Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)

Patito - entrega #5

Profesores:

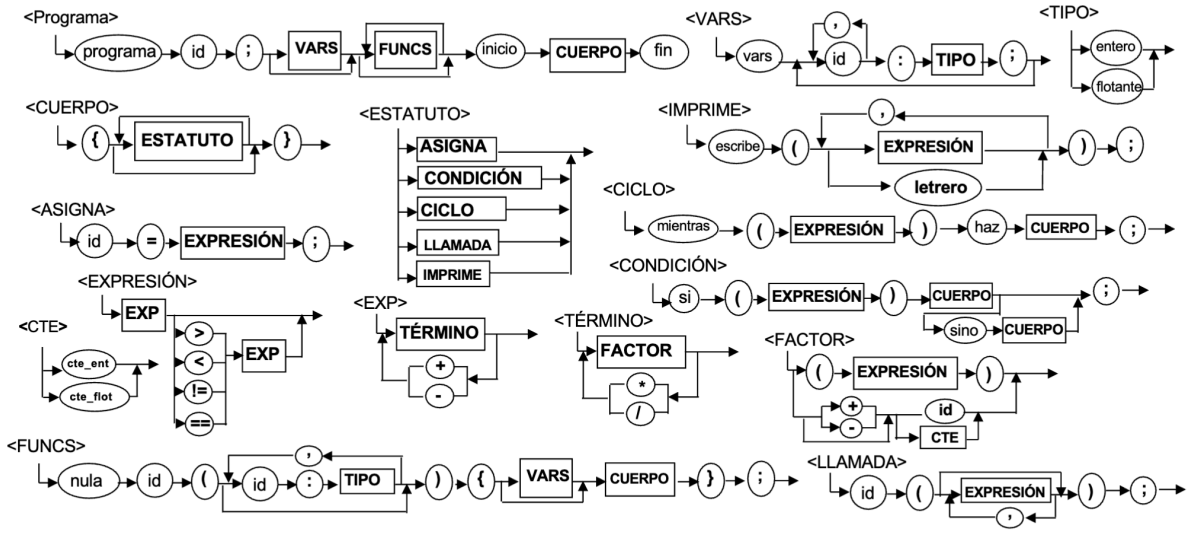
Elda G Quiroga

Alumnos:

Nicolás Aguirre Villafañe A00832772

a 21 de Noviembre del 2024

Descripción gráfica del mini-lenguaje *patito*:



Ya que lo haré en C++ con la libreria de ANTLR, pondré todo de una vez en ANTLR.

- ***Diseñar las Expresiones Regulares que representan a los diferentes elementos de léxico que ahí aparecen:***

Los únicos elementos que necesitan expresiones regulares son los siguientes:

```
// Identificadores y constantes
ID: [a-zA-Z_][a-zA-Z0-9_]*;
CTE_ENT: [0-9]+;
CTE_FLOT: [0-9]+'.'[0-9]+;
LETRERO_VALOR: '"' .*? '"'; // Para cadenas de texto (letrero)
// Ignorar espacios en blanco
WS: [ \t\r\n]+ -> skip;
```

- ***Listar todos los Tokens que serán reconocidos por el lenguaje:***

En mi léxico cambie algunas palabras a inglés para mejor comprensión y también eliminé la palabra ’haz’ ya que me parecía redundante debido a que después de la evaluación de un if o un while es obvio que lo que sigue después de la llave sería el “haz”.

```
// Palabras clave
PROGRAM: 'program';
START: 'start';
END: 'end';
VARS: 'vars';
INT: 'int';
FLOAT: 'float';
PRINT: 'print';
WHILE: 'while';
VOID: 'void';
IF: 'if';
ELSE: 'else';
// Operadores y símbolos
```

PUNTOYCOMA: ';';
COMA: ',';
DOSPUNTOS: ':';
LLAVEIZQ: '{';
LLAVEDER: '}';
PARENIZQ: '(';
PARENDER: ')';
IGUAL: '=';
MAYORQUE: '>';
MENORQUE: '<';
DIFERENTE: '!=';
IGUALIGUAL: '==';
MAS: '+';
MENOS: '-';
MULT: '';*
DIV: '/';

- ***Diseñar las reglas gramaticales (Context Free Grammar) equivalentes a los diagramas.***

En mi gramática reestructuré el "cuerpo" ya que había doble llaves cuando se creaba una función y luego adentro llevaba un cuerpo, por lo tanto para mejor fluidez quité las llaves del cuerpo y las puse manualmente antes y después del cuerpo cuando se llama. También las vars la estructuré para que lleve llaves.

programa: PROGRAM ID PUNTOYCOMA vars? funcs START LLAVEIZQ cuerpo LLAVEDER END;*
vars: VARS LLAVEIZQ decl_var+ LLAVEDER;
decl_var: ID (COMA ID) DOSPUNTOS tipo PUNTOYCOMA;*
tipo: INT | FLOAT;
funcs: VOID ID PARENIZQ params? PARENDER LLAVEIZQ vars? cuerpo LLAVEDER PUNTOYCOMA;
params: ID DOSPUNTOS tipo (COMA ID DOSPUNTOS tipo);*
cuerpo: estatuto;*
estatuto: asigna | condicion | ciclo | llamada | imprime;
asigna: ID IGUAL expresion PUNTOYCOMA;
expresion: exp ((MAYORQUE | MENORQUE | DIFERENTE | IGUALIGUAL) exp)?;
exp: termino ((MAS | MENOS) termino);*
termino: factor ((MULT | DIV) factor);*
factor: PARENIZQ expresion PARENDER | MENOS? (ID | cte);
cte: CTE_ENT | CTE_FLOT;
condicion: IF PARENIZQ expresion PARENDER LLAVEIZQ cuerpo LLAVEDER (ELSE LLAVEIZQ cuerpo LLAVEDER)? PUNTOYCOMA;
ciclo: WHILE PARENIZQ expresion PARENDER LLAVEIZQ cuerpo LLAVEDER PUNTOYCOMA;

