



Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Físico Químicas y Naturales

Departamento de Computación

Taller de Diseño de Software

Proyecto Tercera entrega

Campregher Bruno
Machuca Franco
Natali Valentino

Generador Código Intermedio

1. Introducción

En esta etapa del compilador se realiza la **traducción del árbol sintáctico** obtenido en las fases anteriores a una **representación intermedia del programa**, conocida como **código intermedio**.

El generador de código intermedio toma como entrada la **representación del programa validado semánticamente** y produce una **secuencia de instrucciones** intermedias en forma de cuádruplos.

Cada cuádrupla representa una operación elemental mediante cuatro campos: el operador, los operandos de entrada y el resultado.

Repositorio del proyecto

El **código fuente completo** del proyecto está disponible en nuestro **repositorio de GitHub**.

La rama Entrega-08/10 contiene la versión correspondiente a la **entrega actual**.

2. Representación intermedia

En esta etapa, el código intermedio se representa mediante **cuádruplos**, una estructura que permite almacenar de forma explícita cada operación junto con sus operandos y resultado.

Cada cuádrupla describe una instrucción elemental del programa, facilitando la posterior generación de código objeto y posibles optimizaciones.

Para la implementación, se definieron dos estructuras principales:

- Cuadruplo, que representa una instrucción individual.

```
typedef struct Cuadruplo
{
    Tipo_Operador op;
    Simbolo *arg1;
    Simbolo *arg2;
    Simbolo *resultado;
} Cuadruplo;
```

La estructura **Tipo_Operador** define los posibles **tipos de operaciones** que pueden aparecer en las cuádruplos, incluyendo operaciones aritméticas (ADD, MINUS, MULT, DIV, MOD), lógicas (AND, OR, NOT), comparaciones (LT, GT, COMP), saltos y control de flujo (IF_FALSE,

JMP, JMPC, END_IF, END WHILE), y manejo de funciones (CALL, RET, START_FUN, END_FUN, PARAM).

- Instrucciones, que permite enlazar múltiples cuádruplos formando una lista que representa el programa completo en su forma intermedia.

```
typedef struct Instrucciones
{
    Cuadruplo *expr;
    struct Instrucciones *next;
} Instrucciones;
```

La estructura **Cuadruplo** contiene cuatro campos:

- op: indica el tipo de operación que se realizará (por ejemplo, suma, resta, salto condicional, asignación, etc.).
- arg1 y arg2: representan los operandos sobre los cuales actúa la operación.
- resultado: almacena el símbolo donde se guardará el resultado de la operación.

Por su parte, la estructura Instrucciones actúa como una **lista enlazada de cuádruplos**, permitiendo mantener el orden de ejecución de las operaciones generadas durante la traducción intermedia.

De esta forma, el programa completo puede representarse como una secuencia de cuádruplos encadenadas, donde cada nodo refleja una operación del flujo de ejecución.

3. Generación de cuádruplos

El proceso de **generación de código intermedio** consiste en recorrer la representación semánticamente validada del programa (el árbol sintáctico y semántico) y traducir cada constructo del lenguaje fuente en una o más **cuádruplos** equivalentes.

A continuación se describen los casos más representativos según la implementación:

Tipo de operación	arg1	arg2	resultado
Aritmética (Ejemplo: ADD)	Primer operando	Segundo operando	Variable temporal
Asignación (MOV)	Valor a asignar		Variable destino
Condicional (IF_FALSE)	Resultado de la expresión		Etiqueta donde salta si la condición es false
Parámetro actual (PARAM)	Expresión del parámetro		
Llamada a función (CALL)	Cantidad de parámetros		Símbolo asociado al nombre de la función

Ejemplo

Código fuente	Código intermedio
<pre>x = 3 > 2; if (x) then { y = 5; } else { y = 7; }</pre>	<pre>GT 3 2 t0 MOV t0 x IF_FALSE x L0 MOV 5 y JMP L1 L0: MOV 7 y L1:</pre>

4. Pruebas y verificación

No se realizaron nuevos tests específicos para la etapa de generación de código intermedio. Para verificar su funcionamiento se utilizaron los tests desarrollados durante la etapa anterior de análisis semántico.

5. Ejecución del compilador

El archivo principal del proyecto (main.c) es reutilizado de la entrega anterior. Se incorpora la etapa de generación de código intermedio, la cual se ejecuta exclusivamente una vez que el programa ha superado satisfactoriamente la verificación semántica, garantizando que el código intermedio se genere a partir de una representación previamente validada.

Uso del programa

```
programa [-target etapa] archivo.ctds
```

Donde:

- `-target scan` ejecuta únicamente el analizador léxico y genera como salida un archivo con extensión `.lex`.
- `-target parse` ejecuta el analizador sintáctico (por defecto si no se especifica `-target`) y genera un archivo con extensión `.sint`.
- `-target ast` realiza la creación del archivo `dot` en base al árbol, aparte de todas las etapas anteriores (`scan` y `parse`).
- `-target sem`: ejecuta el análisis semántico del programa. En esta etapa se generan los archivos `.lex` y `.sint` como en las fases previas, y adicionalmente un archivo con extensión `.sem` que contiene todos los errores semánticos detectados, gracias al uso del módulo de manejo de errores.
- `-target ci`: ejecuta el análisis semántico del programa. En esta etapa se generan los archivos `.lex`, `.sint` y `.sem`, y adicionalmente un archivo con extensión `.ci` que contiene el código intermedio.
- `archivo.ctds` corresponde al programa fuente a compilar. Debe finalizar con la extensión `.ctds`.

Manejo de archivos

- El compilador valida que el archivo fuente sea correcto (no debe comenzar con `-` y debe terminar en `.ctds`).
- En caso de error al abrir el archivo fuente o crear el archivo de salida, se informa al usuario mediante un mensaje descriptivo.
- El nombre del archivo de salida se deriva automáticamente del archivo de entrada, reemplazando la extensión `.ctds` por `.lex`, `.sint` o `.sem` según corresponda.

Asunciones realizadas

- El parámetro `-target` solo admite los valores `scan`, `parse`, `ast`, `sem` y `ci`.
- Si no se especifica `-target`, la etapa ejecutada por defecto es `sem`.
- Se asume que el archivo de entrada existe y es accesible para lectura.