Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer semestre 2024
Catedráticos: Ing. Bayron López, Ing. Edgar Saban e Ing. Luis Espino
Tutores Académicos: Henry Mendoza, Daniel Santos y Damián Peña

OakLand

1. Competencias

1.1. Competencia General

Diseñar e implementar un compilador para el lenguaje de programación OAKLAND, que traduzca programas de alto nivel a código ensamblador x32 para la arquitectura RISC-V. Se evaluará la capacidad de los estudiantes para utilizar herramientas de desarrollo de compiladores y aplicar principios de diseño de software en la construcción de un sistema complejo.

1.2. Competencias Específicas

- Que el estudiante utilice PeggyJs como herramienta léxica y sintáctica para el reconocimiento y análisis del lenguaje.
- Que el estudiante implemente la traducción orientada por la sintaxis utilizando reglas semánticas haciendo uso de atributos sintetizados y/o heredados.
- Familiarizar al estudiante con las herramientas y estructuras de datos disponibles para la creación de un compilador, proporcionando una comprensión profunda de los procesos involucrados en la traducción y ejecución de programas escritos en OakLand en entornos basados en la arquitectura RISC-V.

2. Descripción

Se solicita que el estudiante desarrolle un compilador para el lenguaje de programación **OakLand**, este lenguaje está inspirado en la sintaxis de Java, sin embargo esta no es su principal característica. OakLand destaca por su capacidad para manejar múltiples paradigmas de programación, incluyendo la orientación a objetos, la programación funcional y la procedimental.

Además de esto, se necesita que se desarrolle una plataforma sencilla, pero lo suficientemente robusta para poder crear, abrir, editar y compilar el código **OakLand.** Este IDE deberá ser escrito en JavaScript vanilla, y será desplegado en Github Pages.

Los analizadores (Léxico y Sintáctico) deberán ser construidos con la herramienta **PeggyJS**, para ello se deberá escribir una gramática que describa la sintaxis del lenguaje.

Este compilador tiene una estructura particular, ya que no se enfocará en las fases tradicionales de generación de código de tres direcciones ni en la optimización correspondiente. En lugar de ello, se saltará directamente a la etapa de traducción a código ensamblador.

El proyecto requerirá que los estudiantes adquieran un entendimiento profundo de los principios fundamentales de la compilación, así como de los aspectos específicos de la traducción a código ensamblador para la arquitectura RISC-V. Los estudiantes explorarán de manera práctica cómo se realiza la traducción de un lenguaje de alto nivel a

instrucciones de bajo nivel en una arquitectura de conjunto de instrucciones reducido (RISC).

Para validar la traducción realizada, se emplea la herramienta en línea ripes.me para ejecutar el código ensamblador resultante. Esto permitirá verificar que el código fuente tenga una traducción equivalente en ensamblador y que el comportamiento del programa sea el esperado.

2.1. Componentes de la aplicación

2.1.1. Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades, características y herramientas que serán de utilidad al usuario. La función principal del editor será el ingreso del código fuente que será analizado.
En este se podrán abrir diferentes archivos al mismo tiempo y deberá mostrar la línea actual.

El editor de texto se tendrá que mostrar en la interfaz de usuario (GUI). Queda a discreción del estudiante el diseño.

2.1.2. Funcionalidades

Las funcionalidades básicas que se solicitan implementar, serán las siguientes:

Crear archivos: El editor deberá ser capaz de crear archivos en blanco.
Abrir archivos: El editor deberá abrir archivos **.oak**
Guardar archivo: El editor deberá guardar el estado del archivo en el que se estará trabajando con extensión **.oak**
Guardar traducción a Ensamblador: El usuario deberá de ser capaz de guardar la traducción generada en un archivo con extensión ".s" Ej: calculadora.s

2.1.3. Herramientas

Compilar: hará el llamado al compilador, el cual se hará cargo de realizar los análisis léxico, sintáctico y semántico. Además generará una traducción equivalente en lenguaje ensamblador para la arquitectura RISCV.