llamada: ID PARENIZQ (expresion (COMA expresion))? PARENDER PUNTOYCOMA;*
imprime: PRINT PARENIZQ impr (COMA impr) PARENDER PUNTOYCOMA;*
impr: expresion | LETRERO_VALOR;

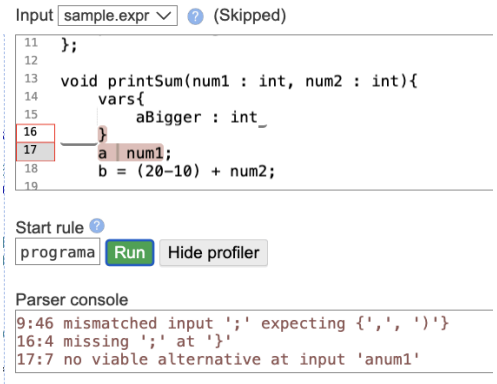
- ***Diseñar un Test-Plan y comprobar que funciona adecuadamente:***

Diseñé esta tabla con diferentes pruebas de la entrada y el resultado esperado, posteriormente lo testee en el ANTLR lab en el cual debo poner el léxico y la gramática y luego ya puedo poner código de mi lenguaje para comprobar el funcionamiento.

Prueba	Entrada	Resultado Esperado
Declaración de variables	a, b, c: int;	Aceptada
Expresión aritmética	a = (20 - 10) + b;	Aceptada
Condicional con else	if (a < b) { ... } else { ... };	Aceptada
Condicional sin punto y coma	if (a < b) { ... }	Rechazada
Ciclo While	while (a == 1) { ... };	Aceptada
Llamada a print	print("Texto:", num1);	Aceptada
Expresión sin punto y coma	a = 10	Rechazada
Uso de variable no declarada	x = 10;	Rechazada
Función válida sin parámetros	void printMessage() { print("Hello World"); };	Aceptada
Función válida con parámetros	void printSum(a: int, b: int) { print(a + b); };	Aceptada
Función mal definida (sin void)	printSum(a: int, b: int) { print(a + b); };	Rechazada
Función mal definida (tipo de parametros)	printSum(int a, int b) { print(a + b); };	Rechazada
Parámetros en llamada a función	printNums(a, b);	Aceptada
Llamada a función sin parámetros	printMessage();	Aceptada
Llamada a función con pocos args	printNums(a);	Aceptada pero debería ser rechazada.

Comparación válida en condicional	if (a != b) { a = b; };	Aceptada
Comparación con operador inválido =>	if (a => b) { a = b; };	Rechazada
Ciclo sin condición válida	while () { a = a - 1; };	Rechazada
Imprimir número y cadena	print("Resultado: ", 10);	Aceptada
Uso de operadores no permitidos	a %% b;	Rechazada
Cuerpo de programa vacío	inicio { } fin	Aceptada
Función con parámetros de tipos diferentes	void test(x: int, y: float) { print(x + y); }	Aceptada
Declaración de múltiples variables con tipos distintos	vars{ a, b: int; c, d: float; }	Aceptada
Declaración sin tipo	vars{ a, b; }	Rechazada
Imprimir múltiples expresiones	print(a, b + 5, "texto", c + d);	Aceptada
Expresión sin cierre de paréntesis	print((a + b;	Rechazada
Declaración con coma extra	a, b, : int;	Rechazada

Adjunto imagen de cómo de ve testeado en el ANTLR lab:



- ***Diseñar tabla de consideraciones semánticas (cubo semántico):***

Para mi cubo semántico creé una clase llamada 'SemanticCube' en donde tendré un map/diccionario donde la llave será la combinación de operaciones y el valor será el tipo a retornar. La llave será un string donde se concatena el tipo de la variable1, el operador, y la

variable2. Tendré un enumerador de tipos donde me dirá el tipo que és (un número representando el tipo):

```
C/C++
enum Type { INT, FLOAT, VOID, ERROR, BOOL, STRING};
```

Luego al concatenar por ejemplo dos integers para una suma, obtendré el string ‘0+0’ y esa string será la llave de mi diccionario. El diccionario luce más o menos de esta manera:

```
C/C++
std::unordered_map<std::string, Type> cube;
cube["0+0"] = INT;
cube["0+1"] = FLOAT;
cube["1+0"] = FLOAT;
cube["1+1"] = FLOAT;
...
```

Estas combinaciones están preestablecidas en el código con los demás operadores como resta, multiplicación, división, mayor que, menor que, igual que y diferente que.
Después, para encontrar el tipo resultante solo se hará una búsqueda en el diccionario para obtener el tipo resultante de esa operación, en este caso la llave ‘0+0’ nos da un INT.
Este es el método que nos da el tipo resultante:

```
C/C++
Type SemanticCube::getResultType(Type type1, Type type2, const std::string
&op) {
    std::string key = std::to_string(type1) + op + std::to_string(type2);
    return cube.count(key) ? cube[key] : ERROR;
}
```

- ***Implementación del directorio de funciones y tablas de variables:***

Para implementar el **Directorio de Funciones** y las **Tablas de Variables** en el compilador, ambos se diseñan para organizar funciones y variables, manteniendo el control de los alcances (scopes) y validaciones durante la compilación.

