Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Segundo semestre 2020
Catedráticos: Ing. Edgar Saban, Ing. Bayron López, Ing. Erick Navarro e Ing. Luis Espino
Tutores académicos: Pavel Vásquez, Erik Flores, Juan Carlos Maeda, Cristian Alvarado

MatrioshTS

Contenido

1. Competencias

1.1. Competencia general

Que el estudiante aplique la fase de síntesis del compilador para realizar un traductor e intérprete utilizando herramientas.

1.2. Competencia especifica

- Que el estudiante utilice un generador de analizadores léxicos y sintácticos para construir un traductor.
- Que el estudiante aplique el análisis semántico para validar que el código de entrada es válido.
- Que el estudiante aplique los conocimientos aprendidos en clase para generar código intermedio.
- Que el estudiante aplique reglas de optimización por bloque y mirilla para reducir el código intermedio generado.

2. Descripción

MatrioshTS es un lenguaje de programación inspirado en Typescript, su característica principal es la inclusión de tipos explícitos.

El sistema de tipos de MatrioshTS realiza una formalización de los tipos de Javascript, mediante una representación estática de sus tipos dinámicos. Esto permite a los desarrolladores definir variables y funciones tipadas sin perder la esencia. Otra inclusión importante de MatrioshTS es la inclusión de funciones anidadas por lo que su traducción genera el mismo lenguaje solo que las funciones estarán desanidadas.

2.1. Componentes de la aplicación

Se requiere la implementación de un entorno de desarrollo que servirá para la creación de aplicaciones en lenguaje MatrioshTS.

2.1.1. Características básicas

- Numero de línea y columna
- Botón para compilar
- Botón para optimizar
- Reporte de errores
- Reporte de tabla de símbolos
- Reporte de AST

2.1.2. Consola

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

3. Flujo de la aplicación

A continuación, se explica el flujo de la aplicación.

La aplicación es sencilla por lo que su funcionalidad se basa en compilar, generar código intermedio, optimizar código intermedio y desplegar reportes.

- **Traducir:**
  Esta opción nos va a permitir traducir una entrada. El programa recibe un archivo de entrada de código de alto nivel y traduce a código intermedio en la sintaxis de tres direcciones.
- **Optimizar:**
  Esta opción nos va a permitir aplicar reglas de optimización a la traducción de código intermedio. Se debe implementar optimización por bloques y mirilla.
- **Reportes:**
  Esta opción nos va a permitir visualizar los reportes generados después de traducir una entrada.

En la ejecución se debe implementar los dos métodos de recuperación: uno para los errores léxicos y sintácticos descartando hasta el ";"; y otro para los errores semánticos, se debe descartar la instrucción con error.

3.1. Flujo especifico de la aplicación

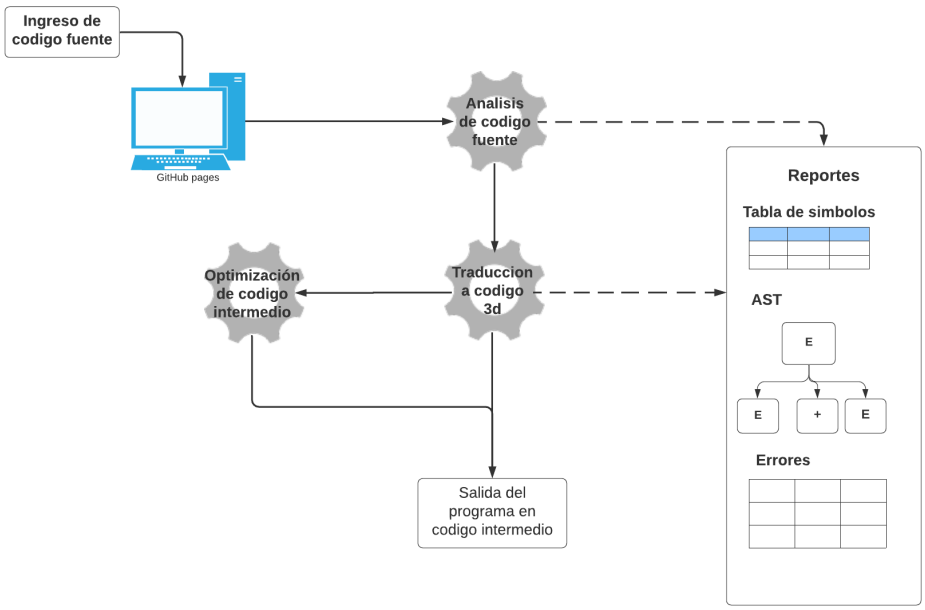

Ilustración 1. Flujo de la aplicación

3.1.1. Generación de reportes

Cuando el programa traduce el código fuente o ejecuta el código resultado, es posible generar los reportes de tabla de símbolos, ast y errores para la verificación de como el estudiante usa las estructuras internas para la interpretación del lenguaje

El reporte de errores generará los errores encontrados en el código al momento de generar su traducción a código intermedio.