2.1.4. Reportes

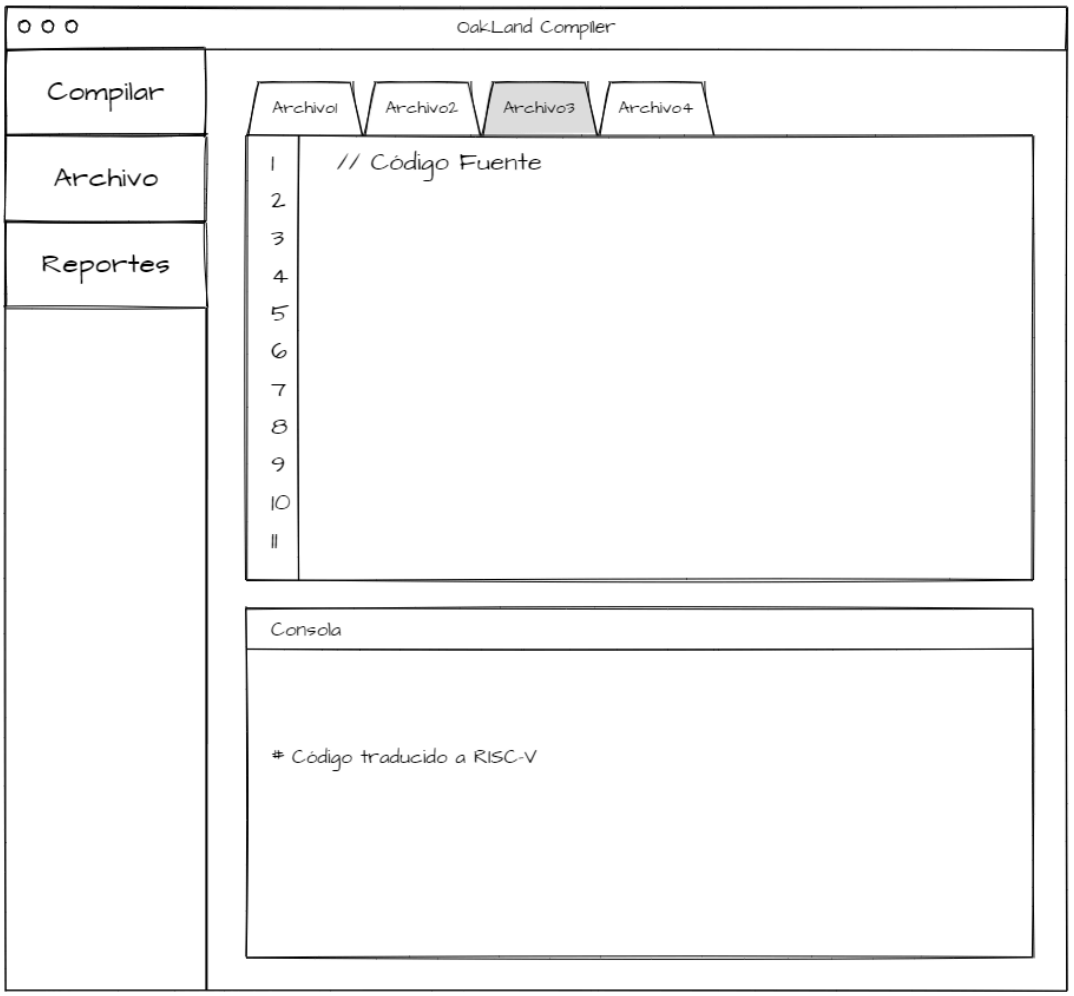
Reporte de Errores: Se mostrarán todos los errores encontrados al realizar el análisis léxico, sintáctico y semántico.

Reporte de Tabla de Símbolos: Se mostrarán todas las variables, métodos y funciones que han sido declarados dentro del flujo del programa, así como el entorno en el que fueron declarados.

