Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Primer Semestre 2021
Catedráticos: Ing. Edgar Sabán, Ing. Bayron López, Ing. Erick Navarro e Ing. Luis Espino
Tutores académicos: Erik Flores, Cristian Alvarado, Ronald Romero, Manuel Miranda, Haroldo Arias

Compi Pascal

Contenido

1. Competencias

1.1. Competencia general

- Que el estudiante aplique la fase de síntesis del compilador para que pueda implementar un compilador utilizando herramientas de análisis ascendente.

1.2. Competencia específica

- Que el estudiante utilice un generador de analizadores léxicos y sintácticos para construir un compilador.
- Que el estudiante implemente la generación de código de 3 direcciones de un lenguaje de alto nivel.
- Que el estudiante aplique las reglas de optimización en el código de 3 direcciones.

2. Descripción

2.1 Descripción General

Para el segundo proyecto se deberá desarrollar un compilador que acepta un lenguaje basado en pascal, el compilador generará una salida en código en formato de 3 direcciones que se especifica en la sección 5, este código intermedio será ejecutado por un compilador de C.

Una de las características del lenguaje Pascal es la definición y utilización de funciones anidadas dentro del código, el funcionamiento se encuentra detallado en la sección 4 de este enunciado.

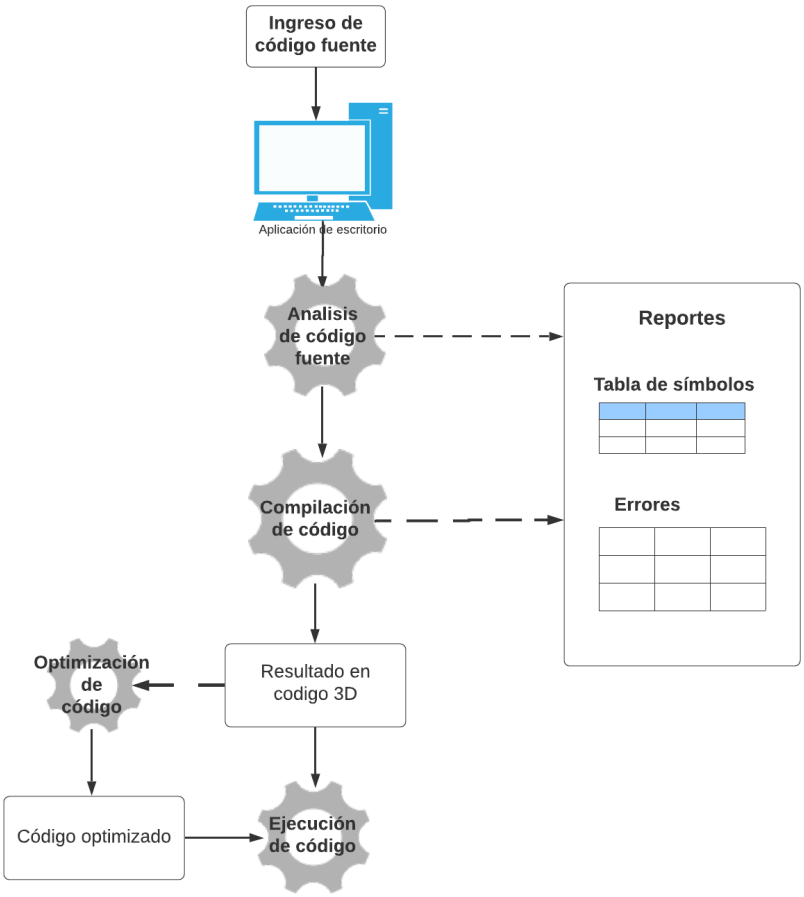
El programa también debe ser capaz de analizar una entrada de código en formato de 3 direcciones y optimizar dicho código utilizando reglas de optimización, las cuales están descritas a detalle en la sección 6 de este enunciado.

La aplicación contará con los componentes descritos en la sección 3 y debe ser desarrollado en el lenguaje .Net en su Framework .Net Core 3.1, utilizando la herramienta Irony para la implementación de un analizador ascendente.