Tipo	Descripción	Línea	Columna
Sintáctico	Se esperaba "==", en lugar se encontró "="	112	15
Semántico	El tipo string no puede multiplicarse con un numeric	80	10
sintáctico	Se esperaba la palabra reservada "if" en su lugar se encontró “else”	1000	5

Tabla 1. Ejemplo de posible reporte de errores

El reporte de la tabla de símbolos mostrara una tabla de todos los registros que ingresaron a la tabla, la información a mostrar queda a discreción del estudiante.

Nombre	Tipo	Ámbito	Fila	Columna
Temp	string	global	5	8
A	number	local	10	7
B	number	local	12	5
C	boolean	local	13	10

Tabla 2. Ejemplo de posible reporte de tabla de símbolos

El reporte de AST mostrara el árbol de análisis sintáctico del código de entrada.

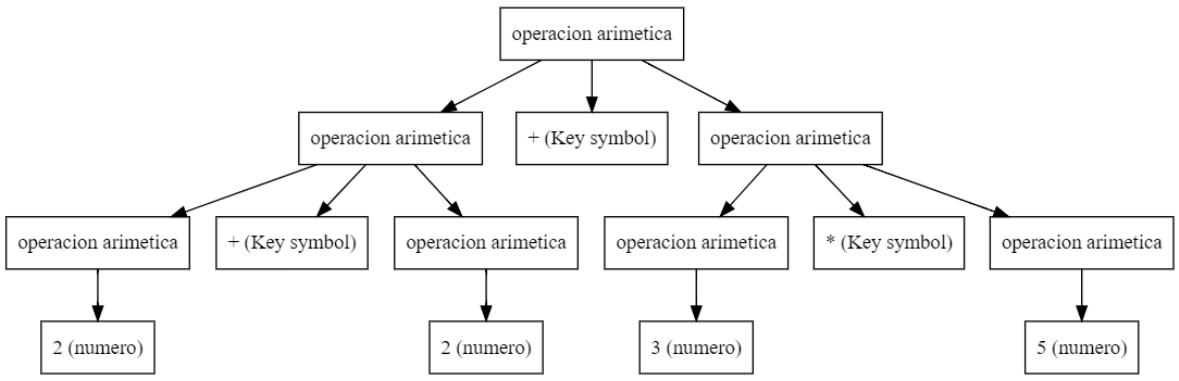

Ilustración 2. Ejemplo de posible salida del reporte AST

4. Sintaxis de MatrioshTS

MatrioshTS nos provee la posibilidad de tipado estático, funciones anidadas.

La sintaxis de MatrioshTS es similar a typescript, pero con la salvedad que no se utilizaran todas las funcionalidades que este lenguaje posee, a continuación, se describen las instrucciones válidas.

En el siguiente enlace se encuentra de forma más detallada la sintaxis y ejemplos: https://www.tutorialsteacher.com/typescript/

En el siguiente enlace se encuentran restricciones sobre funcionalidades del proyecto.

https://docs.google.com/document/d/1BwgZUDp3BDvgTamqCjx-9Z81frXPT5zw1NHp77AkFjI/edit?usp=sharing

4.1. Generalidades

- El lenguaje es case insensitive, por lo tanto, un identificador declarado como PRUEBA es igual a uno declarado como prueba, esto también aplica para las palabras reservadas.

- Los comentarios pueden ser de una línea (//) y de múltiples líneas (/* … */)
- No existe el valor undefined
- Secuencias de escape

Secuencia	Descripción
\”	Comilla doble
****	Barra invertida
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación

4.2. Tipos de dato validos

4.2.1. String:

Se pueden utilizar comillas simples y dobles. Este tipo de dato cuenta con las siguientes propiedades y métodos.

