Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas

Trabajos Prácticos 3 y 4

Diseño de Compiladores I

Ayudante a cargo: José A Fernández León

GRUPO 8

Martín Vazquez Arispe martin.vazquez.arispe@gmail.com

Joaquín Benecier joaquinbenecier@hotmail.com

Burckhardt David burck432@gmail.com

24/11/2023

Índice

Introducción

Este informe se centra en la segunda fase del desarrollo integral del compilador construido en el lenguaje C++. Después de haber completado con éxito la fase léxica y sintáctica, la atención se concentra, ahora, en el desarrollo de la generación de código intermedio, los chequeos semánticos y la generación de código Assembler.

Durante esta etapa, se transita de la lista de reglas a tercetos, marcando una transición clave para la representación del código fuente. La gestión eficaz del código intermedio se convirtió en un aspecto central, donde se implementó un sistema de almacenamiento robusto para facilitar la manipulación eficiente de los tercetos generados. Para garantizar una ejecución coherente de las sentencias de control, se diseñó e implementó un algoritmo de bifurcaciones específico, optimizando así la representación y ejecución de las estructuras condicionales y de bucle en el código intermedio.

Por otro lado, se abordaron los chequeos semánticos fundamentales como la compatibilidad de tipos en asignaciones y operaciones, las respectivas conversiones implícitas, alcance y la correcta declaración de elementos como variables, funciones, clases, entre otros chequeos. Para asegurar la coherencia y la integridad del código generado, implementamos una tabla de símbolos eficiente, de la que se habló en el trabajo anterior, que sirvió como fundamento para la verificación semántica. Se buscó, mediante algoritmos, validar de manera precisa las relaciones y operaciones en el código, contribuyendo así a la fiabilidad del compilador. Además, se integró un sistema de detección y manejo de conflictos de tipos, con mensajes claros y descriptivos, facilitando la identificación y corrección de posibles errores en el código fuente.

La fase culminante del proceso involucró la traducción de tercetos al código assembler de Pentium mediante la utilización de la herramienta especializada MASM32 Editor. Dicha herramienta desempeñó una función esencial al posibilitar el ensamblaje eficiente y preciso, contribuyendo de manera significativa al desarrollo del código final. En esta etapa, se implementaron procedimientos rigurosos para la identificación y corrección de errores, garantizando la coherencia entre el código fuente y el ejecutable final. Adicionalmente, se introdujeron estructuras auxiliares con el propósito de identificar el tipo de dato y la operación de cada terceto, logrando así generar código assembler de manera más precisa y eficiente.

Generacion de Codigo Intermedio

Almacenamiento y manejo de código intermedio

Como se mencionó en la introducción, la estructura intermedia asignada al grupo fueron los tercetos. Un terceto se compone de tres partes fundamentales: el operador, el primer operando y el segundo operando.

Para la implementación en C++, se utilizó una *struct* de datos llamado "*terceto*" para representar cada una de estas unidades. Cada terceto contiene campos para el operador, los operandos y tiene además algunos campos agregados para la gestión de los mismos y algunos chequeos semánticos posteriores. Se agregó el tipo resultante, y una variable auxiliar (*varAux*) que almacena el resultado de la operación. Esta variable auxiliar es crucial para la generación posterior del código assembler, ya que se utiliza como referencia para las operaciones. Por otro lado el tipo resultante también es importante para las instrucciones del procesador pero además tiene un rol crucial en lo que refiere al chequeo de compatibilidad de tipos y conversiones implícitas.

Además, para organizar los tercetos en relación con los diferentes ámbitos del programa, se creó un nuevo vector por cada ámbito detectado que contiene los tercetos específicos de ese ámbito. Estos vectores se almacenan en una estructura de datos tipo *unordered_map* llamada "*listaTercetos*". Al mapa se accede por una clave que hace referencia al ámbito actual; cada entrada, entonces, tiene como segundo valor este vector recién mencionado. Esta organización jerárquica nos permite gestionar eficientemente los tercetos y garantizar una representación coherente del código intermedio en concordancia con la estructura del programa en C++.

Todas estas estructuras fueron consolidadas en una clase denominada *"Estructuras Tercetos"*, ubicada dentro de la carpeta *"Análisis Semántico"*. Esta clase está diseñada para centralizar y gestionar de manera eficiente todas las operaciones relacionadas con los tercetos. Contiene métodos especializados que facilitan la administración del código intermedio. La creación de esta clase contribuye a una organización modular y eficaz del código, promoviendo la legibilidad y mantenimiento del compilador en su conjunto.

Manejo del Ámbito

El manejo del ámbito emerge como un aspecto fundamental en el proceso, ya que directamente afecta la verificación del alcance de variables y métodos. Este tema se aborda de manera exhaustiva debido a su influencia directa en la tabla de símbolos y la generación de tercetos. Se aplicó la técnica de Name-Mangling, como se detalló en la teoría, que se basa en renombrar los símbolos presentes en la tabla mediante la concatenación del ámbito al que pertenecen. Esto permite diferenciar dos elementos que comparten el mismo nombre.

En la implementación, se creó una clase en C++ denominada "*Ámbito*", que contiene principalmente un string global "*ámbito*" al que se le van concatenando y eliminando los ámbitos correspondientes separados entre sí por un símbolo de ":". Esta clase desempeña un papel fundamental en la gestión y seguimiento de los ámbitos durante el análisis del código fuente. La concatenación y eliminación dinámica de ámbitos facilita la aplicación eficiente de la técnica antes mencionada, asegurando la correcta diferenciación de símbolos con nombres idénticos pero pertenecientes a contextos distintos. De esta manera, el manejo del ámbito en la implementación se convierte en un componente esencial para garantizar la coherencia y precisión en la construcción de la tabla de símbolos y la generación de tercetos.

Los métodos más significativos de esta clase son los siguientes:

- **add(string):** Este método concatena un nuevo ámbito a la variable, lo que permite ampliar el alcance de la referencia actual.
- **del():** El método elimina el último ámbito, reduciendo así el alcance de la referencia.
- **get():** Este método devuelve el ámbito actual, proporcionando acceso a la información del contexto actual en la ejecución del programa.

Uso de notación posicional

En la implementación de las acciones asociadas a las reglas gramaticales en YACC, se adoptó la notación posicional para facilitar la identificación de los elementos pertinentes de cada regla. Se emplea el símbolo "$$" para hacer referencia al no terminal del lado izquierdo de la regla, mientras que "$1", "$2", "$3", etc. Se utilizan para referenciar los elementos del lado derecho de la regla, ya sean terminales o no. Cada valor numérico representa la posición del elemento en la regla.

Esta notación posicional resultó fundamental para la aplicación de diversos chequeos semánticos a medida que se detectaban las reglas gramaticales. Facilitó la manipulación y verificación de los elementos involucrados en las reglas, permitiendo una implementación eficaz de las validaciones necesarias en el análisis sintáctico. Además de su utilidad en la realización de chequeos semánticos, esta notación posicional fue instrumental en la generación del código intermedio, ofreciendo una forma clara y estructurada de acceder y manipular los elementos de las reglas gramaticales durante el proceso de compilación. La adopción de esta notación contribuyó a la coherencia y modularidad del compilador al

permitir un manejo preciso de los datos en las distintas etapas del análisis sintáctico y la generación de código intermedio.