El proceso de compilación es el siguiente:

- **Compilación del código fuente**: El compilador analizará el código fuente basado en pascal y luego generará su respectiva representación en formato 3 direcciones.
- **Optimización de código en 3 direcciones:** Se podrá optimizar el código generado de la compilación por medio de las reglas especificadas en la sección 6, este código optimizado no debe cambiar la funcionalidad del programa original.
- **Generación de reportes:** Luego de la compilación del programa se debe poder generar:
  - Reporte de tabla de símbolos
  - Listado de errores
  - Reporte de optimización.

2.2 Flujo específico de la aplicación



2.2.1. Ingreso de código fuente

El lenguaje está basado en Pascal con instrucciones limitadas, en la sección 4 se especifican las limitaciones del lenguaje.

2.2.2 Compilación de código fuente

El estudiante deberá generar por medio de la compilación del código fuente, un programa equivalente en formato de código de 3 direcciones, si existen errores se deberán de reportar luego de la compilación.

2.2.3. Optimización de código

Se puede optimizar el código de 3 direcciones que se generó a partir del programa fuente, se realizara la optimización utilizando las reglas descritas en la sección 6.

2.2.4. Ejecución

Ejecución sobre el código en formato en 3 direcciones por medio de un compilador de C.

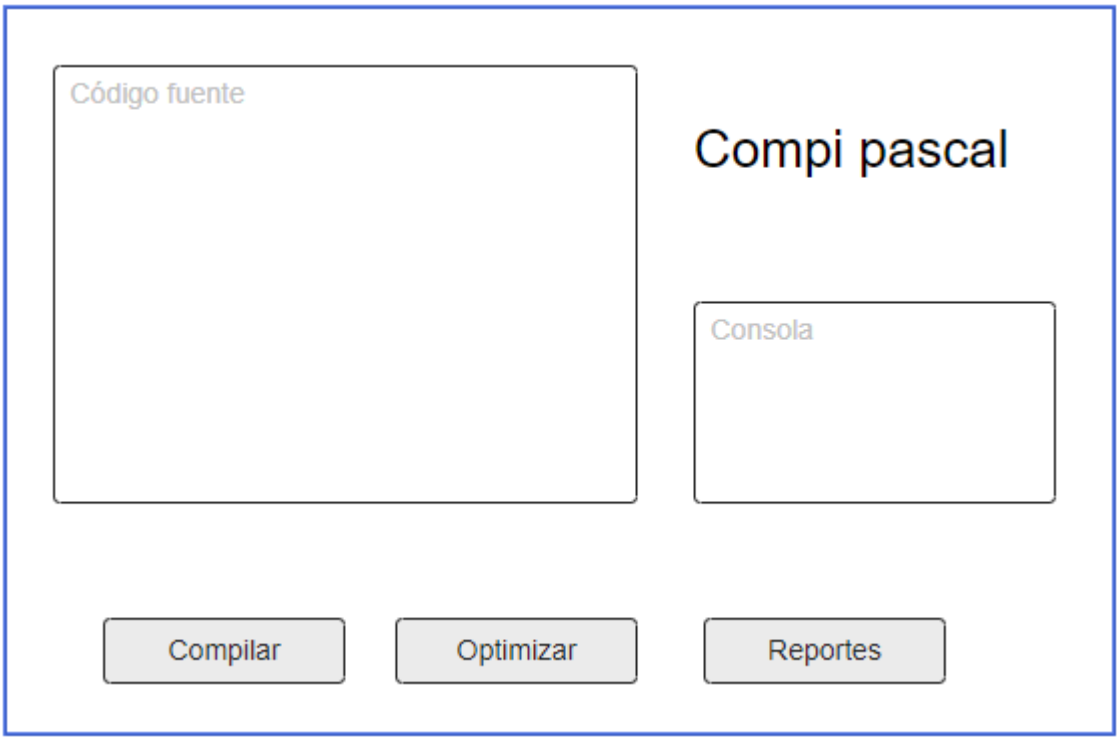
2.2.5. Generación de reportes

Cuando el programa compila el código fuente, es posible generar los reportes de tabla de símbolos, AST, optimización y errores para la verificación de como el estudiante usa las estructuras internas para la compilación del lenguaje.