- **Length:** Devuelve el tamaño del string.
- **CharAt(indice):** Recibe como parámetro un entero que indica la posición del carácter que se desea obtener, devuelve el carácter encontrado.
- **ToLowerCase():** Retorna una nueva cadena con todas las letras mayúsculas.
- **ToUpperCase():** Retorna una nueva cadena con todas las letras minúsculas.
- **Concat(str):** Retorna la concatenación de ambas cadenas como una nueva cadena.

4.2.2. Number:

Pueden ser números entero o decimales, positivos o negativos.

4.2.3. Boolean:

Valores true y false.

4.2.4. Void:

Para funciones de tipo void, que no retornan nada.

4.2.5. Types

Estos pueden contener cualquier tipo de dato en su interior, incluso otros types, arreglos o arreglos de types.

Consideraciones:

- Cuando se inicializa el type es obligatorio inicializar sus atributos.
- Si un atributo del type contiene un tipo de dato definido por el usuario, este podrá tener el valor de null.
- La declaración de únicamente será global.
- Usar una función de tipo void como expresión deberá resultar en un error semántico.

4.2.6. Arrays

Los arreglos pueden ser de cualquier tipo de dato valido (Incluso arreglos o types) y son estáticos, además poseen las siguientes propiedades:

- **Length:** Devuelve el tamaño del arreglo.

Consideraciones:

- El tamaño de los arreglos será estático.
- El tamaño del arreglo está definido por su expresión, es decir una expresión [10,20,30,40] corresponderá a un arreglo de tamaño 4.
- El uso de la instrucción **new Array(Expr)** se utilizará para declarar un arreglo, donde la Expr será el tamaño del arreglo.

4.3. Declaraciones de variables

La sintaxis de la declaración puede realizarse utilizando ya sea:

- **let:** Este tipo de declaración proporciona accesibilidad únicamente en el ámbito donde se declaró y en ámbitos hijos, por lo tanto, no se podrá utilizar en ningún ámbito externo.
- **const:** Funciona igual que let, pero con la diferencia de que no permite cambiar el valor de la variable en ningún momento.

Consideraciones:

- Las constantes es obligatorio que se declaren con un valor.
- Todas las variables deben tener un tipo de dato al declararse, el tipo no se va a inferir.
- No puede declararse una variable o constante con un identificador que ya haya sido utilizado dentro del mismo ámbito.
- Las variables solo aceptan un tipo de dato, el cual no puede cambiarse en tiempo de ejecución.
- En Typescript se puede especificar uno o más tipos de dato que soporta una variable, MatrioshTS lo limita a un solo tipo de dato.
- Se debe validar que el tipo de dato y el valor sean compatibles.
- Las variables tienen un tipo definido obligatoriamente, ej: **let x : number; const y : string;**
- Las variables de tipo number y boolean tienen valores por defecto 0 y false respectivamente.
- Las variables de tipo string, type y array tienen valores por defecto null.
- Las constantes no permiten cambiar el valor entero de una variable, pero si permiten cambiar los valores de los elementos de un arreglo y para los types también es valido cambiar actualizar los valores de sus atributos más no reasignar el type.

4.4. Asignación de variables

Las variables no pueden cambiar de tipo de dato, se deben mantener con el tipo declarado inicialmente.

Consideraciones

- No es posible cambiar el tipo de dato de una variable.
- Una constante no puede ser asignada.
- Se debe validar que el tipo de la variable y el valor sean compatibles.

4.5. Operaciones aritméticas

Entre las operaciones disponibles vamos a encontrar las siguientes

- Suma y concatenación
- Resta
- Multiplicación
- División
- Negación
- Exponenciación
- Modulo
- Incremento

- Decremento

Consideraciones

- Se puede concatenar string con boolean y number.
- El resto de las operaciones será solo para number y boolean.

4.6. Operaciones relacionales

- Mayor que
- Menor que
- Mayor o igual que
- Menor o igual que
- Igualdad
- Diferenciación

Consideraciones

- Las operaciones Mayor que, Menor que, Mayor o igual que y Menor o igual que solo están permitidas para tipos de dato number y boolean.
- Las operaciones Igualdad y Diferenciación permiten cualquier tipo de dato.

4.7. Operaciones lógicas

Las operaciones lógicas se evalúan de izquierda a derecha, por lo tanto, se manejará corto circuito para no evaluar todas las condiciones si no es necesario.

- AND
- OR
- NOT

Consideraciones

- Las operaciones lógicas solo están permitidas para tipos de dato boolean.

4.8. Operador ternario

El operador condicional (ternario) es el único operador que tiene tres operandos. Este operador se usa con frecuencia como atajo para la instrucción if.