A continuación se presentan algunos de los muchos usos en los que se presenta la notación posicional:

```
factor: nesting  {$$ = stepsFactor($1);}
      | constant {$$ = $1;}
      | lessless {$$ = $1;}
      ;
```

Imagen 1: Reglas gramaticales de "factor" junto con las acciones semánticas correspondientes.

En este contexto, observamos la estructura de asignación de factores en la gramática. El no terminal "*factor*" se referencia de manera inequívoca mediante la notación $$. Por su parte, $1 se utiliza para hacer referencia a los resultados de las producciones que ocupan la primera posición en el lado derecho de la regla gramatical. En este caso, $1 apunta a las producciones "*constant*" y "*lessless*", que encapsulan valores asociados a constantes numéricas y a variables u objetos sobre los cuales se aplica el operador especial "--".

En particular, en la producción llamada "*nesting*" del lado derecho, se lleva a cabo un preprocesamiento del valor mediante el método "stepsFactor". Este proceso de preprocesamiento es esencial para ajustar y preparar el valor antes de su asignación al lado izquierdo ($$).

```
comparison: expression operatorsLogics expression {$$ = stepsOperation($1, $3, $2);}
          ;

expression: expression'+'termino { $$ = stepsOperation($1, $3, "+"); }
          | expression'-'termino { $$ = stepsOperation($1, $3, "-"); }
          | termino {$$ = $1;}
          ;
```

Imagen 2: Reglas gramaticales de comparación y expresión junto con las acciones semánticas correspondientes.

En la presente ilustración, se evidencian dos aplicaciones distintas de la notación posicional. En primer lugar, se hace referencia al no terminal *"comparison"*, el cual detalla las comparaciones que constituyen las condiciones de las sentencias de control. En este contexto, se emplea la función *stepsOperation*, encargada de generar los tercetos correspondientes según el operador recibido. Antes de crear el terceto, esta función verifica la compatibilidad y realiza las conversiones necesarias, recibiendo tres parámetros. Los dos primeros parámetros se asocian con los operandos, referenciados como $1 y $3, respectivamente. El tercer parámetro hace alusión al operador utilizado en la operación, que, en el caso de las comparaciones, puede ser >, >=, <, <=, !!, ==, y como se observa, se vincula con $2. Por otro lado, en cuanto al no terminal *"expression"*, sólo puede ser + o -, por lo que la notación

posicional no es necesaria en estos casos. No obstante, al igual que en la regla del no terminal *"comparison"*, se relaciona a $1 y $3 con sus operandos.

En ambas reglas, se aprecia que el retorno de *stepsOperation* se asigna a $$, de esta manera, el no terminal del lado izquierdo se carga con el número de terceto en el que se está llevando a cabo la operación actual, ya sea una comparación, suma o resta. Este número podría necesitar ser referenciado para una asignación posterior. De manera similar al caso del "*factor*" mencionado anteriormente, el no terminal "*expression*" podría cargarse con el término formado en los tercetos anteriores, por lo que se observa la acción semántica $$ = $1. Esto es útil en los casos en que "*expression*" no involucra sumas ni restas.

```
operatorsLogics: EQUAL {$$ = "==";}
               | NOTEQUAL {$$ = "!!";}
               | GREATEREQUAL {$$ = ">=";}
               | LESSEQUAL {$$ = "<=";}
               | '<' {$$ = "<";}
               | '>' {$$ = ">";}
               ;
```

Imagen 3: Reglas gramaticales de los operadores lógicos junto con las acciones semánticas correspondientes.

```
type: SHORT {$$="SHORT";}
    | ULONG {$$="ULONG";}
    | FLOAT {$$="FLOAT";}
    ;
```

Imagen 4: Reglas gramaticales de tipos primitivos junto con las acciones semánticas correspondientes.

Otro ejemplo de la notación posicional se aprecia en la asignación de terminales. En este caso, se realiza la asignación del operador lógico de las condiciones y los tipos primitivos. En la representación visual, se observa cómo el símbolo $$, que representa al no terminal *"operatorsLogics"*, adquiere el valor del operador correspondiente en el lado izquierdo, mientras que en el lado derecho, el no terminal "type" se asigna con alguno de los tipos primitivos pertenecientes al grupo: $$ = "SHORT", $$ = "ULONG", $$ = "FLOAT

Algoritmo de bifurcaciones para sentencias de control

En la implementación de las sentencias de control, como el IF y el WHILE, se introdujeron bifurcaciones estratégicas expresadas en forma de tercetos incompletos. La técnica que sustenta el algoritmo a cargo de estas bifurcaciones fue desarrollada en la teoría y se conoce como **backpatching**. Estos tercetos incompletos, denominados bifurcaciones por falso (BF), postergan su completitud hasta que se conoce la dirección del salto correspondiente. Estas bifurcaciones son esenciales para gestionar las decisiones de control de flujo en el programa.
Se incorporaron también bifurcaciones incondicionales (BI) que no involucran el chequeo de ninguna condición para saltar. Estas bifurcaciones estratégicas optimizan el flujo de ejecución y son esenciales para la coherencia y eficiencia de la generación del código intermedio en situaciones de control de flujo condicional y de bucle.

En las siguientes secciones se aborda el algoritmo utilizado para cada una de las sentencias de control. Antes de eso, se explicará brevemente la implementación general que sostiene el backpatching mencionado. Al estar trabajando en C++ se utilizó la estructura predefinida *stack<int>* a la que la instanciamos con el nombre de *"pilaTercetos"* permitiendo apilar y

desapilar enteros. Estos enteros hacen referencia a los números de los tercetos encargados de la bifurcación.

Sentencia IF

A continuación, se presenta una detallada explicación sobre la utilización de la pila para marcar las bifurcaciones en la sentencia IF-ELSE. Este enfoque consiste en que al concluir los tercetos asociados a la condición, se genera una bifurcación por falso (BF) incompleta. Se apila el número del terceto correspondiente, postergando la finalización de la bifurcación hasta que se conozca la dirección del salto. Continuando con los tercetos que componen el cuerpo de la rama THEN, al finalizar, se desapila el entero y se completa la bifurcación por falso anterior con la dirección del primer terceto relacionado con la rama del ELSE. Además, se genera una bifurcación incondicional (BI) y se apila el número del terceto. Esta bifurcación actúa como un salto hacia afuera del IF, evitando la ejecución del cuerpo del ELSE en caso de que se ejecute el cuerpo del THEN. Finalmente, al concluir con el código intermedio asociado a la rama ELSE, se desapila el único valor de la pila y se completa el terceto correspondiente a la BI con el número del primer terceto que continúa la sentencia IF. Este enfoque dinámico y estratégico asegura una gestión eficaz y precisa de las bifurcaciones en la estructura IF-ELSE, optimizando así el proceso de generación del código intermedio.

Por otra parte, en situaciones donde el IF carece de una rama ELSE, el proceso se simplifica a la primera parte. Se genera la bifurcación por falso (BF) después de la condición, dejando el terceto incompleto, en espera de la dirección de salida del cuerpo de la sentencia de control. Al agregar el terceto BF, se apila el entero que lo referencia. Posteriormente, al concluir los tercetos del cuerpo de la única rama, se desapila el entero, completando la dirección de salto. De este modo, el código intermedio puede omitir de manera efectiva el cuerpo del IF en caso de que la condición no se cumpla, simplificando así el proceso y garantizando una gestión eficiente de las estructuras de control sin rama ELSE.