Directorio de Funciones:
El Directorio de Funciones es una tabla global que almacena:

1. Nombre y tipo de retorno de cada función.
2. Lista de parámetros (nombre y tipo).
3. Tabla de Variables específica para variables locales y parámetros de la función.
4. Dirección de inicio de la función en memoria.

El directorio permite al compilador verificar si una función existe, validar tipos de retorno y parámetros en las llamadas, y acceder a las variables locales de esa función.

Ejemplo en el código:

```C/C++
struct FunctionInfo {
    std::string name;
    Type returnType;
    std::vector<VariableInfo> parametersTable;
    VariableTable variableTable;
    int startAddress = -1;
    int numVars = 0;
    int numParams = 0;
};

class FunctionDirectory {
public:
    bool addFunction(const std::string &name, Type returnType);
    FunctionInfo* getFunction(const std::string &name);

private:
    std::unordered_map<std::string, FunctionInfo> directory;
};
```

Tabla de Variables:
Cada función tiene su propia Tabla de Variables que almacena:

1. Nombre y tipo de cada variable en el ámbito de esa función.
2. Dirección de memoria, útil para la generación de código.

La Tabla de Variables permite manejar variables locales, evitar duplicados en un mismo ámbito, y buscar variables en el alcance adecuado.
Ejemplo en el código:

```C/C++
struct VariableInfo {
    std::string name;
    Type type;
    int memoryAddress = -1;
};

class VariableTable {
public:
    bool addVariable(const std::string &name, Type type, int memoryAddress);
    VariableInfo *getVariableInfo(const std::string &name);
    std::unordered_map<std::string, VariableInfo> *getVariables();
    bool setMemoryAddress(const std::string &name, int memoryAddress);

private:
    std::unordered_map<std::string, VariableInfo> variables;
};
```

Interacción y Validaciones:

1. Cada función en el Directorio de Funciones tiene su propia Tabla de Variables para gestionar variables locales.
2. Al declarar variables, se almacenan en la Tabla de Variables de su función.
3. Al llamar a funciones, el compilador verifica que el tipo de retorno y parámetros sean correctos.
4. Validaciones como verificar duplicados o comprobar tipos se facilitan con estas estructuras.

Este diseño permite organizar funciones y variables de manera modular, manteniendo control sobre el alcance y asegurando la corrección semántica del programa.

- ***Estableciendo los puntos neurálgicos de tabla de variables y directorio de funciones para el programa:***

Para llenar el **Directorio de Funciones** y las **Tablas de Variables** en el compilador, se identifican los siguientes puntos neurálgicos (momentos clave) para realizar las validaciones y asegurar una compilación semánticamente correcta:

1. **Definición de Funciones**
    - **Creación de una entrada** en el Directorio de Funciones al detectar una definición de función.
    - **Validación de duplicados:** verificar si la función ya existe en el directorio para evitar redefiniciones.
    - **Inicialización de la Tabla de Variables local** asociada a la función.
2. **Declaración de Parámetros de Función**
    - **Al añadir parámetros a la Tabla de Variables** local de la función, se validan nombres duplicados.
    - Los parámetros se registran en el orden en que aparecen, asegurando consistencia en las llamadas.
3. **Declaración de Variables (Locales y Globales)**
    - **Validación de duplicados en el mismo ámbito:** al declarar variables, verificar que no estén ya en la Tabla de Variables del ámbito actual (local o global).
    - **Asignación del tipo** y otras propiedades a cada variable en su Tabla de Variables correspondiente (local o global).
4. **Uso de Variables**
    - **Chequeo de existencia en el ámbito actual y superiores:** al acceder a una variable, verificar que esté declarada en el ámbito adecuado o en un ámbito superior, evitando referencias a variables no declaradas.
5. **Llamadas a Funciones**
    - **Verificación de existencia** en el Directorio de Funciones.
    - **Validación de tipos y cantidad de parámetros** contra la definición en el Directorio de Funciones, asegurando correspondencia entre la llamada y la definición.

6. **Finalización de una Función o Bloque**
    - **Liberación de variables locales:** al salir de una función o bloque, liberar su Tabla de Variables para mantener el alcance y evitar accesos fuera del ámbito.

Estos puntos neurálgicos son clave para llenar correctamente las tablas y garantizar una estructura semántica sólida en el compilador, previniendo errores comunes como duplicados y referencias inválidas.

- ***Implementación de Pilas y Cola para la Generación de Cuádruplos:***

Para generar los cuádruplos se implementaron pilas y una cola para gestionar las operaciones necesarias durante la traducción de expresiones aritméticas, relacionales, y la representación de estatutos lineales y condicionales en cuádruplos.