4.9. Estructuras de control

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. El lenguaje soporta las siguientes estructuras:

- If ... Else
- Switch – Case
- While
- Do While
- For, For … in y For … of

Consideraciones

- Las condiciones únicamente reciben tipos booleanos.

4.10. Sentencias de transferencia

Las sentencias de transferencia permiten manipular el comportamiento normal de los bucles, ya sea para detenerlo o para saltarse algunas iteraciones. El lenguaje soporta las siguientes sentencias:

- Break
- Continue
- Return con expresión
- Return sin expresión

Consideraciones

- Se debe verificar que la sentencia break aparezca dentro de un ciclo o dentro de una sentencia Switch – Case.
- Se debe verificar que la sentencia continue aparezca dentro de un ciclo.
- Se debe verificar que la sentencia return aparezca dentro de una función.

4.11. Funciones

MatrioshTS como cualquier otro lenguaje permite el uso de funciones, estas pueden o no retornar un valor, la característica especial y distintivo con otros lenguajes es que nos permite anidar funciones, por lo que el estudiante deberá aplicar conceptos de atributos heredados para realizar el desanidamiento de estas, protegiendo siempre el acceso de las variables en su ámbito correspondiente, ya que, al generar código intermedio, este no soporta tal anidación.

Consideraciones:

- Las funciones no soportan sobrecarga.
- Las funciones en MatrioshTS se pueden anidar cualquier cantidad de veces.
- Las funciones anidadas pueden invocar otras funciones GLOBALES y locales siempre y cuando estén en el mismo ámbito.
- Los parámetros deben tener distinto nombre.
- Las funciones pueden retornar cualquier tipo de dato y este debe estar especificado.
- Las funciones tipo flecha no son soportadas por MatrioshTS.

4.12. Funciones nativas

4.12.1. console.log

Esta función nos permite imprimir una lista de expresiones. La función está limitada a únicamente imprimir valores de tipo number, string o boolean.

5. Generación de código intermedio

Cuando el compilador termine la fase de análisis de un programa realizará una transformación a una representación intermedia equivalente al código de alto nivel.

Ya que código intermedio no es en sí un lenguaje ya definido sino un concepto, por facilidades de calificación y de pruebas se utilizará la estructura del lenguaje C como referencia a código intermedio, manejando ciertas limitaciones para que se apliquen correctamente los conceptos de generación de código intermedio.

5.1. Tipos de Dato

El lenguaje solo acepta tipos de dato numéricos, es decir tipos int y float. Por facilidad se recomienda trabajar todas las variables como tipo float.

Consideraciones:

- Está prohibido el uso de otros tipos de dato.
- Las estructuras Heap y Stack se realizan por medio de arreglos de tipo float.

- El uso de arreglos está permitido, queda a discreción del estudiante como implementarlo.

5.2. Temporales

Los temporales serán creados por el compilador en el proceso de generación de código intermedio. Estos serán variables de tipo float, el identificador asociado queda a discreción del estudiante, pero se recomienda utilizar la que se muestra a continuación.

```
t[0-9]+

t1
t13924
```

5.3. Etiquetas

Las etiquetas serán creadas por el compilador en el proceso de generación de código intermedio. El identificador asociado a las etiquetas queda a discreción del estudiante, aunque se recomienda utilizar la que se muestra a continuación.

```
L[0-9]+

L1
L43
```

5.4. Identificadores

Un identificador será utilizado para dar un nombre a variables, métodos o estructuras. Es una secuencia de caracteres alfabéticos [A-Z a-z] incluyendo el guion bajo [_] o dígitos [0-9] que comienzan con un carácter alfabético o guion bajo.

```
_id1
Id1
Id_32
```

5.5. 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 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 “*/”.

```
// COMENTARIO DE UNA LINEA

/**
* COMENTARIO MULTILINEA
*/
```

5.6. Operadores aritméticos

Las operaciones aritméticas contarán con un argumento 1, argumento 2, campo para el resultado y un operador perteneciente a la siguiente tabla:

Operación	Símbolo	C
SUMA	+	t1 = t0 + 1;
RESTA	-	t1 = t0 - 1;
MULTIPLICACIÓN	*	t1 = t0 * 1;
DIVISIÓN	/	t10 = 4 / 2;
MODULO	%	t11 = 2 % 1;

Consideraciones

- Únicamente se permite el uso de dos argumentos en una expresión.