A partir de acá, se desarrollará la implementación en el proyecto el algoritmo general explicado hasta el momento.

```
condition: '('comparison')' {EstructuraTercetos::apilar();EstructuraTercetos::addTerceto("BF",$2,"");}
```

Imagen 5: Regla gramatical referida a la condición junto a las acciones semánticas.

Aquí se puede ver como al finalizar la condición, se llama el método apilar, que agrega el número de terceto actual a la pila. También se agrega el determinado terceto incompleto con la BF y la referencia a la comparación correspondiente. A partir de aquí comienza la construcción de código intermedio para el cuerpo del THEN. Al llegar a un posible ELSE se ejecuta lo siguiente:

```
else: ELSE {jumpEndThen();}
```

Imagen 6: Regla gramatical referida al divisor ELSE junto a las acciones semánticas.

Se observa la invocación a la función *jumpEndThen* que se encarga de la lógica y el manejo de la pila, se adjunta el código correspondiente:

```
void jumpEndThen(){
    int tercetoFalse = EstructuraTercetos::desapilar();
    EstructuraTercetos::apilar();
    EstructuraTercetos::addTerceto("BI","","");
    EstructuraTercetos::updateTerceto(tercetoFalse,EstructuraTercetos::nroSigTerceto());
    EstructuraTercetos::addLabel();
}
```

Imagen 7: Función encargada de gestionar las bifurcaciones al finalizar el cuerpo THEN del IF

Esta función por un lado desapila el número del terceto y lo almacena en una variable. Luego se apila el número del terceto actual para completar a futuro la nueva bifurcación. Esta nueva bifurcación se agrega como un terceto con la etiqueta BI esperando ser completado. Luego se hace uso de un método *updateTerceto* que coloca la dirección de salto correspondiente en caso de que deba saltarse el cuerpo del THEN directamente al ELSE. Por último se agrega una nueva etiqueta "label" para delimitar el comienzo del cuerpo del ELSE, etiqueta que luego utilizará el código assembler para identificar las porciones de código.

Al finalizar el procedimiento entonces continúa el análisis del bloque del ELSE hasta terminar la sentencia de control. En este punto se hace una invocación al método *jumpEndIf*.

```
ifStatement: IF condition iterativeBody else iterativeBody ENDIF {yymenssage("IF");jumpEndIf();}
           | IF condition iterativeBody ENDIF {yymenssage("IF");jumpEndIf();}
```

Imagen 8: Reglas gramaticales referidas a la sentencia de control IF junto a las acciones semánticas.

```
void jumpEndIf(){
    EstructuraTercetos::updateTerceto(EstructuraTercetos::desapilar(),EstructuraTercetos::nroSigTerceto());
    EstructuraTercetos::addLabel();
}
```

Imagen 9: Función encargada de gestionar las bifurcaciones al finalizar el IF

Esta función permite completar la BI del terceto creado al finalizar el THEN (en caso de contar con rama THEN y ELSE) o el terceto BF en caso de que no se tenga rama ELSE. Para ello se debe desapilar el valor actual de la pila y se aplica nuevamente el llamado a *updateTerceto* para modificar el terceto que referencia al recién desapilado y completar su dirección de salto con el identificador del próximo, es decir, el que prosigue al IF. Por último se crea la etiqueta correspondiente para delimitar el final del IF y el comienzo del siguiente bloque de código.

Sentencia WHILE

En el contexto de la sentencia WHILE, el uso de la pila adopta una dinámica particular. Inicialmente, se apila el entero que referencia el primer terceto de la condición. Tras la generación de todo el código intermedio asociado a la condición, se crea un terceto

incompleto de bifurcación por falso (BF). En este punto, se apila el número del entero correspondiente, dejando la bifurcación a la espera de la dirección de salida del cuerpo del bucle. Al llegar al final del cuerpo del WHILE, se procede a desapilar el entero del tope, el cual referencia a la bifurcación por falso. Se completa dicha bifurcación con la dirección del primer terceto luego del WHILE. Finalmente, se desapila la dirección del terceto inicial y se genera una bifurcación incondicional (BI), completando con la dirección recién desapilada. Este enfoque dinámico y secuencial de la pila optimiza la gestión de bifurcaciones en la estructura WHILE, asegurando un control eficiente del flujo de ejecución en el código intermedio.

Este algoritmo funciona correctamente para varias sentencias de control anidadas independientemente del tipo. Como se realizó para la sentencia anterior, se desarrollará la implementación en el proyecto el algoritmo general explicado hasta el momento.

```
while: WHILE {EstructuraTercetos::apilar();EstructuraTercetos::addLabel();}
```

Imagen 10: Regla gramatical referida al encabezado del bucle WHILE junto a las acciones semánticas.

A diferencia de la sentencia IF, primero se apila la dirección del terceto actual y se crea una etiqueta que delimita el cuerpo de la condición. Esto se debe a las posibles iteraciones, dado que las etiquetas funcionan como direcciones de salto en el código assembler. Luego de generar todo el código intermedio de la condición se repiten los mismo pasos que para el caso del IF. Ver imagen 5.

Allí se apila el número del terceto actual que se agrega con la bifurcación por falso (BF) y se deja incompleto, a la espera de la dirección. Una vez hecho esto, se generan los tercetos para el cuerpo del bucle en sí, para que al final se completen las direcciones.

```
whileStatement: while condition DO iterativeBody {yymenssage("While");jumpEndWhile();}
```

Imagen 11: Regla gramatical referida al bucle WHILE junto a las acciones semánticas.

```
void jumpEndWhile(){
    int tercetoFalse = EstructuraTercetos::desapilar();
    EstructuraTercetos::addTerceto("BI","["+to_string(EstructuraTercetos::desapilar())+"]","");
    EstructuraTercetos::updateTerceto(tercetoFalse,EstructuraTercetos::nroSigTerceto());
    EstructuraTercetos::addLabel();
}
```

Imagen 12: Función encargada de gestionar las bifurcaciones al finalizar el WHILE

El método *jumpEndWhile* es crucial para la lógica de las bifurcaciones. Comienza desapilando el número del terceto referido a la BF del final de la condición y almacenandolo en una variable. Después se añade el terceto de la bifurcación incondicional (BI) que permitirá saltar al comienzo del bucle, para obtener esta dirección se desapila nuevamente la estructura, obteniendo así el índice del terceto que almacena la etiqueta “label”. Inmediatamente se hace uso una vez más del método *updateTerceto* para actualizar el terceto

incompleto de la BF añadido luego de la condición. La referencia del terceto a modificar se obtiene del valor desapilado al principio y almacenado en la variable auxiliar *tercetoFalse*. Por otro lado, la dirección será la del siguiente terceto que continúa al bucle. Este último terceto se completa con la etiqueta "label" correspondiente para delimitar el final del WHILE y el comienzo del siguiente bloque de código.

Desarrollo de temas especiales asignados

En esta sección, se analiza la implementación de conceptos clave del diseño del compilador, incluyendo la "Herencia por Composición" con el uso de nombres, la "Declaración de Métodos Concentrados" y la estrategia de "Forward Declaration". Se proporcionará una descripción detallada de cómo estos enfoques fueron integrados en el código del compilador, destacando su funcionamiento específico y su relevancia para la estructura y el rendimiento del sistema.