3. Componentes de la aplicación

La aplicación deberá contar con los siguientes elementos para su correcto funcionamiento:

- **Botón de ejecución**: Al presionarlo ejecutará el código de entrada y si existen errores en su análisis y ejecución los deberá reportar.
- **Botón de optimización:** Al presionarlo optimizará el código de la compilación y luego mostrará el nuevo código en la consola de salida.
- **Botón de reportes**: Mostrará al presionarlo el reporte de errores, AST y tabla de símbolos.
- **Consola de salida**: Esta muestra el resultado en formato de 3 direcciones.



Ejemplo de interfaz gráfica

4. Sintaxis de Compi Pascal

Compi Pascal provee la posibilidad de tipado estático, funciones anidadas y que su traducción sea exactamente la misma a excepción de las funciones, ya que las funciones estarán desanidadas luego de la traducción inicial.

La sintaxis de Compi Pascal es similar a Pascal, pero con la salvedad que no se utilizarán 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.tutorialspoint.com/pascal/index.htm

4.1. Generalidades

- El lenguaje no es case sensitive, por lo tanto, un identificador declarado como PRUEBA es igual a uno declarado como prueba, esto también aplica para las palabras reservadas.
- Existen 3 tipos de comentarios:
    - Comentarios de una línea **//**
    - Comentarios de múltiples líneas **(* … *)**
    - Comentarios de múltiples líneas **{…}**

4.2. Tipos de dato validos

4.2.1. String:

Se deben utilizar comillas simples, las comillas dobles no están permitidas para una cadena.

4.2.2. Integer:

Son números enteros positivos o negativos.

4.2.3. Real:

Son números decimales positivos o negativos.

4.2.4. Boolean:

Valores true y false.

4.2.6. Types

En pascal podemos definir diversos tipos de datos, para este proyecto limitaremos solo a los siguientes:

4.2.6.1. Objects

Estos pueden contener cualquier tipo de dato o arreglo en su interior, incluyendo otros objects o arreglos de objects.

Para la definición de objects, Compi Pascal se limita a lo que tiene el lenguaje nativo de Pascal, ya que Compi Pascal utiliza los objects como la definición de structs de C, por lo cual la estructura de estos será la siguiente:

```
{
    Definición de un Type en Compi Pascal
}

program Hello;

type
   Rectangle = object
   var
      length, width: integer;
      area, volumen: real;
end;

var
r1: Rectangle;

begin
    r1.length := 10;
    r1.volumen :=50.0;
    writeln (r1.length);
    writeln (r1.volumen);
end.
```

Consideraciones:

- No es necesario inicializar los valores de los atributos del object.
- Cuando se declara un type, sus variables internas se inicializarán con los valores por defecto de Pascal.
- No se permiten la declaración de funciones o procedimientos dentro del object.

4.2.6.2. Arrays

Los arreglos pueden ser de cualquier tipo de dato válido (Incluso arreglos o types), además poseen la siguiente estructura:

```
type
   array-identifier = array[index-type] of element-type;
```

Consideraciones:

No se utilizarán Enumerated Types y Subrange Types.

4.3. Declaraciones de variables y constantes

La sintaxis de la declaración debe ser igual a la sintaxis de pascal, en donde se pueden definir listas de variables de un tipo en específico y para el caso de constantes únicamente una constante por definición.

Consideraciones:

- Las constantes es obligatorio que se declaren con un valor.
- Si se definen variables con asignación de un valor, únicamente se permite agregar una variable en la definición como lo delimita el lenguaje pascal.
- No puede declararse una variable o constante con un identificador que tengan el mismo nombre.
- Si una variable no se inicializa, tomará los valores por defecto del lenguaje Pascal.
- Las variables solo aceptan un tipo de dato, el cual no puede cambiarse en tiempo de ejecución.
- Se debe validar que el tipo de dato y el valor sean compatibles, según las definiciones del lenguaje Pascal.