2.1.5. Consola

La consola es un área especial dentro del IDE que permite visualizar las notificaciones, errores, advertencias e impresiones que se produjeron durante el proceso de análisis de un archivo de entrada.

Impresión de Traducción a Ensamblador:
Después de completar la traducción del código OakLand a código ensamblador para la arquitectura RISC-V, la traducción resultante se imprimirá en la consola para que el usuario pueda revisar y validar la salida.



(interfaz propuesta)

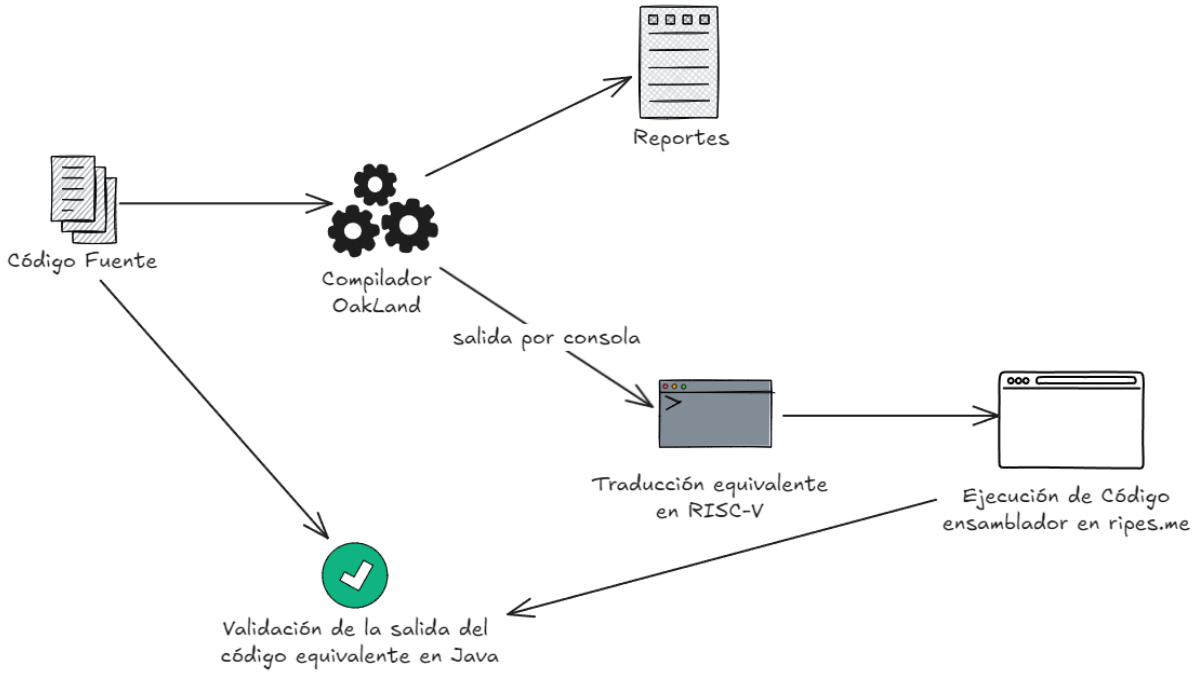
2.2. Flujo del proyecto

- **Inicio del Programa:**
    - El programa comienza con la carga o creación de un archivo que contiene el código fuente en OakLand.
- **Cargar/Crear Archivo:**
    - El usuario tiene la opción de cargar un archivo existente o crear uno nuevo que contenga el código en OakLand.
- **Compilación del Código:**

En el caso del compilador OakLand, se diseñará como un compilador de una pasada, significa que las fases para la construcción del compilador se realizan de manera secuencial, sin tener que volver a examinar el código fuente. Esto permite simplificar el diseño y la implementación del compilador. Se construye una tabla de símbolos que contiene información sobre las variables, funciones y otros elementos definidos en el código. De igual manera por simplicidad se omiten algunas fases del proceso y otras se reemplazan como el caso de generación de código de tres direcciones suplantado por la generación de código ensamblador, a continuación se describen las fases que llevarán a cabo el proceso de compilación:

1. Análisis Léxico: El compilador escanea el código fuente y lo divide en tokens (unidades léxicas como palabras clave, identificadores, operadores, etc.).
2. Análisis Sintáctico: Utiliza la gramática del lenguaje OakLand para verificar la estructura sintáctica del código y construir un árbol de análisis sintáctico (AST).
3. Análisis Semántico: Verifica que el código cumpla con las reglas semánticas del lenguaje, como el tipo correcto de las variables, la correcta aplicación de operadores, etc.
4. Generación de Código Assembler en RISC-V: Utilizando la información del AST y del análisis semántico, el compilador genera código en lenguaje ensamblador compatible con la arquitectura RISC-V.

- **Generación de Resultados:**
    - Se generan resultados y reportes que pueden incluir:
    - Información detallada sobre errores encontrados durante el análisis.
    - Contenido de la tabla de símbolos.
    - Resultados de la traducción en código ensamblador.
- **Fin del Programa:**
    - El programa finaliza, y el usuario puede revisar los resultados generados durante el proceso.
- **Validación de la Salida del Código Equivalente en Java:** Para garantizar una correcta traducción del código fuente en OakLand, se ejecutará directamente el código en Java de alto nivel, verificando que la salida obtenida coincida con la generada al ejecutar las instrucciones en ensamblador. Este proceso comparará los resultados de ambas ejecuciones, asegurando la precisión y consistencia del compilador en la generación y ejecución del código. Como herramienta estándar para la validación del código equivalente, se utilizará el simulador OnlineGDB disponible en el siguiente enlace: https://www.onlinegdb.com/online_java_compiler.



Flujo de ejecución del proyecto

Componentes clave:

- **Código fuente**: La entrada es código fuente en formato .oak (Más adelante se especifican los detalles del lenguaje).
- **Compilador**: El código fuente se procesa por el compilador OakLand, desarrollado por el estudiante. Como salida se obtienen los resultados en consola y los reportes.
- **Resultado de consola**: Se deberá implementar una consola (Como se expone en interfaz propuesta) donde se pueda visualizar las salidas, la traducción producida por el OakLand.
- **Reportes**: Se deberán generar reportes de Errores y Tabla de Símbolos. Estos se detallarán más adelante.
- **Salida**: Se tendrá la posibilidad de guardar la traducción resultante en una archivo con extensión ".s", este se utilizará para hacer la ejecución y validación de la traducción en simriscv.

2.3. Arquitectura propuestas

Para la implementación de la solución, se restringe el uso **exclusivamente a JavaScript del lado del cliente**. Esto significa que todo el código deberá ser generado de manera que pueda ser interpretado directamente por el navegador.

No se permitirá el uso de entornos de ejecución de JavaScript como Node.js, Deno o Bun. Todos los archivos generados deberán ser estáticos, debido a la forma en la que se requiere que sean desplegados (detallado más adelante en la sección de GitHub Pages).

No se permitirá el uso de frameworks, bibliotecas, librerías o empaquetadores de código. La solución debe ser desarrollada utilizando solo JavaScript "vanilla" o puro, sin depender de herramientas adicionales que simplifiquen el desarrollo o gestionen la construcción del código.

2.3.1. Aplicación Web

Dado que la implementación debe limitarse exclusivamente al uso de JavaScript del lado del cliente, el estudiante debe utilizar únicamente tecnologías de frontend. Esto incluye HTML, CSS y JavaScript. Toda la funcionalidad debe estar desarrollada para ejecutarse directamente en el navegador.

2.3.2. Github Pages

Para el despliegue de la aplicación, se utilizará **exclusivamente** GitHub Pages, un servicio proporcionado por GitHub que permite alojar sitios web directamente desde un repositorio de GitHub. Es importante tener en cuenta que GitHub Pages solo admite contenido estático, lo que significa que no puede ejecutar código del lado del servidor.

