Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2
Escuela Vacaciones Primer Semestre 2021
Catedráticos: Ing. Luis Espino
Tutores académicos: Haroldo Arias

TytusX

1. 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.

1.1 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 o estructuras generadas por el estudiante para simular esto.*
- *Para facilidad, se recomienda trabajar todas las variables de tipo float.*

1.2 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
```

1.3 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:
```

1.4 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
```

1.5 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 */
```

1.6 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.*

1.7 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.

1.7.1 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
```

1.7.2 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.*

1.8 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);
```

1.9 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".

1.10 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.

1.11 Printf

Su función principal es imprimir en consola los resultados de las consultas generadas por XPATH y XQUERY, 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
```

1.12 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, dichas estructuras no son obligatorias en este proyecto, sin embargo, si deben de definir únicamente listas globales para el manejo de almacenamiento de información en memoria.

1.12.1 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.*

1.12.2 Heap

También conocido como montículo, es una estructura de control del entorno de ejecución 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.*

1.12.3 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.*

1.13 Encabezado

En esta sección se definirán todas las variables y estructuras a utilizar. Únicamente en esta 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.*

1.14 Método Main

Este es el método donde iniciara la ejecución del código traducido. Su estructura es la siguiente:

```
 Void main(){
//instrucciones
Return;
}
```

2. 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:

2.1 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:

2.1.1 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;

2.1.2 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>

2.1.3 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;

2.1.4 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;

2.2 Eliminación de instrucciones redundantes de carga y almacenamiento

2.2.1 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>

2.3 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.

2.3.1 Regla 6

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

2.3.2 Regla 7

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.

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

2.3.3 Regla 8

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

2.3.3 Regla 9

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

2.3.5 Regla 10

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

2.3.6 Regla 11

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

2.3.7 Regla 12

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

2.3.8 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;