Pilas

Se utilizaron varias pilas para gestionar los distintos componentes de las expresiones y operaciones:

- **Pila de Operadores:** Esta pila almacena los operadores aritméticos o relacionales (como +, -, *, /, ==, >, etc.) que se encuentran en las expresiones. Los operadores se agregan a medida que se leen y se resuelven de acuerdo con la precedencia y las reglas de la gramática.
- **Pila de Operandos:** En esta pila se almacenan los operandos (como variables, constantes y resultados intermedios) que participan en las expresiones. Los operandos se empujan a la pila cuando se procesan los elementos de la expresión.
- **Pila de Tipos:** Para la verificación de tipos y la correcta generación de los cuádruplos, se usa una pila que guarda los tipos de las variables y los resultados de las operaciones. Esta pila es necesaria para asegurar que las operaciones se realicen entre tipos compatibles.
- **Pila de Saltos:** Se emplea una pila de saltos para gestionar las instrucciones de salto en estructuras condicionales y bucles. Esta pila es útil para guardar las direcciones de salto pendientes, como cuando se realiza una comparación en un condicional o se encuentra una instrucción de goto.

Cola de Cuádruplos

- **Cola de Cuádruplos:** Los cuádruplos son representaciones intermedias de las instrucciones de la máquina. Cada cuádruplo contiene un operador y dos operandos, así como el resultado de la operación. Los cuádruplos se generan durante el análisis de las expresiones y los estatutos, y se almacenan en una cola para su posterior uso o generación de código. La cola permite mantener el orden de las operaciones a medida que se procesan.

Algoritmos de Traducción a Cuádruplos

Se implementaron los algoritmos de traducción para generar cuádruplos a partir de las expresiones y los estatutos en el lenguaje. Los algoritmos de traducción cubren las siguientes áreas:

1. **Expresiones Aritméticas:**
    - Las expresiones aritméticas se procesan utilizando las pilas de operadores y operandos. Por ejemplo, en una expresión como ***a + b * c***, primero se evalúa la multiplicación ***b * c*** (que se representa como un cuádruplo) y luego se evalúa la suma ***a + t0***, donde **t0** es el resultado de la multiplicación. El cuádruplo para cada operación se genera y se almacena en la cola de cuádruplos.
2. **Expresiones Relacionales:**
    - Las expresiones relacionales, como ***a == b***, se manejan de manera similar a las aritméticas, pero en este caso, se generan cuádruplos para comparar los operandos. El resultado de la comparación se almacena en una variable temporal y se utiliza en las instrucciones condicionales.
3. **Estatutos Lineales:**
    - Los estatutos lineales como asignaciones ***(x = y + z)*** y declaraciones se traducen a cuádruplos de manera directa, donde el operador de asignación se toma junto con los operandos y el resultado se almacena en una variable o en una temporal.
4. **Estatutos Condicionales:**
    - Los estatutos condicionales, como los ***if*** y ***while***, requieren el uso de la pila de saltos. En estos casos, se generan cuádruplos que gestionan las comparaciones y las instrucciones de salto. Por ejemplo, en un if como ***if (a > b)***, se genera un cuádruplo para la comparación y otro cuádruplo para el salto condicional si la comparación es verdadera.

Ejemplo de Cuádruplos

Considerando la expresión aritmética ***a + b * c***, el proceso sería el siguiente:

1. **Paso 1:** Se empujan los operandos ***a***, ***b*** y ***c*** a la pila de operandos.
2. **Paso 2:** Se empujan los operadores + y * a la pila de operadores.
3. **Paso 3:** Se evalúa ***b * c*** primero, generando un cuádruplo: ***(*, b, c, t0)***.
4. **Paso 4:** Luego, se evalúa ***a + t0***, generando otro cuádruplo: ***(+, a, t0, t1)***.
5. **Paso 5:** Los cuádruplos se almacenan en la cola de cuádruplos y se procesan de manera secuencial.

- ***Ejemplo de cuadruplos despues de ejecución:***

Aquí muestro un ejemplo de cuadruplos después de ejecutar un programa simple como el siguiente:

```C/C++
program easy;
vars{ i : int; }
start {
	i = 20+10*9;
	print("la multiplicacion da como resultado: ", i);
} end
```

Este programa genera como resultado los siguientes cuadruplos:

```Unset
0: GOTO nil nil 1
1: * 10 9 t0
2: + 20 t0 t1
3: = t1 nil i
4: PRINT nil nil "la multiplicacion da como resultado: "
5: PRINT nil nil i
6: ENDPRINT nil nil nil
7: HALT nil nil nil
```

- ***Puntos Neurálgicos de la Implementación***

1. **Inicio del Programa**
    - Punto Neurálgico: Crear una función global o de "main" en el directorio de funciones para almacenar las variables globales.
    - Descripción: Se inicializa la estructura del programa, incluyendo las pilas (operadores, operandos, tipos, saltos) y la cola para los cuádruplos.

2. **Declaración de Variables**
    - Punto Neurálgico: Agregar cada variable al directorio de funciones, actualizando la tabla de variables correspondiente.
    - Descripción: Cuando se declara una variable, se añade su tipo y nombre en la tabla de variables de la función actual. Si ya existe, se lanza un error de duplicidad.
3. **Procesamiento de Operadores y Operandos**
    - Punto Neurálgico: Manejo de las pilas de operadores, operandos y tipos.
    - Descripción: Durante la evaluación de una expresión, cada operando se apila en la pila de operandos y su tipo en la pila de tipos. Los operadores se apilan en la pila de operadores y se resuelven según precedencia para generar cuádruplos.
4. **Generación de Cuádruplos para Expresiones Aritméticas**

- Punto Neurálgico: Creación de cuádruplos para operaciones aritméticas.
- Descripción: Al encontrar un operador de baja precedencia (como + o -), se extraen dos operandos y un operador para crear un cuádruplo con el operador, los dos operandos y el resultado.