18. Herencia por Composición - Uso con nombre

Incorporar, como sentencia declarativa, la declaración de clases con la estructura mostrada en los siguientes ejemplos:

```
CLASS ca {
   INT a;c,    // declaración de atributos
   VOID m() {  // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,    // declaración de atributo
   FLOAT a,    // declaración de atributo
   VOID n() {  // declaración de método
      …
   },
   ca,         // nombre de clase
}
…
```

Incorporar, como sentencia declarativa, la declaración de objetos de una clase determinada:

```
ca a1; a2,
cb b1; b2; b3,
```

Incorporar, dentro de las sentencias ejecutables, la posibilidad de utilizar referencias a métodos y atributos de objetos indicando explícitamente cuando el atributo o método corresponden a la clase heredada por composición. Algunos ejemplos son los que se muestran:

```
a1.a = 3_i,
b1.ca.a = 3_i,
b1.a = 2.3,
b1.b = 1.2,
b1.ca.c = 1_i,
b1.ca.m(),
b1.n(),
```

En primer lugar, ante la ausencia de especificaciones en el enunciado, se determinó que la declaración de herencia debe ubicarse exclusivamente al final de la clase que se está definiendo; de lo contrario, se considerará un error. Con el fin de gestionar la herencia, se

introdujo una columna denominada "*hereda*" de tipo *string* en la tabla de símbolos. Esta columna indica si una clase hereda de otra, y solo se completa para las entradas que representan la declaración de una clase. Este enfoque facilita el seguimiento de la jerarquía de clases y proporciona un manejo efectivo de los objetos.

Una vez implementadas estas modificaciones, se abordó el control del correcto llamado de atributos y métodos heredados por parte de los objetos instanciados. Se diseñó un método booleano llamado "*ChequearDeclObjeto*" para validar el uso correcto del objeto, verificando primero su existencia y presencia en la tabla de símbolos. Esto se cumple siempre que el objeto se haya instanciado previamente con la indicación de la clase a la que pertenece. Luego, se verifica que los atributos y métodos utilizados sean válidos y accesibles. Para recorrer el anidamiento del objeto concatenado con ".", se creó el método "*sigID*", que devuelve el siguiente identificador después de encontrar un símbolo ".". Este enfoque permite analizar gradualmente cada parte del uso del objeto, por ejemplo, si se tiene el uso "*obj.a.b.c*", se analizará primero "*obj*", luego "*a*", "*b*" y finalmente "*c*". El tratamiento para "*obj*" es diferente, ya que solo es necesario verificar su declaración, para lo cual se utiliza el mismo método aplicado a las variables, funciones y otros elementos, denominado "*ChequearDeclaracion*".

Para analizar las partes siguientes, se determina si se trata de un método, un atributo o una clase (esto último debido al uso de la herencia con nombre). En caso de no ser ninguno de estos, se genera un error indicando que el uso del objeto es inválido. Si es un método o atributo, implica que no habrá un siguiente identificador para analizar; en caso de haberlo, se emite un error por uso incorrecto. Si el ID en cuestión corresponde a una clase, se verifica que esta clase sea "padre" de la clase a la cual pertenece "*obj*", utilizando la columna "*hereda*". Si todo esto es validado, se procede con el próximo ID; de lo contrario, se emite un error, ya que no se estaría invocando ningún atributo o método.

Para el próximo ID, se repite el procedimiento explicado hasta el momento. La diferencia radica en que si nuevamente se trata de una clase, se verifica que esta sea "abuelo" de la clase de "*obj*", utilizando nuevamente la nueva columna en la tabla de símbolos. El método continúa analizando el próximo ID, que, debido a las restricciones de niveles de herencia, no puede volver a ser una clase, provocando un error en caso contrario, este tendrá que ser obligatoriamente un atributo o un método de la clase “abuelo”.

19. Declaración de métodos concentrada

Los métodos se declaran dentro de la declaración de la clase a la que pertenecen. Además, el compilador debe chequear que no haya más de 3 niveles de herencia. Por ejemplo, si la clase A hereda de la clase B, y B hereda de la clase C, no se debe permitir que la clase C herede de ninguna otra clase. El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

```
CLASS cx{
   …              // declaración de atributos y métodos de la clase cx
}

CLASS cc{
   …              // declaración de atributos y métodos de la clase cc
   },
   cx,            // Se debe informar como error por exceder el límite de niveles de herencia
}

CLASS cb{
   …              // declaración de atributos y métodos de la clase cb
   },
   cc,           // la clase cb hereda de la clase cc
}


CLASS ca {
   …              // declaración de atributos y métodos de la clase ca
   },
   cb,           // la clase ca hereda de la clase cb
}
…
```

Para implementar la declaración concentrada de métodos, se introdujo la clase en C++ denominada "*InsideClass*", encargada de gestionar las clases, métodos del código fuente. Esta clase facilita la delimitación y diferenciación de las declaraciones de métodos y atributos. Entre las funciones clave de esta clase se encuentran *getClass*, *inClass* y *outClass*, que permiten obtener, establecer y salir de la clase actual, respectivamente. Dos funciones de especial relevancia para la declaración concentrada son *insideClass* e *insideMethod*, ambas devolviendo valores booleanos que indican si la compilación se encuentra dentro de una clase y, en su caso, dentro de un método.

Para determinar si un método se está declarando de manera concentrada, basta con verificar, al detectar una nueva función VOID que pretende ser un método, si la compilación se encuentra dentro de una clase (mediante la función *insideClass*) pero fuera de un método (!*insideMethod*). Si *insideMethod* devuelve *true*, entonces la nueva función declarada no sería un método de la clase, sino una función dentro de otro método, lo cual también es válido pero requiere un análisis adicional. En caso de obtener un resultado *false* desde *insideMethod*, la columna "*uso*" de la tabla de símbolos para el VOID declarado se etiquetará como "*método*".

Al completarse la declaración de la clase, se invoca la función *outClass*, asegurando que las próximas declaraciones de funciones VOID no se consideren métodos de una clase, sino simplemente funciones en el contexto del código. De esta manera, en la tabla de símbolos, estos procedimientos tendrán la palabra "*funcion*" en su columna "*uso*" diferenciándolo así del uso “*metodo*” destinado para funciones específicas de las clases.

Para verificar los niveles de herencia, se incorporó una columna denominada "*nivelHerencia*" de tipo entero a la tabla de símbolos, representando el nivel de herencia de la clase actual. Este valor entero debe permanecer menor a 3; de lo contrario, se genera un error indicando que se han excedido los niveles de herencia. El método "*inicNivelHer*" inicializa este entero, estableciendo su valor en 1 al inicio de la declaración de una nueva clase. A medida que se utiliza el método "*setHerencia*" de la clase que gestiona la tabla de símbolos, este entero

aumenta. Dicha función coloca el nombre de la clase padre en la columna "*hereda*" indicando la herencia e incrementa el contador en la columna "*nivelHerencia*" de la entrada de la clase actual. Este aumento se basa en el valor de niveles que tiene la clase padre. Si la clase padre ya tenía una herencia previa con un valor de 2, la clase actual pasará a tener un valor de 3. Así, se indica que la clase tiene una nueva herencia, controlando que no se exceda el límite establecido de niveles.