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, a menos que el tipo de dato no se haya especificado al declarar la 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 se encuentran las siguientes

- Suma (+)
- Resta (-)
- Multiplicación (*)
- División (/)
- Modulo (MOD)

4.6. Operaciones relacionales

- Mayor que (>)
- Menor que (<)
- Mayor o igual que (>=)
- Menor o igual que (<=)
- Igualdad (=)
- Diferenciación (<>)

4.7. Operaciones lógicas

- AND
- OR
- NOT

4.8. 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-then
- If-then-else
- Case
- Case - else
- While-do
- Repeat-until
- For-do

4.9. 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

Consideraciones

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

4.10. Funciones y procedimientos

Compi Pascal permite el uso de funciones y procedimientos, donde las funciones realizan una acción de retorno y los procedimientos no.

```
program functionsandprocs;
var a:integer = 50;
var min:integer;
// factorial usando la función Exit para asignar el retorno.
function factorial( num : integer) : integer;
begin
  if num = 1 then
    Exit(1)
  else
    Exit(num * factorial( num-1 ));
end;
// factorial usando el nombre de la función para asignar el retorno
function factorial2( num : integer) : integer;
begin
  if num = 1 then
    factorial2 := 1
  else
    factorial2 := num * factorial2( num-1 )
end;
// ejemplo de procedimiento
procedure findMin(x, y, z: integer; var m: integer); {m es por referencia}
(* Finds the minimum of the 3 values *)
begin
   if x < y then
      m:= x
   else
      m:= y;

   if z < m then
      m:= z;
end;
begin
  writeln('el resultado es: ', factorial(5));
  writeln('el resultado es: ', factorial2(5));
  findMin(1, 2, 3, min); (* Procedure call *)
  writeln(' Minimum: ', min);
end.
```

4.10.1. Parámetros de funciones y procedimientos

Todos los parámetros son por valor, a menos que se le anteponga la palabra reservada **var** al nombre del parámetro, lo cual lo convierte a una referencia. Tal como aparece en el ejemplo del inciso 4.10.

4.10.2. Procedimientos anidados

Una de las características del lenguaje pascal es el uso procedimientos anidados, lo que significa que se pueden definir procedimientos o funciones dentro de los mismos procedimientos en el lenguaje y pueden ser llamadas dentro de ellos.

```
(*Ejemplo entrada de un procedimiento adentro de un
procedimiento*)

program Ejemplo_anidadas;
(* procedimiento padre *)
procedure saludo;
    (*Procedimiento hijo de saludo*)
    procedure despedida;
    begin
        writeln('adios compañero');
    end;

begin
    writeln('hola compañero');
    despedida();
end;

(* Main *)
begin
  saludo();
end.
```

4.10.3. Funciones anidadas

Al igual que los procedimientos anidados, las funciones pueden llevar funciones o procedimientos dentro de la definición de sus elementos, pero, a diferencia de los procedimientos, todas las funciones deben retornar un valor.

```
(* Ejemplo entrada de un procedimiento adentro de un procedimiento *)
program Ejemplo_anidadas_2;
(* procedimiento padre *)
function calificacion:integer;
    (*Procedimiento hijo de saludo*)
    function nota:integer;
    begin
        writeln('Generando nota');
        nota :=10;
    end;

begin
    writeln('Creando nota');
    calificacion:=nota()*10;
end;

(* Main *)
begin
  writeln(calificacion());
end.
```

4.11. Funciones nativas

4.11.1. write y writeln

Esta función nos permite imprimir cualquier expresión que le mandemos como parámetro, y en un formato comprensible, tomar las consideraciones y variaciones que posee el lenguaje pascal.

4.11.2. Exit

A diferencia de la asignación del valor de retorno de una función en base a su nombre, esta función también detiene la ejecución en el lugar exacto en donde es invocada.

4.11.3. graficar_ts

Esta función nos va a permitir graficar la tabla de símbolos en cualquier parte del programa donde se encuentre esta instrucción, esto servirá para validar el correcto manejo de los ámbitos.