5.7. Saltos

Para definir el flujo que seguirá el intérprete se contará con bloques de código, estos bloques están definidos por etiquetas y saltos, los saltos son aquellos que, definiendo una etiqueta y una posible condición, el flujo del código se desplaza hasta el bloque contenido en luego de dicha etiqueta. El lenguaje C cuenta con 2 tipos de saltos, los cuales son:

- **Salto condicional:** Contará con una condición para decidir si se realiza el salto o no.
- **Salto incondicional:** Se realizará el salto siempre.

5.7.1. Saltos incondicionales

Los saltos incondicionales contarán únicamente con resultado y operador, donde el resultado contendrá la etiqueta destino y el operador estará definido por la instrucción goto. Este realizará el salto hacia una etiqueta que se le especifique.

```
goto L1;
printf("%c", 64); //Esta instruccion no se ejecuta
L1:
T2 = 100;
```

5.7.2. Saltos condicionales

Para simular los saltos condicionales se utilizará la instrucción IF del lenguaje C que tendrá como condición una expresión relacional.

Operación	Simbolo	C
Menor que	<	4 < 5
Mayor que	>	t1 > t2
Menor o igual que	<=	t3 <= t4
Mayor o igual que	>=	t5 >= t6
Diferente que	!=	t7 != t8
Igual que	==	8 == t2

```
If(5 < 10) goto L1;
goto L2;
```

```
L1:
//código si la condición es verdadera

L2:
//código si la condición es falsa
```

Consideraciones

- Existen expresiones booleanas de uno y dos argumentos, tal como se muestra en la Figura 6.37 del libro. Las expresiones booleanas con operadores relacionales se traducen en instrucciones if con un salto para verdadero y otro salto para falso, página 404 del libro.
- La instrucción If solo permite una instrucción, está prohibido el uso de if anidados.
- No se permite el uso de la instrucción Else.

5.8. Asignación a temporales

La asignación nos va a permitir cambiar el valor de los temporales, para lograrlo se utiliza el operador igual, este permite una asignación directa o una expresión.

```
//Entrada codigo alto nivel
Console.log(1+2*5);

//Salida codigo intermedio en lenguaje C
T1 = 2 * 5;
T2 = 1 + T1;

Print("%d", T2);
```

5.9. Métodos

Estos son bloques de código a los cuales se accede únicamente con una llamada al método. Al finalizar su ejecución se retorna el control al punto donde fue llamado para continuar con las siguientes instrucciones.

```
function funcion1(number a, number b) : number{
   number c = a + b;
   return c;
}

void funcion1(){
   T1 = P + 1;
   T2 = STACK[T1]; //Variable a

   T3 = P + 2;
   T4 = STACK[T3]; //Variable b

   T5 = T2 + T4; //a + b
   T6 = P + 3;
   STACK[T6] = T5; //Variable c

   T7 = P + 3;
   T8 = STACK[T7]; //Obtengo la Variable c
```

```
   STACK[P] = T8; //Guardo el valor que retorna la funcion
   goto L0; //Simulacion de un return con saltos a etiquetas

   L0: //Etiqueta de retorno
   return;
}
```

Consideraciones

- Está prohibido el uso de paso de parámetros en los métodos, el estudiante debe utilizar la pila para el paso de parámetros.
- Los métodos solo pueden ser de tipo void.
- Al final de cada método incluir la instrucción “return;”.

5.10. Llamadas a métodos

Esta instrucción nos permite invocar métodos.

```
Identificador();

// Al finalizar la ejecución del método se ejecutan las instrucciones posteriores a la llamada

Void main(){
  Funcion1(); // Se llama a la funcion1 desde el metodo main
 return;
}

Void Funcion1(){
 printf(“%d”, 100);
 return;
}
```

Al hacer la llamada el flujo cambia y empieza a ejecutar las instrucciones del método, al finalizar regresa al punto donde fue llamado y sigue ejecutando las siguientes instrucciones

5.11. Printf

Su función principal es imprimir en consola un valor definido según el formato del parámetro que se le asigne, la sintaxis y parámetros permitidos son los siguientes:

Parámetro	Acción
%c	Imprime el valor carácter del identificador, se basa según el código ascii.
%d	Imprime únicamente el valor entero del valor.
%f	Imprime con punto decimal el valor.

```
printf(“%e”, (int)900) // imprime 900

printf(“%c”, (char)65) // imprime A

print(“%f”, (float)65.4) // imprime 65.4
```

5.12. Estructuras en tiempo de ejecución

El proceso de compilación genera el código intermedio y este se va a ejecutar, siendo indispensable para el flujo de la aplicación. En el código intermedio no existen cadenas, operaciones complejas, llamadas a métodos con parámetros y muchas otras cosas que sí existen en los lenguajes de alto nivel.