21. Forward declaration

Incorporar la posibilidad de declarar las clases con posterioridad a su uso.

```
CLASS ca {      // Declaración de clase ca
   VOID m() {
      …
      }
   ...
   }
CLASS cc,
CLASS cd {
   cc c1,
   ca a1,
   VOID j() {
     c1.p(),    // Debe ser informado como error por el compilador
     a1.m(),    // Invocación correcta
   }
}
CLASS cc {      // forward declaration para clase cc
  INT z,
  VOID p() {
     …
     }
  …
}
```

La implementación de este tema especial involucró una exhaustiva evaluación mediante casos de prueba, dada la multiplicidad de situaciones que podían surgir con la funcionalidad en cuestión y, por consiguiente, la probabilidad de enfrentar problemas. En el proceso de implementación, se introdujo una columna adicional en la tabla de símbolos, denominada "*forwDecl*", destinada a gestionar la declaración anticipada de clases y objetos. Esta columna puede tener tres valores distintos:

- -1: Si la clase/objeto no cuenta con una declaración anticipada.
- 0: Si la clase/objeto posee una declaración anticipada pero aún no se ha completado su definición
- 1: Si la clase/objeto cuenta con una declaración anticipada que ha sido completamente definida.

Cuando se detecta el uso de atributos y métodos por parte de un objeto, se realiza una revisión de la clase a la que pertenece. Si la columna "*forwDecl*" de dicha clase tiene el valor 0, los atributos y métodos utilizados se incorporan a la tabla de símbolos, y se verificarán al completar la declaración de la clase. Durante esta etapa, se verifica la validez de los componentes previamente declarados. Para cada uno de los atributos y métodos, se busca en la tabla de símbolos para confirmar si fueron declarados anteriormente. Si este es el caso, la columna "*forwDecl*" de ese elemento se actualiza a 1, indicando una completitud exitosa.

Al finalizar el análisis completo del programa, se ejecuta un método llamado *ChequearForwardDeclarations* que verifica si alguna entrada en la tabla tiene la columna "*forwDecl*" con un valor de 0, indicando que en algún momento se utilizó un atributo o método o se declaró una clase pero su declaración completa nunca se concretó. En tales casos, se emite el correspondiente mensaje de error.

Uso de métodos con parámetros en objetos con forward declaration

Para abordar este caso especial, se adoptó un enfoque meticuloso dado su complejo entramado de verificaciones. En primera instancia, se incorporó una columna denominada "*tiene_parametro*", de tipo entero, que puede adoptar los siguientes valores:

- 0: Para todas las entradas no relacionadas con este caso.
- 1: En caso de hacer uso de un método sin parámetros.
- 2: En caso de hacer uso de un método con parámetros.

Inicialmente, se verifica si la invocación al método emplea parámetros reales; en caso contrario, el procedimiento con el forward declaration sigue su curso normal, según lo explicado en la sección anterior. Sin embargo, si se detecta el uso de parámetros, además de lo mencionado anteriormente, se actualiza la columna "*tiene_parametro*" con el valor 2 para esa entrada específica. Se genera un terceto vacío para contemplar posibles conversiones del parámetro real al formal. En caso de no requerir una conversión, el operador del terceto quedará vacío, y posteriormente el ensamblador no procesará este terceto. De inmediato, se crea el terceto relativo a la asignación del parámetro real al formal, el cual queda incompleto y pendiente. Para almacenar estos tercetos, se utiliza un vector llamado "*tercetosVacios*", compuesto por elementos de tipo *struct* denominados "*tercetosIncompletos*". Este struct consta de tres campos: el nombre del método, el índice del terceto incompleto y el nombre de la pila de tercetos correspondiente según el ámbito. Cada uno de estos elementos se revisa al finalizar la declaración de la clase con forward declaration para completar los tercetos restantes, tanto de conversiones como de asignaciones, debido al paso de parámetros.

Algunas aclaraciones relacionadas con las clases que cuentan con forward declaration:

- Se prohíbe la declaración de atributos de tipo objeto en este tipo de clases.
- No se permite que una clase pre declarada herede de otra clase.
- Al detectar el uso de un atributo de una clase con forward declaration, el tipo se infiere al participar en el lado izquierdo de cualquier asignación. Después de este punto, el tipo no puede modificarse.
- Si se utiliza un atributo de este tipo al cual aún no se le ha asignado ningún valor (sin tipo inferido), y se intenta utilizar por primera vez en una operación o asignación en el lado derecho, se emite un error debido a la incompatibilidad de tipos.

24. Sobreescritura de atributos

Una clase que hereda de otra puede sobreescribir atributos declarados en la clase de la que hereda, pero no puede sobreescribir métodos. El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

Este aspecto, en virtud de la implementación del compilador, no generó muchos problemas. La utilización de la herencia por nombre simplificó la diferenciación entre atributos con el mismo nombre pero pertenecientes a clases distintas. Al incluir en el mismo ámbito del uso del objeto los nombres de las clases de las cuales hereda, resulta sencillo determinar a qué clase pertenece el atributo o método en cuestión. La sobrescritura de atributos en sí misma no planteó dificultades, ya que es algo que se permite por defecto y no conlleva cambios específicos. Sin embargo, la necesidad de prohibir la sobrescritura de métodos representó un desafío. Para abordar esta cuestión, se empleó una pila denominada "*métodos*" dentro de la clase "*InsideClass*", que almacena los nombres de los métodos de una clase durante su compilación. Al almacenarlos y detectar una herencia al finalizar la declaración de la clase, se verifica, para cada método de la pila, si está declarado en la clase padre o abuelo. A medida que se chequean se van desapilando hasta alcanzar una pila vacía. En ausencia de herencia, se desapilan automáticamente todos los métodos sin realizar ninguna verificación dado que no es necesario.

Un escenario particular se presenta al tener forward declaration; en este caso, la verificación se pospone hasta la completitud de la declaración de esta clase. Se aplica una metodología inversa: se apilan los nombres de los métodos y se lleva a cabo un recorrido en la tabla de símbolos en busca de clases que sean "hijas" o "nietas" de la clase con forward. Al encontrar estas clases, se realiza una comparación uno a uno con cada método apilado para asegurar que ninguna clase heredera realice una sobrescritura.

26. Comprobaciones de uso de variables

El compilador debe chequear e informar, para cada variable declarada, si no aparece en el lado izquierdo de al menos una asignación en el ámbito donde se declaró.

Para verificar este aspecto, se implementó la clase en C++ denominada "*VarSinInic*", la cual se basa en una pila de hashmaps. Cada mapa en la pila representa un ámbito distinto, donde la clave es el nombre del ámbito y el segundo elemento es un conjunto (*set*) de identificadores que representan variables que aún no han sido asignadas. Durante la compilación del programa, al iniciar la compilación de funciones o métodos, se añade un nuevo mapa a la pila. A medida que se declaran variables en ese ámbito, se agregan al mapa correspondiente. Cuando una variable es asignada, se elimina del conjunto respectivo. Al finalizar la declaración de la función o método, antes de salir del ámbito, se realiza una revisión del mapa para comprobar si el conjunto aún contiene variables. Si es así, se emite un mensaje de advertencia indicando que estas variables nunca han sido asignadas. Al finalizar este análisis, se elimina el hashmap del ámbito en cuestión.

Este mismo proceso se repite al finalizar el programa para verificar si se cumple la condición para todas las variables declaradas en el ámbito "*main*". Si se cumple, el conjunto del hashmap con clave "*main*" deberá estar vacío, y luego se procede a eliminarlo.

Errores considerados

A continuación, se presenta una exhaustiva lista que abarca todos los chequeos semánticos implementados en el compilador, acompañados de los correspondientes mensajes de error que se generan en caso de que el código fuente no cumpla con las restricciones semánticas establecidas. Estos controles son cruciales para garantizar la coherencia y validez del programa, asegurando que el código se adhiera rigurosamente a las reglas semánticas definidas por el compilador. La detección temprana de estos errores facilita una corrección eficiente y contribuye a la robustez y fiabilidad del proceso de compilación.