5. **Generación de Cuádruplos para un print (Impresión)**
    - Punto Neurálgico: Creación de un cuádruplo de impresión.
    - Descripción: Cuando se encuentra una instrucción print, se toma el operando (expresión o variable) que se desea imprimir y se genera un cuádruplo de la forma (PRINT, -, -, valor).
6. **Manejo de Condicionales if**
    - Punto Neurálgico: Creación de cuádruplos para la evaluación de la condición y los saltos.
    - Descripción:
        - Primero, se evalúa la condición del if, creando un cuádruplo con el operador relacional.
        - Después de evaluar la condición, se crea un cuádruplo de salto falso (GOTOF) que salta si la condición es falsa. La dirección del salto se guarda en la pila de saltos.
        - Al cerrar el bloque del if, el salto se completa con la dirección correcta.
7. **Manejo de Bucles (while)**
    - Punto Neurálgico: Generación de cuádruplos para la condición y control de flujo del bucle.
    - Descripción:
        - Antes de la evaluación de la condición del while, se almacena la posición actual en la pila de saltos para regresar al inicio del bucle.
        - Se evalúa la condición y se crea un cuádruplo GOTOF para saltar al final del bucle si la condición es falsa.
        - Al final del bloque, se genera un cuádruplo GOTO que regresa al inicio del bucle, y se actualizan los cuádruplos de salto condicional.

8. **Asignaciones de Variables**
    - Punto Neurálgico: Generación de cuádruplos para asignaciones.
    - Descripción: En una instrucción de asignación, se evalúa la expresión en el lado derecho y se crea un cuádruplo de la forma (=, valorDerecha, -, variableIzquierda).

- ***Traducción de variables, constantes y temporales a dirección virtual***

En el compilador, se asignan Direcciones Virtuales a variables, constantes y temporales para optimizar el manejo de la memoria. La asignación de direcciones virtuales permite organizar espacios de memoria de manera lógica, separando por tipo y alcance, lo que evita conflictos entre variables locales, globales y temporales de distintos tipos de datos.

Diseño y Lógica:

1. **Definición de Rangos por Tipo y Alcance:** Los rangos de direcciones virtuales están segmentados por categorías: variables globales, locales, constantes y temporales. Cada tipo de dato (entero, flotante, etc.) tiene un rango designado dentro de estas categorías. Por ejemplo, las direcciones de enteros globales están asignadas de 1000 a 1999, los flotantes globales 2000 a 2999, etc. estableciendo un orden estructurado.

```C/C++
    // Global memory
    int globalIntBase = 1000, globalIntPointer = 1000;
    int globalFloatBase = 2000, globalFloatPointer = 2000;
    // Local memory
    int localIntBase = 3000, localIntPointer = 3000;
    int localFloatBase = 4000, localFloatPointer = 4000;
    // Temporary memory
    int tempIntBase = 5000, tempIntPointer = 5000;
    int tempFloatBase = 6000, tempFloatPointer = 6000;
    // Constant memory
    int constIntBase = 7000, constIntPointer = 7000;
    int constFloatBase = 8000, constFloatPointer = 8000;
    int constStringBase = 9000, constStringPointer = 9000;
```

2. **Asignación durante la Declaración:** Al declarar una variable, el compilador verifica su tipo y alcance para asignarle una dirección en el rango adecuado. La dirección se registra en la tabla de variables, acompañada del tipo de dato y el alcance.
3. **Constantes y Temporales:**
    a. **Las constantes** se almacenan en una tabla de constantes, evitando reasignaciones y permitiendo reutilización.
    b. **Los temporales** se asignan en tiempo de ejecución para almacenar resultados intermedios, liberándose cuando dejan de ser necesarios.

- ***Generación de Cuádruplos para el Estatuto Cíclico (while)***

La generación de cuádruplos para un ciclo while se centra en el manejo de la condición de entrada y el flujo de control que permite repetir el bloque de instrucciones.

Lógica de Generación:

1. **Inicio del Ciclo:** Antes de evaluar la condición, se marca la posición actual para facilitar el retorno al inicio del ciclo tras cada iteración.
2. **Evaluación de la condición:** Se genera un cuádruplo para evaluar la condición del ciclo. Si el resultado es falso, se utiliza un cuádruplo GOTOF para saltar al final del ciclo, indicando que no se debe ejecutar el bloque.
3. **Bloque del Ciclo:** Las instrucciones dentro del bloque se traducen a cuádruplos de acuerdo a sus operaciones, sin modificaciones especiales.

4. **Salto de Regreso:** Al concluir el bloque del ciclo, se genera un cuádruplo GOTO que redirige al inicio del ciclo, lo que permite reevaluar la condición. El salto del GOTOF al final del ciclo se actualiza con la dirección correcta.

- ***Ejemplo de cuadruplos para while despues de ejecución:***

Aquí muestro un ejemplo de cuadruplos después de ejecutar un programa simple como el siguiente:

```
C/C++
program easy;
vars{ i : int; }
start {
	i = 2;
	while(i < 4){
		i = i+1;
	};
} end
```

Este programa genera como resultado los siguientes cuadruplos:

```
Unset
0: GOTO nil nil 1
1: = 2 nil i
2: < i 4 t0
3: GOTOF t0 nil 7
4: + i 1 t1
5: = t1 nil i
6: GOTO nil nil 2
7: HALT nil nil nil
```

- ***Generación de Cuádruplos para Funciones***

La generación de cuádruplos para funciones permite estructurar las llamadas a funciones y gestionar sus valores de retorno, asegurando un flujo de ejecución organizado.