Para el proyecto se va a contar con 2 estructuras lineales y por convención llevaran el nombre de stack y heap, estas nos van a servir para almacenar los valores de nuestros temporales y poder hacer la simulación de ejecución del código intermedio.

5.12.1. STACK

El stack es la estructura que se utiliza en código intermedio para controlar las variables locales, y la comunicación entre métodos (paso de parámetros y obtención de retornos en llamadas a métodos). Se compone de ámbitos, que son secciones de memoria reservados exclusivamente para cierto grupo de sentencias.

Cada llamada a método o función que se realiza en el código de alto nivel cuenta con un espacio propio en memoria para comunicarse con otros métodos y administrar sus variables locales. Esto se logra modificando el puntero del Stack, que en el proyecto se identifica con la **letra P**, para ir moviendo el puntero de un ámbito a otro, cuidando de no corromper ámbitos ajenos al que se está ejecutando.

El puntero se identifica con la letra “P” y este contendrá la dirección de memoria donde comenzará el ámbito actual, su asignación se realizará exactamente igual que a como los terminales.

```
float p;

P = P + 1;
```

5.12.2. HEAP

Es la estructura de control del entorno de ejecución encargada de guardar las referencias a variables globales o valores de cadenas y arreglos. El puntero se definirá con el identificador H y a diferencia de P (que aumenta y disminuye según lo dicta el código intermedio), este únicamente aumenta y aumenta, su función es brindar siempre la próxima posición libre de memoria.

Esta estructura también será la encargada de almacenar las cadenas de texto, guardando únicamente en cada posición el ASCII de cada uno de los caracteres que componen la cadena a guardar.

```
// Puntero H
H = H + 1;
T1 = H;
```

5.12.3. Acceso y asignación a estructuras en tiempo de ejecución

La sintaxis para acceso o asignación es la siguiente.

```
//Asignacion
Heap[0] = T1;
Stack[P] = 100;


//Acceso
T1 = Heap[0];
```

```
T2 = Stack[P];
```

Consideraciones

- La asignación a las estructuras se realiza por medio de temporales o valores constante, no se permite el uso de expresiones aritméticas o lógicas para la asignación a estas estructuras.
- El acceso a las estructuras se realiza por medio de temporales, no se permite la asignación a una estructura mediante el acceso a otra estructura, es decir "stack[0] = heap[100];".
- Si el acceso a las estructuras se hace por medio de temporales, se debe realizar un casteo ya que los temporales son de tipo float. Ej. "stack[(int)t1] = 1".

5.13. Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar. Únicamente en esta sección se permite el uso de declaraciones, no se pueden realizar declaraciones adentro de métodos. La estructura del encabezado se muestra a continuación.

```
#include <stdio.h> //Importar para el uso de Printf

float heap[16384]; //Estructura para heap
float stack[16394]; //Estructura para stack
float p; //Puntero P
float h; //Puntero H
float t1, t2, t3, t4, t5, t6, t7, t8, t9, t10; //Lista de temporales utilizados
```

5.14. Método Main

En este método iniciará el flujo de ejecución del código traducido. Su estructura es la siguiente:

```
void main() {
   //Instrucciones
   return;
}
```

6. Optimización de código intermedio

Se debe poder realizar una optimización sobre el código generado, estas optimizaciones se pueden realizar:

- **A nivel local:** sólo utilizan la información de un bloque básico para realizar la optimización.
- **A nivel global:** que usan información de varios bloques básicos.

Un bloque básico es una unidad fundamental de código. Es una secuencia de proposiciones donde el flujo de control entra en el principio del bloque y sale al final del bloque. Los bloques básicos pueden recibir el control desde más de un punto en el programa (se puede llegar desde varios sitios a una etiqueta) y el control puede salir de más de una proposición (se podría ir a una etiqueta o seguir con la siguiente instrucción)

Los tipos de transformación para realizar la optimización local serán los siguientes:

- Eliminación de instrucciones redundantes de carga y almacenamiento.
- Eliminación de código muerto.

- Simplificación algebraica y reducción por fuerza.