Por lo tanto, toda la lógica y funcionalidad de la aplicación deben ser implementadas utilizando JavaScript del lado del cliente. Esto implica que no se puede ejecutar código que requiera un entorno de ejecución de servidor, como Node.js, o servicios que necesiten procesamiento en el servidor.

3. Generalidades del lenguaje OakLand

El lenguaje OakLand está inspirado en la sintaxis del lenguaje Java, por lo tanto se conforma por un subconjunto de instrucciones de este, con la diferencia de que OakLand tendrá una sintaxis más reducida pero sin perder las funcionalidades que caracterizan al lenguaje original.

3.1. Expresiones en el lenguaje OakLand

Cuando se haga referencia a una 'expresión' a lo largo de este enunciado, se hará referencia a cualquier sentencia u operación que devuelve un valor. Por ejemplo:

- Una operación aritmética, comparativa o lógica
- Acceso a un variable
- Acceso a un elemento de una estructura
- Una llamada a una función

3.2. Compilación:

OakLand carece de una función **main**, por ello para el inicio de la compilación del programa, el compilador deberá de traducir las órdenes conforme estas sean declaradas en el archivo de entrada. Dicho comportamiento se detalla a lo largo de este enunciado.

3.3. 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

```
&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)
`int`	**Acepta valores números enteros**	**-2,147,483,648 a +2,147,483,647**
`float`	**Acepta valores numéricos de punto flotante**	**1.2E-38 a 3.4E+38 (6 dígitos de precisión)**
`string`	**Acepta cadenas de caracteres**	**[0, 65535] caracteres (acotado por conveniencia)**
boolean	**Acepta valores lógicos de verdadero y falso**	**true false**
`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

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.

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;

// 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

boolean 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)

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 + float`	`int` `float`	`1 + 1 = 2` `1 + 1.0 = 2.0`
`float + float` `float + int`	`float` `float`	`1.0 + 13.0 = 14.0` `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 - float`	`int` `float`	`1 - 1 = 0` `1 - 1.0 = 0.0`
`float - float` `float - int`	`float` `float`	`1.0 - 13.0 = -12.0` `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	Ejemplo
`int * int` `int * float`	`int` `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

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
```

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`**

Operandos	Tipo resultante	Ejemplo
`-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`

A	B	A && A	A || B	! A
`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 .

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

4.8. 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.

4.8.1. 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.

4.8.2. 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:
    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.

4.8.3. 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
6
5
4
3
2
1
0
*/
```

Consideraciones:

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

4.8.4. 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+=1) {
   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
2
*/
```

Consideraciones:

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

4.9. Sentencias de transferencia

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

4.9.1. 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.

4.9.2. 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) {
    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.

4.9.3. 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:

```
// 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

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}
```

6. 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.

6.1. 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 **exclusivamente** en el ámbito global.
- 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(){
  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() {
  int[] arr = {1, 2, 3, 4, 5};
  return arr;
}

// 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;
}
```

6.1.1. 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 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;
}
```

6.2. 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

set(arr, 0, 100);
set(arr, 1, 200);

// ‘arr’ se vería como {100,200, 3,4,5,6,7}
```

6.3. 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.

6.3.1. 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
```

6.3.2. 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
```

6.3.3. 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
```

6.3.4. 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
```

6.3.5. 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;
System.out.println(toLowerCase(num));  // Error: tipo de dato incorrecto
```

6.3.6. 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
```

6.3.7. 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"
```

7. Generación de Código Assembler

La generación de código assembler para la arquitectura RISC-V implica la traducción de las estructuras sintácticas y semánticas del lenguaje OakLand a instrucciones específicas del conjunto de instrucciones RISC-V. Durante este proceso, se mapean las construcciones del lenguaje, como declaraciones de variables, expresiones aritméticas, control de flujo y llamadas a funciones, a secuencias de instrucciones RISC-V que implementan la funcionalidad deseada.

