque "if" es una palabra
reservada
// ejemplo de asignaciones
valor1 = 200; // correcto
valor3 = "otra cadena"; //correcto
valor4 = 10; //error tipos incompatibles (bool, int)
valor2 = 200; // correcto conversión implícita (float, int)
```

14

```
caracter = "otra cadena"; //error tipos incompatibles (char, string)
```

4.4. Operadores Aritméticos

Los operadores aritméticos toman valores numéricos de expresiones y retornan un valor
numé rico **único** de un determinado **tipo**. Los operadores aritméticos estándar son adición
o suma +, sustracción o resta -, multiplicación *, y división /, adicionalmente vamos a
trabajar el módulo %.

4.4.1. Suma

La operación suma se produce mediante la suma de tipos numéricos o Strings concatenados, debido a que OakLand está pensado para ofrecer una mayor versatilidad
ofrecerá conversión de tipos de forma implícita como especifica la siguiente tabla:

Operandos	Tipo resultante	Ejemplo
int + int	**int**	**1 + 1 = 2**
int + float	**float**	**1 + 1.0 = 2.0**
float + float	**float**	**1.0 + 13.0 = 14.0**
float + int	**float**	**1.0 + 1 = 2.0**
string + string	**string**	**"ho" + "la" = "hola"**

Consideraciones:

● Cualquier otra combinación será inválida y se deberá reportar el error.

4.4.2. Resta

La resta se produce cuando existe una sustracción entre tipos numéricos, de igual manera
que con otras operaciones habrá conversión de tipos implícita en algunos casos

Operandos	Tipo resultante	Ejemplo
int - int	**int**	**1 - 1 = 0**
int - float	**float**	**1 - 1.0 = 0.0**
float - float	**float**	**1.0 - 13.0 = -12.0**
float - int	**float**	**1.0 - 1 = 0.0**

4.4.3. Multiplicación

La multiplicación se produce cuando existe un producto entre tipos numéricos, de igual
manera que con otras operaciones habrá conversión de tipos implícita en algunos casos.

Operandos	Tipo resultante
int * int	**int**
int * float	**float**

float
1 * 10 = 10
1 * 1.0 = 1.0
float * float
float * int
float
float
1.0 * 13.0 = 13.0
1.0 * 1 = 1.0
Consideraciones:
● Cualquier otra combinación será inválida y se deberá reportar el error.

4.4.4. División

La división produce el cociente entre tipos numéricos, de igual manera que con otras
operaciones habrá conversión de tipos implícita en algunos casos a su vez truncamiento
cuando sea necesario.

Operandos Tipo resultante Ejemplo
int / int
int / float
int
float
10 / 3 = 3
1 / 3.0 = 0.3333
float / float
float / int
float
float
13.0 / 13.0 = 1.0
1.0 / 1 = 1.0
Consideraciones:
● Cualquier otra combinación será inválida y se deberá reportar el error.
● Se debe verificar que **no haya división por 0**, de lo contrario se debe mostrar una
advertencia y por conveniencia el resultado será un valor **null**.

4.4.5. Módulo

El módulo produce el residuo entre la división entre tipos numéricos de tipo **int**.

Operandos Tipo resultante Ejemplo
int % int int 10 % 3 = 1
Consideraciones:
● Cualquier otra combinación será inválida y se deberá reportar el error.
● Se debe verificar que **no haya división por 0**, de lo contrario se debe mostrar una
advertencia y por conveniencia el resultado será un valor **null**.

4.4.6. Operador de asignación

4.4.6.1. Suma

16
El operador += indica el incremento del valor de una **expresión** en una **variable** de tipo ya
sea **int** o de tipo **float**. El operador += será como una suma implícita de la forma:
variable = variable + expresión Por lo tanto tendrá las validaciones y restricciones
de una suma.
Ejemplos:
int *var1 = 10;*
float *var2 = 0.0;*
var1 += 10; //var1 tendrá el valor de 20
var1 += 10.0; // error, no puede asignar un valor de tipo float a un int
var2 += 10: // var2 tendrá el valor de 10.0
var2 += 10.0; //var tendrá el valor de 20.0
string *str =* ***"cad";***
str += ***"cad";*** *//str tendrá el valor de "cadcad"*
str += 10; //operación inválida string + int

4.4.6.2. Resta

El operador -= indica el decremento del valor de una **expresión** en una **variable** de tipo ya
sea **int** o de tipo **float**. El operador -= será como una resta implícita de la forma:
variable = variable - expresión Por lo tanto tendrá las validaciones y restricciones
de una resta.
Ejemplos:
int *var1 = 10;*
float *var2 = 0.0;*
var1 -= 10; //var1 tendrá el valor de 0
var1 -= 10.0; // error, no puede asignar un valor de tipo float a un int
var2 -= 10; // var2 tendrá el valor de -10.0
var2 -= 10.0; //var tendrá el valor de -20.0
17

4.4.7. Negación unaria

El operador de negación unaria precede su operando y lo niega (*-1) esta negación se
aplica a tipos numéricos

Operandos Tipo resultante Ejemplo
-int int -(-(10)) = 10
-float float -(1.0) = -1.0

4.5. Operaciones de comparación

Compara sus operandos y devuelve un valor lógico en función de si la comparación es