Lógica de Generación:

1. **Declaración de Función:** Durante la declaración, se almacena cada función en el directorio de funciones junto con información sobre su tipo, número de parámetros y la dirección de inicio en la que comienza su ejecución.
2. **Cuádruplos de Llamada:** En una llamada a función, se generan cuádruplos para cada parámetro, cargándolos en sus respectivas direcciones virtuales dentro del contexto de la función. A continuación, se crea un cuádruplo GOSUB que realiza el salto a la dirección de inicio de la función.
3. **Retorno de Valores:** Si la función devuelve un valor, este se almacena en una variable temporal para ser utilizado por el código que efectuó la llamada. Al finalizar

la función, se genera un cuádruplo ENDPROC, indicando el fin de la función y liberando las variables locales.

- ***Ejemplo de cuadruplos para funciones despues de ejecución:***

Aquí muestro un ejemplo de cuadruplos después de ejecutar un programa simple como el siguiente:

```C/C++
program easy;
void print_sum(num1 : int, num2 : int){
    print("la suma es: ", num1+num2);
};
start {
    print_sum(10, 5);
} end
```

Este programa genera como resultado los siguientes cuadruplos:

```Unset
0: GOTO nil nil 6
1: PRINT nil nil "la suma es: "
2: + num1 num2 t0
3: PRINT nil nil t0
4: ENDPRINT nil nil nil
5: ENDFUNC nil nil nil
6: ERA print_sum nil nil
7: PARAM 10 nil 3000
8: PARAM 5 nil 3001
9: GOSUB print_sum nil 1
10: HALT nil nil nil
```

- ***Traducción de variables, constantes y temporales a direcciones virtuales:***

En el diseño de este compilador, las direcciones virtuales se utilizan para identificar de manera única las variables, constantes y temporales en memoria durante la ejecución del programa. Este esquema asegura una separación lógica de los diferentes tipos de datos (enteros, flotantes, booleanos, etc.) y los diferentes ámbitos (global, local, y temporales). A continuación, se detalla cómo se realizó la asignación de estas direcciones:

1. **Esquema de Direcciones Virtuales**

Las direcciones virtuales están divididas en bloques definidos para cada ámbito y tipo de dato:

- **Globales:** Variables declaradas en el nivel del programa principal.
- **Locales:** Variables declaradas dentro de funciones.

- **Temporales:** Variables generadas durante la evaluación de expresiones o como resultado de operaciones intermedias.
- **Constantes:** Valores literales (como números o cadenas) que son usados en el programa.

Cada bloque de direcciones está segmentado por tipo de dato, en el caso de mi compilador, se asigna de la siguiente manera:

- Las variables globales enteras empiezan en 1000.
- Las variables globales flotantes empiezan en 2000.
- Las variables locales enteras empiezan en 3000.
- Las variables locales flotantes empiezan en 4000.
- Las temporales enteras empiezan en 5000.
- Las temporales flotantes empiezan en 6000.
- Las constantes enteras empiezan en 7000.
- Las constantes flotantes empiezan en 8000.
- Las constantes strings empiezan en 9000.

2. **Asignación de Direcciones Virtuales**
    a. **Variables Globales**
        i. Al momento de declarar una variable global, se asigna una dirección virtual de acuerdo con su tipo de dato.
        ii. Se mantiene un contador separado para cada tipo de dato global (e.g., global_int_counter, global_float_counter).
        iii. Ejemplo: La primera variable global entera se asigna a la dirección 1000, la siguiente a 1001, y así sucesivamente.
    b. **Variables Locales**
        i. Durante el análisis semántico de cada función, se asignan direcciones virtuales a las variables locales siguiendo el mismo principio que las globales.
        ii. Cada función tiene sus propios contadores (local_int_counter, local_float_counter) que se reinician al entrar en el contexto de una nueva función.
        iii. Ejemplo: En una función, la primera variable local flotante se asigna a la dirección 6000.
    c. **Temporales**
        i. Las direcciones para variables temporales se generan durante la evaluación de expresiones y la creación de cuadruplos.
        ii. Cada vez que se requiere un resultado intermedio, se asigna una dirección temporal utilizando un contador separado para cada tipo de dato (temp_int_counter, temp_float_counter).
        iii. Estas direcciones son esenciales para almacenar resultados parciales que se utilizan en operaciones más complejas.
        iv. Ejemplo: Un resultado intermedio entero podría asignarse a la dirección 9000.

d. Constantes

i. Las constantes encontradas en el código fuente se almacenan en una tabla de constantes durante el análisis léxico/sintáctico.

ii. Si una constante no ha sido registrada previamente, se le asigna una dirección virtual única basada en su tipo de dato (constant_int_counter, constant_float_counter).

iii. Ejemplo: El literal entero 42 podría asignarse a la dirección 12000 la primera vez que aparece en el código.

3. Estructuras para Manejar las Direcciones Virtuales

Para realizar esta traducción, se implementaron las siguientes estructuras de datos:

- **Tablas de Variables:**
  Cada ámbito (global, local, y temporal) tiene una tabla donde se mapean los nombres de las variables a sus direcciones virtuales.
  Estas tablas permiten validar el uso correcto de las variables y acceder rápidamente a sus direcciones virtuales.