Para la generación de código assembler para la arquitectura RISC-V, se optará por utilizar las instrucciones del conjunto RV32I, que representa una implementación base de 32 bits de RISC-V, ofreciendo un equilibrio óptimo entre simplicidad y funcionalidad. Este conjunto de instrucciones proporciona un conjunto fundamental de operaciones aritméticas, lógicas y de transferencia de datos que son esenciales para la ejecución de programas OakLand. Al seleccionar este conjunto de instrucciones, se garantiza la portabilidad y la compatibilidad del código generado en una amplia gama de dispositivos y sistemas que admiten la arquitectura RISC-V de 32 bits. Además, la elección de RV32I simplifica el proceso de desarrollo del compilador al reducir la complejidad asociada con características avanzadas presentes en otros conjuntos de instrucciones, permitiendo un enfoque más centrado en la generación de código eficiente y confiable para aplicaciones OakLand.

Para verificar la salida del compilador, se emplea el entorno Ripes, es un simulador visual de arquitectura informática y un editor de código ensamblador creado para la arquitectura del conjunto de instrucciones RISC-V que funciona en cualquier navegador web. Ripes proporciona un entorno interactivo para escribir, ejecutar y depurar programas en lenguaje

ensamblador RISC-V, asimismo brinda una interfaz gráfica con la capacidad de visualizar los registros en memoria, el funcionamiento del procesador, la interacción con la memoria caché . Utilizando esta plataforma, se podrá cargar el código assembler generado por el compilador para evaluar su correctitud y eficiencia en la arquitectura RISC-V. A través de las capacidades de Ripes, se podrán realizar pruebas exhaustivas y verificar el comportamiento del código generado en diferentes escenarios y condiciones.

Asimismo, se requerirá que el compilador produzca un archivo de salida con extensión ".s" que contenga el código assembler generado. Este archivo será utilizado para la validación dentro del simulador de RISC-V, lo que permitirá ejecutar el código generado en un entorno simulado y verificar su funcionamiento. La generación de este archivo facilitará la integración del compilador con otras herramientas de desarrollo y depuración, así como con flujos de trabajo de compilación más complejos que puedan requerir la interacción con herramientas externas. De esta manera, se garantizará que el código generado por el compilador sea compatible con las herramientas y entornos de desarrollo estándar para la arquitectura RISC-V.

- Para acceder al simulador utilizar el siguiente enlace: https://ripes.me/
- Para acceder a la documentación oficial de RISC-V utilizar el siguiente enlace: https://riscv-programming.org/book/riscv-book.html

7.1. Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis de la entrada. Se utilizará el formato del mismo RISC-V para comentarios:

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

```
# Este es un comentario en RISC-V
# La siguiente línea carga el valor 10 en el registro x10
 li x10, 10
/*  Esto  es
 un
 comentario
 multilínea*
 /
```

En este ejemplo, las líneas que comienzan con # son comentarios y no tienen ningún impacto en el código ensamblador. Los comentarios son útiles para explicar el código y hacer anotaciones.

7.2. Registros

En ensamblador RISC-V, los tipos de datos generalmente se manejan a nivel de registros y direcciones de memoria. No hay tipos de datos explícitos como en lenguajes de más alto nivel. Sin embargo, podemos considerar los registros como almacenando diferentes tipos de datos según su uso.

7.2.1. Registros de propósito general:

Estos registros se pueden utilizar para almacenar enteros, direcciones de memoria o cualquier otro tipo de datos que se pueda representar en un tamaño de registro.

```
# Cargar un entero en el registro x10
 li x10, 42

# Cargar una dirección de memoria en el registro x11
 la x11, mi_etiqueta
```

7.2.2. Registros de coma flotante:

Estos registros se utilizan para operaciones de punto flotante y pueden contener números de punto flotante en formatos como IEEE-754 de precisión simple o doble.

```
# Cargar un número de punto flotante de precisión simple
# en el registro f10
flw f10, etiqueta_fp

# Cargar un número de punto flotante de doble precisión
# en el registro f11
fld f11, etiqueta_fp_doble
```