Cada uno de estos tipos de transformación utilizados para la optimización, tendrán asociadas reglas las cuales en total son 16, estas se detallan a continuación:

6.1. Eliminación de código muerto

Consistirá en eliminar las instrucciones que nunca serán utilizadas. Por ejemplo, instrucciones que estén luego de un salto incondicional, el cual direcciona el flujo de ejecución a otra parte y nunca llegue a ejecutar las instrucciones posteriores al salto incondicional. Las reglas aplicables son las siguientes:

Regla 1

Si existe un salto condicional de la forma Lx y exista una etiqueta Lx:, todo código contenido entre el **jmp ,,, Lx** y la etiqueta Lx, podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

Ejemplo	Optimización
goto L1; <instrucciones> L1: T51 = t1 + t50;	goto L1; L1: T51 = t1 + t50;

Regla 2

En un salto condicional si existe inmediatamente un salto después de sus etiquetas verdaderas se podrá reducir el número de saltos negando la condición, cambiando el salto condicional hacia la etiqueta falsa Lf: y eliminando el salto innecesario a Lf y quitando la etiqueta Lv:.

Ejemplo	Optimización
If(4 == 4) goto L10; goto L11; L10: <instrucciones1> L11: <instrucciones2>	If (4!= 4) goto L11; <instrucciones1> L11: <instrucciones2>

Regla 3

Si se utilizan valores constantes dentro de las condiciones y el resultado de la condición es una constante verdadera, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta falsa Lf.

Ejemplo	Optimización
If(1 == 1) goto L1; goto L2;	If(1 == 1) goto L1;

Regla 4

Si se utilizan valores constantes dentro de las condiciones y el resultado de la condición es una constante falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera Lv.

Ejemplo	Optimización
If(5!=1) goto L1; goto L2;	goto L2;

6.2. Eliminación de instrucciones redundantes de carga y almacenamiento

Regla 5

Si existe una asignación de valor de la forma a = b y posteriormente existe una asignación de forma b = a, se eliminará la segunda asignación siempre que a no haya cambiado su valor. Se deberá tener la seguridad de que no exista el cambio de valor y no existan etiquetas entre las 2 asignaciones

Ejemplo	Optimización
T10 = B; B = t10;	T10 = B;

6.3. Simplificación algebraica y reducción por fuerza

La optimización local podrá utilizar identidades algebraicas para eliminar las instrucciones ineficientes.

Para las reglas 6,7,8,9 se eliminan las expresiones algebraicas que no afectan el valor de una variable y que se asigna a ella misma, por ejemplo, las sumas/restas con 0 y la multiplicación/división por 1.

Regla 6

Ejemplo	Optimización

T1 = t1 + 0;	<codigo eliminado>

Regla 7

Ejemplo	Optimización
t2 = t2 + 0;	<codigo eliminado>

Regla 8

Ejemplo	Optimización
t3 = t3 * 1;	<codigo eliminado>

Regla 9

Ejemplo	Optimización
t4 = t4 / 1;	<codigo eliminado>

Las reglas 10, 11, 12, 13 describen operaciones con diferentes variables de asignación y una constante si estas operaciones son sumas/restas de 0 y multiplicaciones/divisiones con 1, la instrucción se transforma a una asignación

Regla 10

Ejemplo	Optimización
Y = x + 0;	Y = x;

Regla 11

Ejemplo	Optimización
Y = x - 0;	Y = x;

Regla 12

Ejemplo	Optimización
Y = x * 1;	Y = x;

Regla 13

Ejemplo	Optimización
Y = x / 1;	Y = x;

Para las reglas 14, 15, 16 se deberá realizar la eliminación de reducción por fuerza para sustituir por operaciones de alto costo por expresiones equivalentes de menor costo.

Regla 14

Ejemplo	Optimización
X = y * 2;	X = y + y;

Regla 15

Ejemplo	Optimización
X = y * 0;	X = 0;

Regla 16

Ejemplo	Optimización
X = y / 0 ;	X = 0;

7. Reportes generales

7.1. Tabla de símbolos

Este reporte mostrará la tabla de símbolos durante y después de la compilación 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.

7.2. AST

Este reporte mostrara el árbol de análisis sintáctico que se produjo al analizar el archivo de entrada. Este debe de representarse como un grafo, se recomienda se utilizar Graphviz. El Estudiante deberá mostrar los nodos que considere necesarios y se realizarán preguntas al momento de la calificación para que explique su funcionamiento.