Consideraciones:

- ***Las funciones o métodos creados en 3 direcciones no deben contener anidamientos, es decir, se deben desanidar las funciones del código fuente.***
- Las funciones y procedimientos no soportan sobrecarga.
- Las funciones y procedimientos en Compi Pascal 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.
- No es posible declarar variables dentro del cuerpo del procedimiento o función.
- Los parámetros deben tener distinto nombre.
- Las funciones pueden retornar cualquier tipo de dato y este debe estar especificado.
- Únicamente las funciones pueden retornar un valor.
- Se tomarán en cuenta los primeros 5 decimales de un resultado para la calificación.
- En el cuerpo de la función debe haber una asignación de la forma **name := expresión;** que asigna un valor al nombre de la función. Este valor es retornado cuando la función es ejecutada.

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.

El código intermedio es un tema conceptual, no un lenguaje, en el proyecto se utilizará la estructura del lenguaje C como referencia a código intermedio, manejando este lenguaje con limitaciones para que se apliquen correctamente los conceptos de generación de código intermedio aprendido en la clase magistral.

Se prohíbe el uso de toda función o característica del lenguaje C NO DESCRITA en este apartado.

5.1 Tipos de dato

El lenguaje solo acepto tipos de dato numéricos, es decir, tipos int y float.

Consideraciones:

- *Está prohibido el uso de otros tipos de dato.*
- *El uso de arreglos no está permitido, únicamente para las estructuras heap y stack que se explican más adelante*
- *Para facilidad, se recomienda trabajar todas las variables de tipo float.*

5.2 Temporales

Los temporales serán creados por el compilador en el proceso de generación de código intermedio. Estas serán variables de tipo float, el identificador asociado queda a discreción del estudiante.

El siguiente ejemplo es una recomendación para nombrar las variables temporales.

```
//T [0 – 9] +

t10
t152
```

5.3 Etiquetas

Las etiquetas son identificadores únicos que indican una posición en el código fuente, estas mismas serán creadas por el compilador en el proceso de generación de código intermedio. El identificador asociado a las etiquetas queda a discreción de los estudiantes.

El siguiente ejempló es una recomendación para nombrar las etiquetas en el código c.

```
//L[0-9]+
L125:
L200:
```

5.4 Identificadores

Los identificadores serán utilizados para dar un nombre a las 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 un guion bajo.

```
Id_100
Id123
_id425
```

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 formas de escribir los comentarios:

- **Los comentarios de una línea**, que serán delimitados al inicio con los símbolos "//" y al final con un carácter de salto de línea.
- **Los comentarios con múltiples líneas** iniciaran con los símbolos "/*" y finalizaran con los símbolos "*/"

```
//Comentario de una linea

/* Estos comentarios
   Pueden llevan muchas líneas * y otros simbolos
*/
```

5.6 Operadores aritméticos

Las operaciones aritméticas contaran con:

- Resultado
- Argumento 1
- Argumento 2
- Operador

La tabla de operadores permitidos para el código intermedio es la siguiente

Operación	Símbolo	C
SUMA	+	t1 = t0 + 1
RESTA	-	t100 = 10 - 5
MULTIPLICACION	*	t50 = 3 * 3
DIVISION	/	T2 = 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 programa se contara con bloques de código, estos bloques están definidos por etiquetas.

Los saltos son instrucciones que indican al interprete "mover" la ejecución hacia otra línea de código dentro del programa, la instrucción que indica que se realizará un salto condicional es la palabra reservada **"goto"**.

En el proyecto se utilizarán los 2 formatos de saltos que son:

- **Condicional:** Son saltos que cuentan con una condición para decidir si se realiza el salto o no.
- **No condicional:** Son saltos que se realizan obligativamente.

5.7.1 Saltos no condicionales

El formato de saltos no condicionales contara únicamente con una instrucción **goto** que indicara una etiqueta destino específica, desde esta etiqueta se continua con la ejecución del programa.

```
//Ejemplo de salto no condicional
    goto L1
    print("%c", 64); //código inalcanzable
L1:
    T2 = 100 + 5
```

5.7.2 Saltos condicionales

El formato de los saltos condicionales utilizará instrucciones IF del lenguaje C donde se realizará un salto a una etiqueta donde se encuentre el código a ejecutar si la condición es verdadera, seguida de otro salto a una etiqueta donde están las instrucciones si la condición no se cumple.