En los siguientes errores se pueden encontrar algunas palabras que hacen referencia a variables de la implementación que dependiendo el valor que toman en base al código fuente se muestran de una u otra manera. Esto permite ser más específico para el usuario a la hora de imprimir los errores. Las palabras son:
check: Es la parte del anidamiento analizada en el momento. Ejemplo de un anidamiento: *obj.a.b.c,* aquí *check* podría ser *a, b* o *c.*
antCheck: Es la parte del anidamiento anteriormente analizada. Ejemplo de un anidamiento: *obj.a.b.c,* aquí si *check* es *b*, entonces *a* es *antiCheck,* siempre y cuando se traten de clases. En caso de que check sea un atributo, entonces antCheck almacenará su tipo.
clase: Se trata del nombre de la clase analizada
InsideClass::getClass(): Permite obtener la clase actual donde que se está compilando
declarado, var, clave: Nombre del identificador que produjo el error.
metodoActual: Método el cual se está analizando.
herencia: Clase padre de la clase actual en compilación.
función: Función la cual se está analizando.
tipoDer, tipoPR: Se refiere al tipo del lado derecho de la asignación actual.
tipoIzq, tipoPF : Se refiere al tipo del lado izquierdo de la asignación actual.
tipoOp1: Se refiere al tipo del lado izquierdo de la operación actual.
tipoOp2: Se refiere al tipo del lado derecho de la operación actual.
uso, usoOriginal: Contiene el uso de un identificador. Este uso puede ser: Método, Atributo, Función, Variable, Clase, Objeto o Parámetro formal.

CLASES, MÉTODOS Y ATRIBUTOS

La clase no puede tener un atributo de su mismo tipo ni métodos con su mismo nombre:

- **ERROR:** No es posible declarar un atributo el cual su tipo sea la misma clase a la que pertenece
- **ERROR:** No es posible declarar un método con el mismo nombre al de la clase a la que pertenece.

Se prohíbe el uso recursivo de métodos dentro de las clases:

- **ERROR:** Se está haciendo un llamado recursivo del método

Si al final del anidamiento el objeto se encuentra con el nombre de una clase en lugar de un atributo o método:

- ➢ **ERROR:** Objeto está intentando utilizar una clase ***check*** en lugar de un atributo.

Al final del anidamiento se está intentando utilizar un objeto. Por ejemplo *obj.c* siendo *c* una instancia de un objeto que es un atributo de la clase a la que pertenece *obj*.

- ➢ **ERROR:** Se debe operar con tipos primitivos | SHORT - ULONG - FLOAT.

Se está intentando utilizar un atributo pero en el anidamiento del objeto figura como una invocación a un método, es decir con "()":

- ➢ **ERROR:** Uso no válido de atributo en el llamado a método

Se está intentando invocar un método pero en el anidamiento del objeto figura como un atributo, es decir sin "()" y mucho menos sin parámetros:

- ➢ **ERROR:** Uso no válido de atributo en la invocación a método.

Se está intentando invocar un método pero no se encuentra al final del anidamiento:

- ➢ **ERROR:** Uso no válido de invocación a método.

El nombre de clase o atributo en el anidamiento del objeto no corresponde con ninguna entrada de la tabla de símbolos.

- ➢ **ERROR:** Clase/atributo no existente.

La clase que se intenta declarar ya se encuentra declarada completa previamente (no se trata de forward declaration):

- ➢ **ERROR:** Clase ***clase*** se encuentra re-declarada.

No se permite declarar clases anidadas dentro de otras:

- ➢ **ERROR:** No es posible declarar una clase dentro de otra - Clase ***clase*** dentro de clase ***InsideClass::getClass()***

No es posible tener atributos con el nombre de la clase a la que pertenecen:

- ➢ **ERROR:** El nombre del atributo ***declarado*** es igual al nombre de la clase

Todos los atributos y métodos de una misma clase deben tener nombres distintos entre sí:

- ➢ **ERROR:** La clase no puede tener atributos y métodos con el mismo nombre.

HERENCIA

La herencia máxima permitida es de 3, es decir se podrá tener una clase como máximo, puede heredar de otra que a su vez herede de una última:

- ➢ **ERROR:** La clase ha excedido el nivel de herencia (máximo nivel = 3).

Una clase no puede heredar de ella misma:

- ➢ **ERROR:** La clase hereda de ella misma.

Al utilizar la herencia por nombre en el anidamiento del objeto, se debe verificar que esta herencia sea válida:

- ➢ **ERROR:** Clase ***antCheck*** No hereda de ***check***.

Al utilizar objetos se está haciendo un uso de herencia por nombre incorrecto.

- ➢ **ERROR:** Invocación incorrecta de la clase ***check***.

SOBRESCRITURA DE MÉTODOS

La sobreescritura de métodos está prohibida al heredar de alguna clase:

- ➢ **ERROR:** No es posible en la clase ***clase*** sobreescribir el método ***metodoActual*** de la clase ***herencia*** de la cual hereda.

FORWARD DECLARATION

Al finalizar el programa puede haber métodos, atributos y clases declaradas con Forward Declaration que no se terminaron de concretar o que no figuran finalmente en la clase declarada:

- **ERROR:** Atributo ***clave*** no finalizada su declaración.
- **ERROR:** Metodo ***clave*** no finalizado su declaración.
- **ERROR:** Clase ***clave*** no finalizada su declaración.

Una clase con Forward Declaration tiene prohibido heredar de otra clase o tener atributos que sean instancias de clases (objetos):

- **ERROR:** No es posible que una clase incompleta herede de otra clase o tenga un atributo de tipo objeto previo a su declaración completa.

La declaración final del método de la clase pre declarada debe respetar el uso de parámetros de la primera invocación que se haya hecho del mismo en el código:

- **ERROR:** Metodo ***función*** se ha indicado que NO tiene parámetro en su primer invocación.

Al intentar usar un método todavía no declarado de una clase con pre declaración, se debe respetar el uso de parámetros si corresponde. El requerimiento de parámetros va a estar definido por el primer uso presente en el código de ese método. Es decir, los siguientes llamados del método, deben respetar el uso de parámetros del primer llamado:

- **ERROR:** Método ***función*** primera vez utilizado con parámetro.
- **ERROR:** Método ***función*** primera vez utilizado sin parámetro.

Los atributos de clases pre declaradas no pueden utilizarse en operaciones ni asignaciones si todavía se desconoce su tipo:

- **ERROR:** No es posible asignarle un atributo pre declarado sin tipo a una variable.
- **ERROR:** No es posible realizar operaciones con atributos pre declarados sin tipo.

Se quiere inferir un tipo que no coincide con el tipo del atributo declarado en la clase:

- **ERROR:** Tipo detectado en inferencia no coincide con su declaración de atributo ***var***
- **WARNING:** Se ha inferido el tipo de ***tipoOp1***como **tipoDer**

FUNCIONES

Se prohíbe la recursión de cualquier tipo de llamado y en cualquier ámbito:

- **ERROR:** Se está haciendo un llamado recursivo a la función.

Existen límites de anidamiento para las funciones y para las funciones declaradas dentro de métodos:

- **ERROR:** No es posible anidar otra función, excede los niveles permitidos.