7.2.3. Memoria:

La memoria se puede considerar como un tipo de dato, ya que se pueden almacenar diferentes valores en ella. Sin embargo, en ensamblador RISC-V, no hay un tipo de datos explícito para la memoria, sino que se trata más como un almacén de bytes direccionables.

```
# Almacenar un valor en la memoria
sw x10, 0(x11) # Almacena el valor de x10 en la dirección apuntada por
 x11

# Cargar un valor de la memoria en un registro
lw x12, 0(x11) # Carga el valor de la dirección apuntada por x11 en x12
```

7.3. Etiquetas

Las etiquetas serán identificadores únicos que indican una ubicación específica en el código fuente, permitiéndonos realizar saltos a lo largo en el código en la ejecución, estas se definirán durante el proceso de compilación, el formato de la etiqueta está definido de la siguiente manera:

```
# Ejemplos
L1:
# Código para inicializar un bucle #
    ...

L3:
# Código de bucle #
    ...

# Actualización del contador del bucle
addi x10, x10, 1  # Incrementa el contador en 1

# Comprobación de la condición de salida del bucle
bne x10, x11, bucle # Salta al inicio sí x10 no es igual a x11

L100:
# Código después del bucle
```

En el ejemplo anterior se presenta un bucle utilizando etiquetas.

7.4. Saltos

Los saltos son instrucciones que permiten cambiar el flujo de ejecución del programa. En RISC-V, los saltos condicionales e incondicionales se realizan mediante distintas instrucciones.

Instrucciones de salto condicional: Estas instrucciones se utilizan para realizar saltos condicionales basados en el estado de los registros.

```
beq x2, x3, L1 # Salta a la L1 si x2 es igual a x3
bne x4, x5, L2 # Salta a la L2 si x4 no es igual a x5
blt x6, x7, L3 # Salta a la L3 si x6 es menor que x7
```

Instrucciones de salto incondicional: Estas instrucciones siempre realizan un salto a la ubicación especificada, independientemente de alguna condición.

```
j L1 # Salta a la etiqueta L1
jal L2 # Salta a la etiqueta L2 y almacena la dirección de retorno
jr x9 # Salta a la dirección almacenada en el registro x9
```

7.5. Operadores

A continuación se detallan algunas de las operaciones aritméticas básicas que se pueden realizar en lenguaje ensamblador RISC-V. Dependiendo de la extensión específica y de las características del conjunto de instrucciones, puede haber otras operaciones aritméticas disponibles, pero las principales son las siguientes:

Suma:

```
add x3, x1, x2 # x3 = x1 + x2
```

Resta:

```
sub x3, x1, x2 # x3 = x1 - x2
```

Multiplicación:

```
mul x3, x1, x2 # x3 = x1 * x2
```

División:

```
div x3, x1, x2 # x3 = x1 / x2
```

Incremento y decremento:

```
addi x3, x3, 1 # Incrementa x3 en
subi x3, x3, 1 # Decrementa x3 en 1
```

7.6. 8.5. Ejemplo de Entrada y Salida

Entrada:

```
int a = 0;
int b = 7;
int result;

if (a > b) {
  result = a;
} else {
  result = b;
}

System.out.println(result);
```

Salida:

```
.text
.align 2
.globl _start

_start:
  # Asignar valores a y b
  li a0, 10   # a = 10
  li a1, 7    # b = 7

  # Comparar a y b
  bge a0, a1, greater_than    # if (a > b) jump to greater_than
  mv a0, a1                   # b to result
  j end                       # jump to end

greater_than:
  mv a0, a0                   # a to result

end:
  # Imprimir el resultado
  li a7, 1                   # syscall: imprimir entero
```

```
ecall

# Terminar el programa
li a7, 10                    # syscall: terminar programa
ecall

.data
.align 2
```

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:

8.1. Reporte de errores

El compilador 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
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 compilador tradujo correctamente el código de entrada.