- **Tabla de Constantes:**
  Mapea los valores literales del código fuente a sus direcciones virtuales.
  Ejemplo: {"42": 12000, "3.14": 13000}.

- **Contadores de Direcciones:**
  Contadores separados para cada tipo de dato y ámbito aseguran que las direcciones sean únicas dentro de su segmento.

4. Ejemplo de Traducción

Consideremos el siguiente código fuente:

```
C/C++
int a;           // Variable global
float b;         // Variable global
void foo() {
    int x;       // Variable local
    float y;     // Variable local
    x = a + 42;  // Uso de un temporal y una constante
}
```

El proceso de traducción sería:

1. Variables globales:
   a. a (int) → Dirección virtual: 1000
   b. b (float) → Dirección virtual: 2000
2. Variables locales en foo:
   a. x (int) → Dirección virtual: 5000
   b. y (float) → Dirección virtual: 6000
3. Constantes:

a. 42 (int) → Dirección virtual: 12000 (registrado en la tabla de constantes)

4. Temporales:
    a. Resultado de a + 42 (int) → Dirección virtual: 9000

El cuádruplo generado para la asignación sería:

```
C/C++
+  1000  12000  9000    # Suma de `a` y `42`, resultado en temporal `9000`
=  9000  -1     5000    # Asignación del resultado a `x`
```

- ***Implementación de direcciones virtuales:***

Para implementar las direcciones virtuales de manera correcta, creé una clase de MemoryManger para poder gestionar de una manera más fácil y centralizada las memorias y direcciones de las variables, funciones, temporales, constantes, etc.

```
C/C++
class MemoryManager {
private:
    // Global memory
    int globalIntBase = 1000, globalIntPointer = 1000;
    int globalFloatBase = 2000, globalFloatPointer = 2000;
    // Local memory
    int localIntBase = 3000, localIntPointer = 3000;
    int localFloatBase = 4000, localFloatPointer = 4000;
  ...
    // Maps to track constants for deduplication
    std::unordered_map<int, int> intConstants;
    std::unordered_map<float, int> floatConstants;
    std::unordered_map<std::string, int> stringConstants;
public:
    int allocateGlobalInt();
...
```

Gracias a esta clase puedo asignar direcciones de una manera más sencilla a las variables y usar estas para generar cuadruplos con las direcciones virtuales. Con esta clase, al generar cuádruplos de operaciones, puedo acceder directamente a las memorias para obtener o asignar valores. Ejemplo de código:

```
C/C++
program easy;
vars{ i : int; }
start {
       i = 20+10*9;
       print("la multiplicacion da como resultado: ", i);
} end
```

Este programa genera como resultado los siguientes cuadruplos con direcciones virtuales:

```
Unset
0: GOTO -1 -1 1
1: * 7001 7002 1001
2: + 7000 1001 1002
3: = 1002 -1 1000
4: PRINT -1 -1 9000
5: PRINT -1 -1 1000
6: ENDPRINT -1 -1 -1
7: HALT -1 -1 -1
```

- ***Manejo de Direcciones Virtuales en la Máquina Virtual***

El compilador después de ejecutar un código de entrada, va a generar un código intermedio en un archivo .obj el cual tendremos que interpretar mediante una máquina virtual.
La máquina virtual utiliza direcciones virtuales para gestionar variables, constantes y temporales en memoria. Estas direcciones son traducidas en tiempo de ejecución para acceder a diferentes áreas de memoria.

Estructura de Memoria
La memoria está segmentada en:

- Global: Para variables globales el cual se interpreta como un stack de registros por default.
- Local: Un stack de registros de activación para variables locales y temporales.
- Constantes: Tabla precargada con valores constantes.

Carga Inicial

- Constantes: Las direcciones y valores se cargan desde el archivo .obj en constants_table.
- La información de las funciones se inserta en una estructura de datos para poder manipularla después.
- Se cargan todos los cuádruplos a una lista para recorrerla al momento de ejecutar el código.

Traducción de Direcciones Virtuales

- Constantes: Se buscan en constants_table.
- Globales: Se acceden en el stack de registro de activación por default para la función principal del programa.
- Locales y Temporales: Se administran dentro del registro de activación actual.

Ejemplo del método get_value:

```python
    def get_value(self, address: Union[int, str]) -> Any:
        if address == '-1':
            return None
        address = int(address)
        # Check constants first
        if address in self.constants:
            return self.constants[address]
        # Get value from memory stack
        value = self.memory_stack.get_value(address)
        return value if value is not None else address
```

Ejecución de Cuádruplos

Durante la ejecución:

- Los valores son leídos y escritos en memoria según sus direcciones virtuales.
- Ejemplo:
  + 7000 7001 9000
    - 7000 y 7001 se obtienen de constants_table (en este caso 10 y 20 respectivamente).
    - Se calcula 10 + 20.
    - El resultado (30) se almacena en temps[9000].