5.12. Reporte de Optimización

Este reporte mostrará las reglas de optimización que fueron aplicadas sobre el código intermedio. Se debe indicar que tipo de optimización utilizó y en que sección. Como mínimo se piden la siguiente información:

- Tipo de optimización (Mirilla o por bloques).
- Regla de optimización aplicada.
- Código eliminado.
- Código agregado.
- Fila

7.3. Reporte de errores

El traductor e 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.

Los tipos de errores se deberán de manejar son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos.

La tabla de errores debe contener la siguiente información:

- Línea: Número de línea donde se encuentra el error.
- Columna: Número de columna donde se encuentra el error.
- Tipo de error: Identifica el tipo de error encontrado. Este puede ser léxico, sintáctico o semántico.
- Descripción del error: Dar una explicación concisa de por qué se generó el error.
- Ámbito: Si fue en una función (decir en cual fue) o en el ámbito global.

8. Entregables y calificación

Para el desarrollo del proyecto se deberá utilizar un repositorio de github, este repositorio deberá ser privado.

8.1. Entregables

Todo el código fuente se va a manejar en github, por lo tanto, el estudiante es el único responsable de mantener actualizado dicho repositorio hasta la fecha de entrega, ya que, si se agregan más archivos luego de la fecha y hora establecidos, entonces no se tendrá derecho a calificación.

- Código fuente publicado en un repositorio de github.
- Enlace al repositorio y permisos a los auxiliares para poder acceder.
- Aplicación web con la funcionalidad del proyecto publicada en github-pages.

8.2. Restricciones

- La herramienta para generar los analizadores del proyecto será JISON. La documentación se encuentra en el siguiente enlace http://zaa.ch/jison/docs/
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ciencias y Sistemas los involucrados.
- El método de calificación principal y óptimo será por medio de Github-pages.
- El método alterno de calificación será de manera local, indicando las herramientas utilizadas para que el auxiliar las instale y corra su proyecto.
- El desarrollo y entrega del proyecto es individual.
- Únicamente se permite el uso de las instrucciones del lenguaje C establecidas en el enunciado.

8.3. Consideraciones

- No hay requerimientos mínimos.
- Durante la calificación se realizarán preguntas sobre el código para verificar la autoría de este, de no responder correctamente la mayoría de las preguntas se reportará la copia.
- Es válido el uso de cualquier framework o librería para el desarrollo de aplicaciones con javascript siempre y cuando la aplicación final pueda ser publicada en github pages (Se recomienda la librería de react o los frameworks de angular o ionic para facilitar el desarrollo).
- El repositorio únicamente debe contener el código fuente empleado para el desarrollo, no deben existir archivos pdf o docx (utilizar un archivo .gitignore).
- El sistema operativo para utilizar es libre.
- Los lenguajes están basados en Typescript, por lo que el estudiante es libre de realizar archivos de prueba en estas herramientas, el funcionamiento debería ser el mismo y limitado a lo descrito en este enunciado.
- Los estudiantes NO DEBEN REALIZAR UN INTERPRETE DE C.
- Los estudiantes son libres de probar su código traducido en lenguaje C en un compilador local o en un compilador online.
- Debido a la alta cantidad de preguntas, se harán foros semanales para limitar la cantidad de preguntas y se mas sencillo buscar respuestas, el primero se estará llevando en el siguiente enlace. https://github.com/PvasquezF/ArchivosPruebaOLC2/issues/52
- En la siguiente página se encuentra con compilador de C online: https://www.onlinegdb.com/online_c_compiler.
- Typescript online https://www.typescriptlang.org/play

- Se van a publicar archivos de prueba en el siguiente repositorio: https://github.com/PvasquezF/ArchivosPruebaOLC2

8.4. Calificación

- La calificación del proyecto será mediante la aplicación web publicada en github pages.
- El tiempo de calificación será de 20 minutos (puede aumentar dependiendo la cantidad de proyectos entregados).
- 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.
- Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrollo.
- Los archivos de entrada podrán ser modificados si contienen errores léxicos, sintácticos o semánticos no descritos en el enunciado o provocados para verificar el manejo y recuperación de errores.
- Se utilizará un compilador de C para evaluar que la salida del archivo de entrada sea la correcta.

8.5. Entrega del proyecto

- La entrega será mediante github, y se va a tomar como entrega el código fuente publicado en el repositorio a la fecha y hora establecidos.
- Cualquier commit luego de la fecha y hora establecidos invalidará el proyecto, por lo que no se tendrá derecho a calificación.
- No habrá prorroga
- Fecha de entrega:

Viernes 8 de noviembre a las 23:59 PM