verdadera (**true**) o falsa (**false**). Los operandos pueden ser numéricos, Strings o lógicos,
permitiendo únicamente la comparación de expresiones del mismo tipo.

4.5.1. Igualdad y desigualdad

● **El operador de igualdad** (==) devuelve **true** si ambos operandos tienen el mismo valor, en caso contrario, devuelve **false**.

● **El operador no igual a** (!=) devuelve **true** si los operandos no tienen el mismo valor, de lo contrario, devuelve **false**.

Operandos Tipo resultante Ejemplo

int [==,!=] int boolean 1 == 1 = true
1 != 1 = false
float [==,!=] float boolean 13.0 == 13.0 = true
0.001 != 0.001 = false
int [==,!=] float
float [==,!=] int
boolean 35 == 35.0 = true
98.0 = 98 = true
boolean [==,!=] boolean boolean true == false = false
false != true = true
string [==,!=] string boolean "ho" == "Ha" = false
"Ho" != "Ho" = false
char [==,!=] char boolean 'h' == 'a' = false
'H' != 'H' = false

Consideraciones

● Cualquier otra combinación será inválida y se deberá reportar el error.

● Las comparaciones entre cadenas se hacen lexicográficamente (carácter por carácter).

4.5.2. Relacionales

Las operaciones relacionales que soporta el lenguaje OakLand son las siguientes:

● **Mayor que**: (>) Devuelve **true** si el operando de la izquierda es mayor que el operando de la derecha.

● **Mayor o igual que**: (>=)Devuelve true si el operando de la izquierda es mayor o
igual que el operando de la derecha.

● **Menor que**: (<) Devuelve true si el operando de la izquierda es menor que el operando de la derecha.

● **Menor o igual que**: (<=) Devuelve true si el operando de la izquierda es menor o
igual que el operando de la derecha.

Operandos Tipo
resultante
Ejemplo

int[>,<,>=,<=] int boolean 1 < 1 = false
float [>,<,>=,<=] float boolean 13.0 >= 13.0 = true
int [>,<,>=,<=] float boolean 65 >= 70.7 = false
float [>,<,>=,<=] int boolean 40.6 >= 30 = true
char [>,<,>=,<=] char boolean 'a' <= 'b' = true

Consideraciones

● Cualquier otra combinación será inválida y se deberá reportar el error.

● La comparación de valores tipos char se realiza comparando su valor ASCII.
● La limitación de las operaciones también se aplica a comparación de literales.

4.6. Operadores Lógicos

Los operadores lógicos comprueban la veracidad de alguna condición. Al igual que los operadores de comparación, devuelven el tipo de dato **boolean** con el valor **true** ó **false**.

● **Operador and** (&&) devuelve **true** si ambas expresiones de tipo **boolean** son **true**, en caso contrario devuelve **false**.
● **Operador or** (||) devuelve **true** si alguna de las expresiones de tipo **boolean** es **true**, en caso contrario devuelve **false**.
● **Operador not** (!) Invierte el valor de cualquier expresión booleaneana.

A	B	A && A	A || B	! A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Consideraciones:

● Ambos operadores deben ser booleaneanos, si no se debe reportar el error.

4.7. Precedencia y asociatividad de operadores

La precedencia de los operadores indica el orden en que se realizan las distintas operaciones del lenguaje. Cuando dos operadores tengan la misma precedencia, se utilizará la asociatividad para decidir qué operación realizar primero.
A continuación, se presenta la precedencia en orden de mayor a menor de operadores lógicos, aritméticos y de comparación .

4.8. Operador Ternario

El operador ternario es un operador que hace uso de 3 operandos para simplificar la instrucción ‘if’ por lo que a menudo este operador se le considera como un atajo para la instrucción ‘if’. El primer operando del operador ternario corresponde a la condición que debe de cumplir una expresión para que el operador retorna como valor el resultado de la expresión segundo operando del operador y en caso de no cumplir con la expresión el operador debe de retornar el valor de la expresión del tercer operando del operador.

Operador Asociatividad
( ) [ ] izquierda a derecha
! - derecha a izquierda
/ % * izquierda a derecha
+ - izquierda a derecha
< <= >= > izquierda a derecha
== != izquierda a derecha
&& izquierda a derecha
|| izquierda a derecha
<CONDICIÓN> ? <EXPRESIÓN> : <EXPRESIÓN> ;
Ejemplo:
```
int edad = 10;
boolean es_mayor = edad > 18 ? true : false;
```

4.9. Sentencias de control de flujo

Las estructuras de control permiten regular el flujo de la ejecución del programa. Este flujo de ejecución se puede controlar mediante sentencias condicionales que realicen ramificaciones e iteraciones. Se debe considerar que estas sentencias se encontrarán únicamente dentro funciones.

4.9.1. Sentencia If Else

Ejecuta un bloque de sentencias si una condición especificada es evaluada como verdadera. Si la condición es evaluada como falsa, otro bloque de sentencias puede ser ejecutado.

SIntaxis:

Ejemplo:

```
if( 3 < 4 ){
// Sentencias
} else if( 2 < 5 ){
// Sentencias
} else {
// Sentencias
}
if (true) { // Sentencias }
if (false) { // Sentencias } else { // Sentencias }
if (false){ // Sentencias } else if (true) { // Sentencias }
```

Consideraciones:

- Puede venir cualquier cantidad de if de forma anidada
- La expresión debe devolver un valor tipo **boolean** en caso contrario debe tomarse como error y reportarlo.

4.9.2. Sentencia Switch - Case

Evalúa una expresión, comparando el valor de esa expresión con un case, y ejecuta
declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.
Si ocurre una coincidencia, el programa ejecuta las declaraciones asociadas correspondientes. Si la expresión coincide con múltiples entradas, la primera será la
seleccionada. Si no se encuentra una cláusula de case coincidente, el programa busca la
cláusula **default** opcional, y si se encuentra, transfiere el control a ese bloque, ejecutando
las declaraciones asociadas. Si no se encuentra un default el programa continúa la
ejecución en la instrucción siguiente al final del switch. Por convención, el default es la
última cláusula.
Sintaxis:

```
switch (<Expresión>) {
case expr1:
// Declaraciones ejecutadas cuando el resultado de la
//expresión coincide con el expr1
break;
case expr2:
// Declaraciones ejecutadas cuando el resultado de la
//expresión coincide con el expr2
break;
// ...
case exprN:
// Declaraciones ejecutadas cuando el resultado de la
//expresión coincide con el exprN
break;
// [OPCIONAL]
default:
// Declaraciones ejecutadas cuando ninguno de
// los valores coincide con el valor de la expresión
}
```

Ejemplo:

```
int numero = 2;
switch (numero) {
case 1:
22
System.out.println("Uno");
break;
case 2:
System.out.println("Dos");
break;
case 3:
System.out.println("Tres");
```

```
break;
default:
System.out.println("Invalid day");
}
```

Consideraciones:

● No se implementará un break implícito. Esto significa que, si no se coloca un break explícito al final de cada caso, se ejecutará los siguientes case de forma secuencial.

4.9.3. Sentencia While

Crea un bucle que ejecuta un bloque de sentencias especificadas mientras cierta condición se evalúe como verdadera (**true**). Dicha condición es evaluada **antes** de ejecutar el bloque de sentencias y al final de cada iteración.

Sintaxis:

```
while (<Expresión) {
<BLOQUE SENTENCIAS>
}
```

Ejemplo:

```
while (true) {
//sentencias
}
int num = 10;
while (num != 0) {
num -= 1;
System.out.println(num);
}
/* Salida esperada:
9
8
7
23
6
5
4
3
2
1
0
*/
```

Consideraciones:

● El ciclo while recibirá una expresión de tipo **boolean**, en caso contrario deberá mostrar un error.

4.9.4. Sentencia For

Un bucle **for** en el lenguaje OakLand se comportará como un for moderno, que recorrerá
alguna estructura compuesta. La variable que recorre los valores se comportará como una
constante, por lo tanto no se podrán modificar su valor en el bloque de sentencias, su valor
únicamente cambiará con respecto a las iteraciones.

Ejemplo

```
// for que recorre un rango
for (int i = 1; i <= 5; i++) {
System.out.println(i);
}
string[] letras = {"O", "L", "C", "2"};
// for que recorre un arreglo unidimensional (foreach)
for (string letra : letras) {
System.out.println(letra);
letra = "cadena"; //error no es posible asignar algo a letra
}
/*Salida esperada:
1
2
3
4
5
O
L
C
24
2
*/
```

Consideraciones:

● El tipo de la variable que recorre un arreglo será del mismo tipo de dato que contiene
el arreglo.

4.10. Sentencias de transferencia

Estas sentencias transferirán el control a otras partes del programa y se podrán utilizar en
entornos especializados.

4.10.1. Break

Esta sentencia termina el bucle actual ó sentencia switch y transfiere el control del programa
a la siguiente sentencia a la sentencia de terminación de estos elementos.

Ejemplo:

```
while (true) {
int i = 0;
break; //finaliza el bucle en este punto
}
```

Consideraciones:

● Si se encuentra un **break** fuera de un ciclo y/o sentencia switch se considerará como un error.

4.10.2. Continue

Esta sentencia termina la ejecución de las sentencias de la iteración actual (en un bucle) y continúa la ejecución con la próxima iteración.

Ejemplo:

```
while (3 < 4) {
continue
}
int i = 0;
int j = i;
while (i < 2) {
25
if j == 0{
i = 1;
i += 1
continue;
}
i += 1
}
// i posee el valor de 2 al finalizar el ciclo
```

Consideraciones:

● Si se encuentra un **continue** fuera de un ciclo se considerará como un error.

4.10.3. Return

Sentencia que finaliza la ejecución de la función actual, puede o no especificar un valor para ser devuelto a quien llama a la función.

```
int funcion1() {
return 1; // retorna un valor int
}
void funcion() {
return; // no retorna nada
}
```

5. Estructuras de datos

Las estructuras de datos en el lenguaje OakLand son los componentes que nos permiten almacenar un conjunto de valores agrupados de forma ordenada, las estructuras básicas que incluye el lenguaje son los **Array.**

5.1. Array

Los array son la estructura compuesta más básica del lenguaje OakLand, los tipos de array

que existen son con base a los tipos **primitivos y structs** del lenguaje. Su notación de
posiciones por convención comienza con 0.

5.1.1. Creación de array

Para crear vectores se utiliza la siguiente sIntaxis.
Sintaxis:
26