Las instrucciones IF tendrán como condición una expresión relacional, dichas expresiones se definen en la siguiente tabla:

Operación	Símbolo	C
Menor que	*<*	*4<5*
Mayor que	*>*	*10>100*
Menor o igual que	*<=*	*5 <= 5*
Mayor o igual que	*>=*	*T10 >= 50*
Igual que	*==*	*T11 != t245*
Diferente que	*!=*	*T120 != 4*

```
//Ejemplo de saltos condiciones

    If (10 == 10) goto L1;
    goto L2;

L1:
    //código si la condición es verdadera

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

Consideraciones:

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

```
//Entrada código alto nivel
wrileln(1+2*5);

//Salida código 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 llamada para continuar con las siguientes instrucciones.

```
//Definición de métodos

Void function x () {
     Goto L0;
     Print("%d", 100);
  L0:
     Return;
}
```

Consideraciones:

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

5.10 Llamada a métodos

Esta instrucción nos permite invocar a los métodos.

```
//Llamada a métodos
Identificador();

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

Void main(){
   Funcion1(); //se llama a la función 1 desde el método main
   Return();
}

Void Funcion1() {
   Print("%d",100);
  Return;
}
```

- Al realizar la llamada a función 1 el flujo cambia y se inician a ejecutar las instrucciones del método "funcion1".
- Al finalizar el método se regresa al punto donde fue llamada y sigue ejecutando las siguientes instrucciones.

5.11 Printf

Su función principal es imprimir en consola un valor, el primer parámetro que recibe la función es el formato del valor a imprimir, y el segundo es el valor mismo.

La siguiente tabla lista los parámetros permitidos para el proyecto.

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

```
//instrucciones de impresión
Printf("%d", (int)900); // imprime 900
Printf("%c", (char)65); // imprime A
Printf("%f", (float)65.4); //imprime 65.4
```

5.12 Estructuras en tiempo de ejecución.

El proceso de compilación genera el código intermedio, el cual se ejecutará en un compilador separado.

En las interpretaciones intermedias no existen, cadenas, operaciones complejas, llamadas a métodos con parámetros y otras funciones que si están presentes en los lenguajes de alto nivel.

Para el proyecto se implementarán 2 estructuras de una sola dimensión y llevarán los siguientes nombres:

- Stack (pila)
- Heap (Montículo)

Estas estructuras se utilizan para guardar los valores que sean necesarios dentro de la ejecución.

5.12.1 STACK

También conocido como pila de ejecución, es una estructura que se utiliza en código intermedio para guardar los valores de las variables locales, así como también los parámetros y el valor de retorno de las funciones en alto nivel.

Cuando un nuevo método es llamado a ejecución se le debe asignar un espacio de memoria en el stack, que utiliza para guardar sus variables locales, parámetros y retorno. El tamaño de este espacio es igual al tamaño del método.

Para lograr acceder a los valores dentro de la estructura STACK se utilizará una variable llamada "Stack Pointer", que en este proyecto se identifica con el nombre "**SP**", este valor va cambiando conforme se ejecute el programa, y su manejo debe ser cuidadoso para no corromper espacios de memoria ajenos al método que se está ejecutando.

El "apuntador" se identifica con las letras **SP** y tendrá asignada la dirección de memoria del STACK donde inicia el ámbito actual, su asignación se realizará exactamente igual que a como en los terminales.

```
Int SP;
SP = SP + 0
```

Consideraciones:

- *El stack debe reutilizar espacios de memoria cuando estos ya no son necesarios en la ejecución, se recomienda un buen control de su apuntador SP.*

5.12.2 Heap

También conocido como montículo, es una estructura de control del entorno de ejecucioooon encargada de guardas las referencias a las cadenas, arreglos y estructuras. Esta estructura también cuenta con un "apuntador" que se identifica con el nombre **"HP".**

A diferencia del apuntador **SP**, este apuntador no decrece, sino que sigue aumentando su valor, su función es brindar la siguiente posición de memoria libre dentro del heap.

```
//Apuntador del heap
Int HP = 0;
HP¨= HP + 1;
```