Las clases solo se pueden declarar en el ámbito *main*, no se permiten declarar clases dentro de funciones ni métodos.

- **ERROR:** No es posible declarar una clase dentro de una función - Clase ***clase*** dentro de función ***función***.

Las funciones no pueden tener el mismo nombre que sus parámetros formales:

- **ERROR:** La función ***función*** tiene el mismo nombre que el parámetro formal.

Los métodos y funciones solo deben recibir parámetros reales en caso de contar con los parámetros formales en su declaración

- **ERROR:** El método ***función*** NO requiere parámetro.

- ➢ **ERROR:** El método ***función*** requiere parámetro.
- ➢ **ERROR:** La función ***función*** requiere parámetro.

CONVERSIONES Y COMPATIBILIDAD

Los tipos que se quieren asignar o operar son incompatibles, no es posible realizar una conversión implícita:

- ➢ **ERROR:** No es posible asignarle a un ***tipoIzq*** un ***tipoDer***
- ➢ **ERROR:** No es posible operar entre un ***tipoOp1*** y un ***tipoOp2***
- ➢ **ERROR:** No es posible asignarle a un ***tipoPF*** a un ***tipoPR***.

DECLARACIONES

No es posible hacer uso de una variable que todavía no ha sido declarada:

- ➢ **ERROR:** ***uso var*** NO declarada.

No es posible tener dos identificadores iguales en el mismo ámbito, independientemente de su uso:

- ➢ **ERROR:** ***uso var*** se encuentra re-declarada como ***usoOriginal***.

Todas las variables u objetos deberían tener al menos una asignación de valor, ya sea en el main o en alguna función:

- ➢ **WARNING:** Variable/Objeto ***var*** Sin asignación de un valor en el main.
- ➢ **WARNING:** Variable/Objeto ***var*** Sin asignación de un valor dentro de la función donde se declaró.

Generacion de Codigo Assembler

Tras la generación del código intermedio mediante tercetos, se inicia la fase crucial de la creación de las instrucciones que el procesador ejecutará. Este paso representa la transición del nivel abstracto del código intermedio a las operaciones concretas y específicas del procesador. En esta etapa, cada terceto se traducirá a código assembler, con el objetivo de producir un conjunto de instrucciones eficiente y coherente que refleje fielmente la lógica del programa original.

Mecanismo utilizado para la generación del código Assembler

Pasando a la lógica del mecanismo utilizado para la generación del código Assembler, se destaca el método "*generarCodigo*" dentro del archivo "*Assembler.cpp"*. Este método se encarga de la creación y gestión de los archivos *archivoASM* y *archivoASMCODE*. El primero se carga con todos los includes, encabezados y todas las variables declaradas en la sección .DATA. Es importante destacar que estas variables se cargan al final, después de generar todas las instrucciones, ya que en este punto se conocen todas las variables auxiliares necesarias. Por otro lado, en el archivo *archivoASMCODE*, es donde se almacena todo el código assembler como tal. A medida que se van generando las instrucciones para todos los tercetos, se van almacenando en la tabla de símbolos las nuevas variables auxiliares utilizadas. El contenido de este archivo se mueve, al finalizar, al final del archivo *archivoASM* una vez que se ha cargado por completo la sección .DATA.

En este proceso central, el método generarCodigo hace uso del método "*crearAssembler*", que se explicará más adelante, encargado de la generación de instrucciones. Es relevante destacar que este último método comunica al primero la aparición de errores, que pueden ser: división por cero, overflow en suma de flotantes y overflow en producto de enteros. Los mensajes correspondientes a estos errores se añaden al final del archivo final obtenido, junto con la respectiva instrucción que permite visualizarlos.

Finalmente, en este procedimiento, se recorre la tabla de símbolos por completo para cargar cada una de las declaraciones como un elemento necesario de la sección .DATA. Para manejar los objetos, se utiliza una clase específica llamada "*EstrDeclObj*", que facilita la concatenación y anidamiento de strings para destacar el uso de los objetos con sus atributos, métodos y herencia de clases en caso de existir.

Núcleo de la generación de instrucciones

El método central encargado de la generación de instrucciones es "*crearAssembler*". Este método recibe como parámetro un vector de tercetos, donde cada vector de tercetos representa un ámbito diferente. Durante su ejecución, itera sobre estas estructuras y analiza cada terceto. Cada terceto contiene un operador, dos operandos y el tipo resultante de la operación. Para diferenciar las instrucciones que se deben generar, el compilador concatena los campos del operador y del tipo en un único string, por ejemplo, "+SHORT", "-FLOAT", "<=ULONG", entre muchas otras combinaciones.

Este string obtenido sirve como clave para indexar un hashmap llamado *"codigos"* contenido en la clase "*EstructurasAssembler*". Este hashmap almacena, para cada entrada, el llamado a una función encargada de la construcción de las instrucciones específicas para ese tipo de operación especificado en la clave. Las secciones siguientes proporcionan más detalle sobre la utilidad de estas funciones para las operaciones aritméticas.

Mecanismo utilizado para efectuar operaciones aritméticas

En la siguiente sección se explicara lo anterior mencionado para el caso de las operaciones aritméticas. Se adjunta una imagen del mapa *"codigos"* recortado.

```
unordered_map<string,EstructurasAssembler::FunctionType> EstructurasAssembler::codigos = {
        {"+SHORT", EstructurasAssembler::getSumaShort},
        {"-SHORT", EstructurasAssembler::getRestaShort},
        {"*SHORT", EstructurasAssembler::getMultShort},
        {"/SHORT", EstructurasAssembler::getDivShort},
        {"=SHORT", EstructurasAssembler::getEqualShort},
        {"+ULONG", EstructurasAssembler::getSumaUlong},
        {"-ULONG", EstructurasAssembler::getRestaUlong},
        {"*ULONG", EstructurasAssembler::getMultUlong},
        {"/ULONG", EstructurasAssembler::getDivUlong},
        {"=ULONG", EstructurasAssembler::getEqualUlong},
        {"+FLOAT", EstructurasAssembler::getSumaFloat},
        {"-FLOAT", EstructurasAssembler::getRestaFloat},
        {"*FLOAT", EstructurasAssembler::getMultFloat},
        {"/FLOAT", EstructurasAssembler::getDivFloat},
        {"=FLOAT", EstructurasAssembler::getEqualFloat},
```

Imagen 13: Entradas del hashmap "códigos" referidas a las operaciones aritméticas para todos los tipos primitivos

Aquí se puede observar la clave compuesta por: operador+Tipo y la llamada a la función correspondiente. Debido a la cantidad de operaciones y tipos se procede a señalar algunas a modo de ejemplo, dado que el procedimiento es repetitivo. Cabe destacar que en todas estas funciones se retorna una variable del tipo string "*salida*" en el que se van concatenando todas las instrucciones de la operación separadas por saltos de línea.