```
// Declaración con inicialización de valores
int[] numbers = {1, 2, 3, 4, 5};
// Declaración reservando una cantidad de elementos
int[] numbers = new int[5];
// numbers2 es una copia de numbers
int[] numbers2 = numbers;
```

Consideraciones:

- La lista de expresiones debe ser del mismo tipo que el tipo del array.
- El tamaño de un arreglo no puede ser negativo
- El tamaño del array no puede aumentar o disminuir a lo largo de la ejecución.
- Cuando la definición de un array sea otro array, *se hará una copia* del array dando
origen a otro nuevo array con los mismos datos del array.

5.1.2. Valores por defecto

Al inicializar un array especificando únicamente su tamaño, los espacios del array se
rellenan automáticamente con valores por defecto, según el tipo de datos del array. A
continuación, se detallan los valores por defecto para algunos tipos comunes:

Tipo primitivo	**Valor por defecto**
int	0
float	0.0
string	“” (string vacía)
boolean	false
char	¥u0000 (carácter nulo)
struct	null

27

5.1.3. Función indexOf(<Expresion>)

Retorna el índice de la primer coincidencia que encuentre, de lo contrario retornará -1

5.1.4. Join()

Une todos los elementos del array en un string, separado por comas **Ej**: [1,2,3] -> “1,2,3”

5.1.5. length

Este atributo indica la cantidad de elementos que posee el vector, dicha cantidad la
devuelve con un valor de tipo int

5.1.6. Acceso de elemento:

Los arreglos soportan la notación para la asignación, modificación y acceso de valores,
únicamente con los valores existentes en la posición dada, en caso que la posición no
exista deberá mostrar un mensaje de error y retorna el valor de null.

Ejemplo:

```
// arreglo con valores
int[] arr1 = {10,20,30,40,50};
// funcionamiento de indexOf
System.out.println(arr1.indexOf(20)); // output: 1
System.out.println(arr1.indexOf(100)); // output: -1
var cadena = arr1.join();
System.out.println(cadena); // output: “10,20,30,40,50”
System.out.println(arr1.length); // output: 5
//se realiza una copia completa de vector
int[] copiaVec = arr1;
//Acceso a un elemento
int val = arr1[3]; // val = 40
//asignación con []
vec1[1] = arr1[0]; // [10,10,40,50]
```


5.2. Matrices / Arrays multidimensionales

Las matrices en OakLand nos permiten almacenar datos de tipo primitivo o compuesto, la
diferencia principal entre el array y la matriz es que esta última organiza sus elementos en n
dimensiones y la manipulación de datos es con la notación [ ] *además que* ***su tamaño no puede cambiar en tiempo de ejecución.***

5.2.1. Creación de matrices

Las matrices en OakLand pueden ser de **2 a n dimensiones** pero solo de un tipo específico, además su tamaño será constante y será definido durante su declaración.

Consideraciones:

- La declaración del tamaño puede ser explícita o en base a se definición.
- Si la declaración es explícita pero su definición no es acorde a esta declaración se
debe marcar como un error. Por lo tanto se debe verificar que la cantidad de dimensiones sea acorde a la definida.
- La asignación y lectura valores se realizará con la notación [ ]
- Los índices de declaración comienzan a partir de 1
- Los índices de acceso comienzan a partir de 0
- Las matrices **no** van a cambiar su tamaño durante la ejecución.
- Si se hace un acceso con índices en fuera de rango se devuelve **null** y se debe notificar como un error.

● Si se declara una matriz con índices negativos o 0, será considerado un error
Ejemplo:

```
// Definición e inicialización directa
int[][] mtx1= { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
System.out.println(mtx1[0][0] ); //imprime 1
// Definición de una matriz reservando una cantidad de elementos
int[][] mtx2 = new int[3][3];
// asignación de valores
intMatrix[0][0] = 7;
intMatrix[0][1] = 6;
intMatrix[0][2] = 5;
System.out.println(mtx2[0][1]); //imprime 6
//error indices fuera de rango - error
mtx1[100][100] = 10;
```


6. Structs

El lenguaje OakLand tiene la capacidad de permitir al programador en crear sus propios
tipos compuestos personalizados, estos elementos se les denomina structs, los structs
permiten la creación de estructuras de datos y manipulación de información de una manera
más versátil. Estos están compuestos por *tipos primitivos* o por otros structs.
Un struct que contiene otro struct como una de sus propiedades no puede ser del mismo
tipo que el struct que lo contiene.
En el caso que un struct posea atributos de tipo struct, estos se manejan por medio de
referencia así como sus instancias en el flujo de ejecución. Si un struct es el tipo de retorno
o parámetro de una función, también se maneja por referencia.

6.1. Definición:

Consideraciones:

● Los structs **solo** pueden ser *declarados* en el ámbito global
● Los structs deben tener al menos un atributo.
● No se podrán agregar más atributos a un struct una vez ha sido definido.

Ejemplo:

```
struct Persona {
// atributos
}
```

6.2. Uso de atributos

El lenguaje OakLand permite la edición y acceso a atributos de los structs por medio del
operador '.', el cual nos permite acceder a los atributos ya sea para asignarles un valor ó

acceder al valor.
Ejemplo:
miInstancia.atributo = “Hola mundo”;

6.3. Funciones Especiales

El lenguaje OakLand ofrece una serie de funciones para los structs.
30

6.3.1. Funcion Object.Keys(<Expresion>)

Esta función permitirá obtener un array con los atributos del struct
Ejemplo:

```
// Definición de un struct
struct Mascota {
string nombre;
}
struct Persona {
string nombre;
int edad;
Mascota mascota;
};
// creación de una instancia
Persona persona1 = Persona{
nombre: “Pablo”,
edad:20,
mascota: Mascota{
nombre: “Firulais”
}
};
// acceso a un atributo
System.out.println(persona1.nombre);
// modificación de un atributo
persona1.nombre = “Juanito”;
System.out.println("Edad", persona1.edad)
// Atributos anidados
System.out.println(persona1.apellido)
// El atributo no existe, debe lanzar un error
System.out.println(persona1.apellido)
System.out.println(persona1.mascota.nombre)
// Object.Keys()
string llaves = Object.keys(persona1);
System.out.println(llaves) // [“nombre”, “edad”, “mascota”]
```

31

```
// uso de la funcion typeof
System.out.println(typeof persona1); // Persona
```

7. Funciones

En términos generales, una función es un "subprograma" que puede ser llamado por código
externo (o Interno en caso de recursión) a la función. Al igual que el programa en sí mismo,

una función se compone de una secuencia de declaraciones y sentencias, que conforman el
cuerpo de la función.
Se pueden pasar valores a una función y la función puede devolver un valor. Para devolver
un valor específico, una función debe tener una sentencia return, que especifique el valor a
devolver. Además la función debe tener especificado el tipo de valor que va a retornar.
En el lenguaje OakLand se manejan los atributos por valor, esto aplica para cualquier tipo
para cualquier tipo de dato excepto structs y arrays, estos últimos se manejan por
referencia. Las funciones al igual que las variables se identifican con un ID válido.

7.1. Declaración de funciones

Consideraciones:

- Las funciones y variables si pueden tener el mismo nombre.
- Las funciones pueden o no retornar un valor, este tipo de funciones se definen
especificando la palabra reservada “void” al inicio.
- El valor de retorno debe de ser del **mismo** tipo del tipo de retorno de la función.
- Las funciones pueden ser declaradas en cualquier ámbito, ya sea global o el ámbito
de un bloque.
- Las funciones solo pueden retornar un valor a la vez.
- No pueden existir funciones con el mismo nombre aunque tengan diferentes parámetros o diferente tipo de retorno.
- El nombre de la función no puede ser una palabra reservada.

Ejemplo:

```
// Ejemplo de función:
int func1(){
return 1;
}
string fn2(){
32
return "cadena";
}
void funcion(){
System.out.println("Hola");
System.out.println(" ");
System.out.println("Mundo");
}
// Función que retorna una nueva referencia del array
int[] obtenerNumeros() {
```