ID	Tipo símbolo	Tipo dato	Ámbito	Línea	Columna
x	Variable	int	Global	2	5

Ackerman	Función	float	Global	5	1
vector1	Variable	Array	Ackerman	10	5

9. Entregables

El estudiante deberá entregar únicamente el link de un repositorio en GitHub, el cual será privado y contendrá todos los archivos necesarios para la ejecución de la aplicación, así como el código fuente, la gramática y la documentación. El estudiante es responsable de verificar el contenido de los entregables, los cuales son:

9.1. Código fuente de la aplicación
9.2. Gramáticas utilizadas
9.3. Manual Técnico
9.4. Manual de Usuario
9.5. Despliegue en GitHub Pages (url)

El nombre del repositorio debe seguir el siguiente formato: **OLC2_Proyecto2_#Carnet**. Ejemplo: OLC2_Proyecto2_202110568

Se deben conceder los permisos necesarios a los auxiliares para acceder a dicho repositorio. Los usuarios son los siguientes:

- damianpeaf
- Henry2311
- danielSG95

10. Restricciones

10.1. El proyecto deberá realizarse como una aplicación web utilizando el lenguaje **Javascript**.
10.2. La solución debe implementarse utilizando únicamente JavaScript del lado del cliente. No se permite el uso de entornos de ejecución, como Node.js, Deno, o Bun.
10.3. Todo el código debe ser estático y ejecutarse en el navegador sin dependencia de servidores o APIs externas. Los archivos generados deben ser HTML, CSS y JavaScript.
10.4. La funcionalidad de la aplicación web debe ser desplegada exclusivamente en **GitHub Pages**. Solo se calificará el uso de este servicio, y no se permitirá el uso de otros para la publicación.
10.5. No se permite el uso de frameworks, librerías de terceros, o empaquetadores. Solo se debe utilizar código vanilla o puro de JavaScript. No será calificado el código que haya sido sometido a un proceso de compilación previo.
10.6. Para el analizador léxico y sintáctico se debe implementar una gramática con la herramienta **PeggyJs**.

10.7. Las copias de proyectos tendrán de manera automática una nota de **0 puntos** y los involucrados serán reportados a la Escuela de Ciencias y Sistemas.
10.8. El desarrollo y la entrega del proyecto son de manera **individual**.

11. Consideraciones

11.1. Durante la calificación se realizarán preguntas sobre el código para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas ***los tutores harán una verificación exhaustiva en busca de copias*.**
11.2. Se necesita que el estudiante al momento de la calificación tenga el entorno de desarrollo y las herramientas necesarias para realizar pequeñas modificaciones en el código para verificar la autoría de este, en caso que el estudiante no pueda realizar dichas modificaciones en un tiempo prudencial, el estudiante tendrá 0 en la sección ponderada a dicha sección y ***los tutores harán una verificación exhaustiva en busca de copias*.**
11.3. Se tendrá un máximo de 30 minutos por estudiante para calificar el proyecto. La calificación será de manera virtual y se grabará para tener constancia o verificación posterior.
11.4. La hoja de calificación describe cada aspecto a calificar, por lo tanto, si la funcionalidad a calificar falla en la sección indicada se tendrá 0 puntos en esa funcionalidad y esa nota no podrá cambiar si dicha funcionalidad funciona en otra sección.
11.5. Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.
11.6. Los archivos de entrada permitidos en la calificación son únicamente los archivos preparados por los tutores.
11.7. Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.
11.8. La sintaxis descrita en este documento son con fines descriptivos, el estudiante es libre de diseñar la gramática que crea apropiada para el reconocimiento del lenguaje OakLand.

12. Entrega del proyecto

12.1. La entrega se realizará de manera virtual, se habilitará un apartado en la plataforma de UEDI para que el estudiante realice su entrega.
12.2. No se recibirán proyectos fuera de la fecha y hora estipulada.
12.3. La entrega de cada uno de los proyectos es **individual.**
12.4. Fecha límite de entrega del proyecto: **25 de octubre de 2024**