- ***Tests cases del funcionamiento del proyecto:***

Test case 1:

- Código de entrada:

```c
program easy;
vars{
    i : int;
    a : int;
}

start
{
    i = 10*9;
    a = i+12*2;
    print("la multiplicacion da como resultado: ", a);
    print("se termino el programa");
}
end
```

- Código intermedio generado en el archivo .obj:

```
C/C++
# Constant Table
CONSTANTS
7000 10
7001 9
7002 12
7003 2
9000 "la multiplicacion da como resultado: "
9001 "se termino el programa"

# Function Directory
FUNCTIONS
easy void 1 1000 2000 5 0

# Quadruples
QUADRUPLES
0: GOTO -1 -1 1
1: * 7000 7001 1002
2: = 1002 -1 1000
3: * 7002 7003 1003
4: + 1000 1003 1004
5: = 1004 -1 1001
6: PRINT -1 -1 9000
7: PRINT -1 -1 1001
8: ENDPRINT -1 -1 -1
9: PRINT -1 -1 9001
10: ENDPRINT -1 -1 -1
11: HALT -1 -1 -1
```

- Output después de ejecutar el código intermedio en la máquina virtual:

```
C/C++
la multiplicacion da como resultado: 114
se termino el programa
```

Test case 2:

- Código de entrada:

```
C/C++
program sum;

void twoSum(target : int){
    vars{
        i:int;
        sum: int;
```

```
    }
    i = 0;
    while(i < 100){
        i = i + 1;
        sum = i + 1;
        if (sum == target){
            print("target matched when i is equal to: ", i);
        };
    };
};

start
{
    twoSum(10);
}
end
```

- Código intermedio generado en el archivo .obj:

```
C/C++
# Constant Table
CONSTANTS
7000 0
7001 100
7003 10
7002 1
9000 "target matched when i is equal to: "

# Function Directory
FUNCTIONS
twoSum void 1 3000 4000 6 1
sum void 15 1000 2000 0 0

# Quadruples
QUADRUPLES
0: GOTO -1 -1 15
1: = 7000 -1 3001
2: < 3001 7001 3003
3: GOTOF 3003 -1 14
4: + 3001 7002 3004
5: = 3004 -1 3001
6: + 3001 7002 3005
7: = 3005 -1 3002
8: == 3002 3000 3006
9: GOTOF 3006 -1 13
10: PRINT -1 -1 9000
11: PRINT -1 -1 3001
```

```
12: ENDPRINT -1 -1 -1
13: GOTO -1 -1 2
14: ENDFUNC -1 -1 -1
15: ERA twoSum -1 -1
16: PARAM 7003 -1 3000
17: GOSUB twoSum -1 1
18: HALT -1 -1 -1
```

- Output después de ejecutar el código intermedio en la máquina virtual:

```
C/C++
target matched when i is equal to: 9
```

Test case 3:

- Código de entrada:

```
C/C++
program simple;
vars{
    a, b: int;
    f: float;
}
void uno(a:int) {
  a = a + b * a;
  print(a, b, a + b);
};
void dos(a:int, b:int, g:float) {
  vars{
    i: int;
  }
  i = b;
  while (i > 0) {
    a = a + b * i + b;
    uno(i * 2);
    print(a);
    i = i - 1;
    print(i);
  };
};
start
{
  a = 3;
  b = a + 1;
  print(a,b);
  f = 3.14;
  dos(a+b*2, b, f*3);
```

```
  print(a,b,f*2+1);
}
end
```

- Código intermedio generado en el archivo .obj:

```
C/C++
# Constant Table
CONSTANTS
7000 0
7001 2
7002 1
7003 3
8000 3.14

# Function Directory
FUNCTIONS
dos void 10 3000 4000 7 3
uno void 1 3000 4000 3 1
simple void 29 1000 2000 9 0

# Quadruples
QUADRUPLES
0: GOTO -1 -1 29
1: * 1001 3000 3001
2: + 3000 3001 3002
3: = 3002 -1 3000
4: PRINT -1 -1 3000
5: PRINT -1 -1 1001
6: + 3000 1001 3003
7: PRINT -1 -1 3003
8: ENDPRINT -1 -1 -1
9: ENDFUNC -1 -1 -1
10: = 3001 -1 3002
11: > 3002 7000 3003
12: GOTOF 3003 -1 28
13: * 3001 3002 3004
14: + 3000 3004 3005
15: + 3005 3001 3006
16: = 3006 -1 3000
17: ERA uno -1 -1
18: * 3002 7001 3007
19: PARAM 3007 -1 3000
20: GOSUB uno -1 1
21: PRINT -1 -1 3000
22: ENDPRINT -1 -1 -1
23: - 3002 7002 3008
```

```
24: = 3008 -1 3002
25: PRINT -1 -1 3002
26: ENDPRINT -1 -1 -1
27: GOTO -1 -1 11
28: ENDFUNC -1 -1 -1
29: = 7003 -1 1000
30: + 1000 7002 1002
31: = 1002 -1 1001
32: PRINT -1 -1 1000
33: PRINT -1 -1 1001
34: ENDPRINT -1 -1 -1
35: = 8000 -1 2000
36: ERA dos -1 -1
37: * 1001 7001 1003
38: + 1000 1003 1004
39: PARAM 1004 -1 3000
40: PARAM 1001 -1 3001
41: * 2000 7003 2001
42: PARAM 2001 -1 4000
43: GOSUB dos -1 10
44: PRINT -1 -1 1000
45: PRINT -1 -1 1001
46: * 2000 7001 2002
47: + 2002 7002 2003
48: PRINT -1 -1 2003
49: ENDPRINT -1 -1 -1
50: HALT -1 -1 -1
```

- Output después de ejecutar el código intermedio en la máquina virtual:

```
C/C++
34
40444
31
3
30434
47
2
20424
59
1
10414
67
0
347.28
```

- ***PanicoCompiler Github:*** ***https://github.com/NicoHwpo/PanicoCompiler***