Consideraciones:

- *Si en el heap se guardan los datos de una cadena, cada espacio debe ocuparlo únicamente un carácter representado por su código ASCII.*
- *El heap crece indefinidamente, nunca reutiliza espacios de memoria.*

5.12.3 Acceso y asignación a estructuras en tiempo de ejecución

Para realizar las asignaciones y el acceso a estas estructuras, se debe respetar el formato de código de 3 direcciones, por lo cual se imponen las siguientes restricciones:

- La asignación a las estructuras se debe realizar por medio de un temporal o un valor puntual, no es permitido el uso de operaciones 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, por ejemplo "stack[0] = heap[100]".

```
//Asignación
Heap[HP] = T1;

<código>

Stack[T2] = 150;

//Acceso
T10 = Heap[T10];
T20 = Stack[t150];
```

Consideraciones

- *Tanto como las variables globales y los arreglos pueden guardarse en el STACK o en el HEAP, la decisión de donde se guardará esta información quedará a discreción del estudiante, siempre y cuando realice un uso valido y adecuado de las estructuras.*

5.13 Encabezado

En esta sección se definiran todas las variables y estructuras a utilizar. Únicamente en este sección se permite el uso de declaraciones en C, no es permitido realizar declaraciones adentro de métodos.

La estructura del encabezado es la siguiente:

```
#include <stdio.h> //importar para el uso de printf

Float Heap[100000]; //estructura heap
Float Stack[100000]; //estructura stack

Float SP; //puntero Stack pointer
Float HP; //puntero Heap pointer

Float T1, T2, T3; //declaración de temporales
```

Consideraciones:

- *No es permitido el uso de otras librerías ajenas a "stdio.h".*
- *Todas las declaraciones de temporales se deben encontrar en el encabezado.*

5.14 Método Main

Este es el método donde iniciara la 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:** considera la información de un bloque básico para realizar la optimización.
- **A nivel global:** considera 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 a nivel local serán los siguientes:

- Eliminación de instrucciones redundantes de carga y almacenamiento.
- Eliminación de código muerto.
- Simplificación algebraica y reducción por fuerza.

Asociadas a los tipos de transformación, se tendrán 16 reglas, las cuales se detallan a continuación:

6.1 Eliminación de código muerto

También llamado eliminación de código inalcanzable. Consiste en eliminar las instrucciones que nunca serán utilizadas. Las reglas aplicables son las siguientes:

6.1.1 Regla 1

Esta regla sostiene que si existe un salto condicional Lx y una etiqueta Lx; todo el código que se encuentre entre ellos podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

Ejemplo	**Optimización**
goto L1; <instrucciones> L1: T3 = T1 + T3;	goto L1; L1: T3 = T1 + T3;

6.1.2 Regla 2

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

Ejemplo	**Optimización**
If (4 == 4) goto L1; goto L2; L1: <instrucciones_L1> L2: <instrucciones_L2>	If (4 != 4) goto L2; <instrucciones_L1> L2: <instrucciones_L2>

6.1.3 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;	goto L1;

6.1.4 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 (4 == 1) goto L1; goto L2;	goto L2;

6.2 Eliminación de instrucciones redundantes de carga y almacenamiento

6.2.1 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**
T3 = T2; <instrucciones> T2 = T3;	T3 = T2; <instrucciones>

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.

6.3.1 Regla 6

Ejemplo	**Optimización**
T1 = T1 + 0;	// Se elimina la instrucción

6.3.2 Regla 7

Ejemplo	Optimización
T1 = T1 - 0;	// Se elimina la instrucción

6.3.3 Regla 8

Ejemplo	Optimización
T1 = T1 * 1;	// Se elimina la instrucción

6.3.4 Regla 9

Ejemplo	Optimización
T1 = T1 / 1;	// Se elimina la instrucción

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.

6.3.5 Regla 10

Ejemplo	Optimización
T1 = T2 + 1;	T1 = T2;

6.3.6 Regla 11

Ejemplo	Optimización
T1 = T2 - 1;	T1 = T2;

6.3.7 Regla 12

Ejemplo	Optimización
T1 = T2 * 1;	T1 = T2;

6.3.8 Regla 13

Ejemplo	Optimización
T1 = T2 / 1;	T1 = T2;

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.