```
return [1, 2, 3, 4, 5];
}
// Función que devuelve una nueva referencia del struct
Producto obtenerProducto() {
var producto = Producto{ id: 1, nombre: “Producto A” };
return producto;
}
// error: ya se ha declarado una función llamada funcion previamente
string funcion(){
return "valor";
}
// error: nombre inválido
void if(){
System.out.println("Esto no debería darse");
}
// error: valor de retorno incompatible con el tipo de retorno
string valor(){
return 10;
}
// error: no se define un tipo de retorno
void invalida() {
return 1000;
}
```

7.1.1. Parámetros de funciones

Los parámetros en las funciones son variables que podemos utilizar mientras nos encontremos en el ámbito de la función.

Consideraciones:

- Los parámetros de tipo struct y array son pasados por referencia, mientras que el resto de tipos primitivos son pasados por valor.
- No pueden existir parámetros con el mismo nombre.
- Pueden existir funciones sin parámetros.
- Los parámetros deben tener indicado el tipo que poseen, en caso contrario será considerado un error.

Ejemplos:

```
// Función suma
int suma(int num1, int num2){
return num1 + num2;
}
// función que recibe un array por referencia
int obtenerNumArr(int[] arr, int index){
return arr[index];
}
void set(int[] arr, int index, int value){
arr[index] = value
}
```

```
// función suma
int suma(int x, int y){
return x + y;
}
//funcion resta
int resta(int x, int y){
return x - y;
}
//función mul
int mul(int x, int y){
return x * y;
}
// Función que devuelve un struct
Producto obtenerProducto(int id): {
var productoUno = { id: 1, nombre: “Producto A” };
var productoDos = { id: 2, nombre: “Producto B” };
if (productoUno.id == id) {
return productoUno;
34
} else {
return productoDos;
}
}
```

7.2. Llamada a funciones

Los parámetros en la llamada a una función son los argumentos de la función. Los argumentos se pasan a las funciones por **valor** o **referencia** según sea el caso. Las llamadas a funciones pueden ser una sentencia o una expresión.

Consideraciones:

- Si se realiza una llamada de una función sin retorno dentro de una expresión, se deberá marcar como un error.
- Se deben verificar que los parámetros sean del mismo tipo esperado que se definió en la declaración.
- Una llamada se puede realizar ya sea si la función fue declarada antes o después de la llamada

Ejemplos:

```
var numero1 = 1;
var numero2 = 1;
var arr = [1,2,3,4,5,6,7];
System.out.println(suma(numero1, numero2)); //imprime 2
System.out.println(resta(numero1, numero2)); //imprime 0
System.out.println(mul(numero1, numero2)); //imprime 1
// llamada por referencia
System.out.println(arr); // imprime [1,2,3,4,5,6,7]
set(arr, 0, 100);
set(arr, 1, 200);
```

```
System.out.println(arr); // imprime [100,200, 3,4,5,6,7]
//imprime Producto A
System.out.println(obtenerProducto(1).nombre);
```
35
```
//imprime Producto B
System.out.println(obtenerProducto(4).nombre);
```

7.3. Funciones Embebidas

El lenguaje OakLand está basado en Typescript y este a su vez es un superset de sentencias de Javascript, por lo que en OakLand contamos con algunas de las funciones
embebidas más utilizadas de este lenguaje.

7.3.1. System.out.println()

Esta función nos permitirá imprimir solamente tipos primitivos definidos en OakLand.

Consideraciones

- Puede venir cualquier cantidad de expresiones separadas por coma.
- Se debe de imprimir un salto de línea al final de toda la salida.

Ejemplo:

```
System.out.println("cadena1","cadena2") //mostraría: cadena1 cadena2
System.out.println("cadena1") // mostraría cadena1
System.out.println("cadena1 \n cadena2") // mostraría cadena1
// cadena2
System.out.println("valor", 10) // mostraría valor 10
System.out.println(true) // mostraría true
System.out.println(1.00001) //imprime 1.00001
```

7.3.2. parseInt()

Esta función permite convertir una expresión de tipo **string** en una expresión de tipo **int**.

Si la cadena que recibe como parámetro no se puede convertir a un valor numérico se debe
desplegar un mensaje de error. Si la **string** representa valor decimal, debe de redondearse
por truncamiento.

Ejemplo:

```
int w = parseInt("3"); // w obtiene el valor de 3
int x= parseInt("3.99999"); // x obtiene el valor de 3
int x1 = parseInt(10.999999) // error: tipo de dato incorrecto
int y = parseInt("Q10.00") // error no puede convertirse a int
```
36

7.3.3. parsefloat()

Esta función permite convertir una expresión de tipo **string** en un valor de tipo **float**. Si la
cadena que recibe como parámetro no se puede convertir a un valor numérico con punto
flotante se debe desplegar un mensaje de error.

```
float w = parsefloat("10") // w obtiene el valor de 10.00
float x = parsefloat("10.001") // x adopta el valor de 10.001
float y = parsefloat("Q10.00") // error no puede convertirse a float
```

7.3.4. toString()

Esta función es la contraparte de las dos anteriores, es decir, toma como parámetro un valor numérico y retorna una cadena de tipo **String.** Además sí recibe un valor **boolean** lo convierte en **"true"** o **"false"**. Para valores tipo **float** la cantidad de números después del punto decimal queda a discreción del estudiante.

```
var num1 = parseInt("1.99999");
var num2 = 23;
System.out.println(toString(num1) + toString(num2)); //imprime 123
System.out.println(toString(true)+"false"); //imprime truefalse
string cadena = toString(false) + "->" + toString(num1);
System.out.println(cadena); // imprime false->1
```

7.3.5. toLowerCase()

Esta función es aplicable a cualquier expresión de tipo **string** y su objetivo es convertir el texto reconocido en letras minúsculas.

```
string mayusculas = "HOLA MUNDO";
string minusculas = toLowerCase(mayusculas);
System.out.println(minusculas); // Imprime: hola mundo
int num = 10;
37
System.out.println(toLowerCase(num)); // Error: tipo de dato incorrecto
```

7.3.6. toUpperCase()

Esta función es la contraparte de la anterior, es aplicable a cualquier expresión de tipo **string** y su objetivo es convertir el texto reconocido en letras mayúsculas.

```
string minusculas = "hola mundo";
string mayusculas = toUpperCase(minusculas);
System.out.println(mayusculas); // Imprime: HOLA MUNDO
int num = 10;
System.out.println(toUpperCase(num)); // Error: tipo de dato incorrecto
```

7.3.7. typeof

Esta función retorna el tipo de dato asociado, este funcionará con tipos primitivos como lo son [string, int, float] y tipos compuestos como los structs.

```
int numero = 42;
string tipoNumero = typeof numero;
System.out.println(tipoNumero); // imprime "int"
boolean esVerdadero = true;
string tipobooleaneano = typeof esVerdadero;
System.out.println(tipobooleaneano); // imprime "boolean"
```

8. Reportes Generales

Como se indicaba al inicio, el lenguaje OakLand genera una serie de reportes sobre el
proceso de análisis de los archivos de entrada. Los reportes son los siguientes:
38

8.1. Reporte de errores

El Intérprete deberá ser capaz de detectar todos los errores que se encuentren durante el
proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un
reporte de errores en el que, como mínimo, debe mostrarse el tipo de error, su ubicación y
una breve descripción de por qué se produjo.

No
.
Descripción Línea Columna Tipo
1 El struct “Persona” no
fue definido.
5 1 semántico
2 No se puede dividir
entre cero.
19 6 semántico
3 El símbolo “¬” no es
aceptado en el
lenguaje.
55 2 léxico

8.2. Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán
de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como
su tipo y toda la información que el estudiante considere necesaria para demostrar que el