6.3.9 Regla 14

Ejemplo	Optimización
T1 = T2 * 2;	T1 = T2 + T2;

6.3.10 Regla 15

Ejemplo	Optimización
T1 = T2 * 0;	T1 = 0;

6.3.11 Regla 16

Ejemplo	Optimización
T1 = 0 / T2;	T1 = 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.

Se verificará en el reporte de la tabla de símbolos si se guardó la información necesaria para la generación correcta del código en 3 direcciones, la información contiene, pero no se limita a:

- Posición de variable en el ambiente.
- Procedimiento o función padre (en el caso de que la función sea anidada).
- Ambiente de la variable, constante, procedimiento o función.
- Numero de parámetros de una función o procedimiento.

7.2. Reporte de errores

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

7.2.1 Tipos de errores

Los tipos de errores que deberán manejar son los siguientes:

- Léxicos
- Sintácticos
- Semánticos

7.2.2 Contenido de tabla de errores

- Línea: línea donde se encuentra el error.
- Columna: columna donde se encuentra el error.
- Tipo de error: identifica el tipo de error encontrado (léxico, sintáctico o semántico).
- Descripción: explicación concisa de la razón del error.
- Ámbito: si sucedió en el ámbito global o indicar el nombre de la función en caso aplique.

Consideraciones

- Queda a discreción del estudiante si presentarlos en reportes separados o en un mismo reporte especificando en qué fase ocurrió el mismo.

7.3 Reporte de Optimización

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

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

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, se calificará hasta el último commit antes de la fecha y hora límite.

- Código fuente publicado en un repositorio de github.
- Enlace al repositorio y permisos a los auxiliares para poder acceder.

8.2. Restricciones

- La herramienta para generar los analizadores del proyecto será Irony. La documentación se encuentra en el siguiente enlace https://archive.codeplex.com/?p=irony
- Para la generación de graficas se debe de utilizar Graphiz.
- Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ciencias y Sistemas los involucrados.
- ***Para el desarrollo del proyecto se deberá de crear una carpeta en esta ruta C:\compiladores2, y aquí deben de colocar el DLL de Irony (esta se debe de llamar Irony.dll) para la referencia del proyecto de Visual Studio, adicional agregar que su IDE guarde en esta ruta las imágenes que generan las gráficas y archivos de reportes, esto para tener un estándar y evitar problemas en la calificación por rutas quemadas en el código, ya que no se podrá modificar nada el día de la calificación.***
- ***Se debe de utilizar Visual Studio 2019 como IDE de desarrollo y se debe utilizar una aplicación de Windows Forms App con .Net Core con el SDK 3.1***
- El desarrollo y entrega del proyecto es individual.

8.3. Consideraciones

- 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.
- El repositorio únicamente debe contener el código fuente empleado para el desarrollo, no deben existir archivos pdf o docx.
- El sistema operativo para utilizar es libre.
- El lenguaje está basado en Pascal, 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.
- Pascal en línea https://www.onlinegdb.com/online_pascal_compiler
- Se van a publicar archivos de prueba en el siguiente repositorio: https://github.com/CrisAlva25/Compi2_1S2021

8.4. Calificación

- La calificación se realizará dentro de la máquina de los auxiliares, ya que es muy importante que tengan la última versión de su proyecto subida a github y las rutas definidas anteriormente.
- Se tendrá un máximo de 30 minutos por estudiante para calificar el proyecto.
- 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.
- Si una función del programa ya ha sido calificada, esta no puede ser penalizada si en otra sección la función falla o es errónea.
- Los archivos de entrada podrán ser modificados solamente antes de iniciar la calificación eliminando funcionalidades que el estudiante indique que no desarrolló.
- 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.

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 establecidas invalidará el proyecto, por lo que se calificará hasta el último commit dentro de la fecha válida.
- ***No habrá prorroga***
- Fecha de entrega:

Lunes 3 de mayo hasta las 23:59 PM