Multiplicación SHORT

La función *getMultShort* en la clase "*EstructurasAssembler*" está diseñada para generar instrucciones en lenguaje ensamblador específicamente para la operación de multiplicación entre operandos de tipo SHORT. Aquí también se verifica la ocurrencia de overflow.

```
string EstructurasAssembler::getMultShort(string operando1, string operando2, string & varAux, bool error[]){
    string salida = MOV+AL+", "+operando1;
    salida = salida + "\n" + IMUL+operando2;
    varAux = generarVariable();
    salida = salida + "\n" + MOV+varAux+", "+AL;
    salida = salida + "\n" + JO + "overflow_mulEnt";
    error[2]=true;
    return salida;
}
```

Imagen 14: Función generadora de código Assembler para la operación de multiplicación entre operandos SHORTs

A continuación, se proporciona una explicación detallada:

- Se carga el valor de *operando1* en el registro AL.
  - MOV AL , *operando1*
- Se realiza la multiplicación con el valor de *operando2* utilizando la instrucción IMUL. Esta instrucción multiplica el contenido de AL (el primer operando) con el operando especificado (operando2) y almacena el resultado en el par de registros DX:AX.
  - IMUL *operando2*
- El resultado de la multiplicación se almacena en el registro AX.
- Se genera una variable auxiliar (*varAux*) para almacenar el resultado de la multiplicación.
- Se mueve el contenido de AL (resultado de la multiplicación) a la variable auxiliar
  - MOV *varAux* , AL
- Se verifica si se produce un overflow después de la multiplicación. La instrucción JO realiza un salto si se produce un overflow, llevando la ejecución a la etiqueta "*overflow_mulEnt*".
  - JO *overflow_mulEnt*
- En caso de detectar overflow, se activa el flag de error correspondiente (*error[2]*) y se dirige a la etiqueta *overflow_mulEnt*, donde se maneja el error asociado a un desbordamiento en la multiplicación de SHORTs.

División ULONG

La siguiente función *"getDivUlong"* se encarga de realizar la división entre operadores del tipo ULONG.

```
string EstructurasAssembler::getDivUlong(string operando1, string operando2, string & varAux, bool error[]){
    string salida = MOV+EAX+", "+operando1;
    salida = salida + "\n" + XOR+EDX+", "+EDX;
    if (isdigit(operando2[0])){
        salida = salida + "\n" + MOV+EBX+", "+operando2;
        salida = salida + "\n" + CMP + EBX + ", " + "0";
    }else{
        salida = salida + "\n" + CMP + operando2 + ", " + "0";
    }
    salida = salida + "\n" + JE + "etiqueta_divcero";
    if (isdigit(operando2[0])){
        salida = salida + "\n" + MOV+ECX+", "+operando2;
        salida = salida + "\n" + DIV+ECX;
    } else {
        salida = salida + "\n" + DIV+operando2;
    }
    varAux = generarVariable();
    salida = salida + "\n" + MOV+varAux+", "+EAX;
    error[0]=true;
    return salida;
}
```

Imagen 15: Función generadora de código Assembler para la operación de división entre operandos ULONGs

En detalle, las instrucciones generadas que se ven en la imagen son las siguientes:

- Comienza moviendo el valor de *operando1* al registro EAX.
  - MOV EAX , *operando1*
- Se establece EDX en cero.
  - XOR EDX , EDX
- Luego, realiza una verificación para asegurarse de que el divisor (*operando2*) no sea cero, evitando así una posible división por cero. En caso de que el divisor sea una constante numérica, se carga su valor en el registro EBX y se realiza la comparación con cero.
  - MOV EBX , *operando2*
  - CMP EBX , 0
- Si el divisor es una variable, se realiza directamente la comparación con cero.
  - CMP *operando2* , 0
- Si la comparación resulta en igualdad (JE), se salta a la etiqueta "*etiqueta_divcero*" para manejar la situación de división por cero.
  - JE *etiqueta_divcero*
- En el caso de que el divisor no sea cero, se procede con la división. Si el divisor es una constante numérica, su valor se carga en el registro ECX antes de ejecutar la instrucción DIV.
  - MOV ECX, *operando2*
  - DIV ECX

- Si es una variable, se utiliza directamente el *operando2* en la instrucción DIV. El resultado de la división se almacena en EAX.
  - DIV *operando2*
- Posteriormente, se genera una variable auxiliar única utilizando la función *generarVariable*, y se mueve el resultado de la división (almacenado en EAX) a esta variable. Además, se activa la señal de error correspondiente, indicando la posibilidad de división por cero (*error[0]*).
  - MOV *varAux* , EAX

Suma FLOAT

La función *getSumaFloat* en la clase "*EstructurasAssembler*" está diseñada para generar instrucciones en lenguaje ensamblador específicamente para la operación de suma entre operandos de tipo FLOAT. La lógica de la función se centra en utilizar instrucciones específicas de la FPU (Unidad de Punto Flotante) para realizar la suma y gestionar posibles desbordamientos o errores asociados con operaciones aritméticas de punto flotante.

```
string EstructurasAssembler::getSumaFloat(string operando1, string operando2, string & varAux, bool error[]){
    string salida = FLD+operando1;
    salida = salida + "\n" + FADD+operando2;
    varAux = generarVariable();
    salida = salida + "\n" + FCOM+" CERO"+"\n"+FSTSW+AX+"\n"+SAHF+"\n"+JE+"realizarAsignacion";
    salida = salida + "\n" + FCOM+" MAXPOSITIVO"+"\n"+FSTSW+AX+"\n"+SAHF+"\n"+JA+"overflow_add_float";
    salida = salida + "\n" + FCOM+" MINNEGATIVO"+"\n"+FSTSW+AX+"\n"+SAHF+"\n"+JB+"overflow_add_float";
    salida = salida + "\n" + FCOM+" MINPOSITIVO"+"\n"+FSTSW+AX+"\n"+SAHF+"\n"+JA+"realizarAsignacion";
    salida = salida + "\n" + FCOM+" MAXNEGATIVO"+"\n"+FSTSW+AX+"\n"+SAHF+"\n"+JA+"overflow_add_float";
    salida = salida + "\n" + "realizarAsignacion:" + "\n" + FSTP+varAux;
    error[1]=true;
    return salida;
}
```

Imagen 16: Función generadora de código Assembler para la operación de suma entre operandos FLOATs

En detalle, las instrucciones generadas que se ven en la imagen son las siguientes:

- Carga el primer operando de tipo FLOAT en el registro de la FPU.
  - FLD *operando1*
- Suma el segundo operando de tipo FLOAT al valor actual en el registro de la FPU.
  - FADD *operando2*
- Se genera una nueva variable auxiliar (*varAux*) para almacenar el resultado de la suma.
- Se hace una comparación con cero del resultado de la suma dado que este es un valor válido. En caso de ser igual a cero, se hace un salto para realizar la asignación.
  - FCOM CERO
  - FSTSW *AX*
  - SAHF
  - JE realizarAsignacion
- Se realizan comparaciones con valores extremos (*MAXPOSITIVO*, *MAXNEGATIVO*, *MINPOSITIVO*, *MINNEGATIVO*) para verificar si la operación de suma ha excedido los límites permitidos para números de punto flotante.

- FCOM *MAXPOSITIVO*
- FSTSW *AX*
- SAHF
- JA overflow_add_float
- FCOM *MINNEGATIVO*
- FSTSW *AX*
- SAHF
- JB overflow_add_float
- FCOM *MINPOSITIVO*
- FSTSW *AX*
- SAHF
- JA *realizarAsignacion*
- FCOM *MAXNEGATIVO*
- FSTSW *AX*
- SAHF
- JA overflow_add_float

➢ En caso de detectar un desbordamiento, se activa el flag de error correspondiente (*error[1]*) y se dirige a la etiqueta *overflow_add_float*, donde se maneja el error asociado a un desbordamiento en la suma de flotantes.

➢ En caso de no detectar desbordamiento, la instrucción almacena el resultado final en la variable auxiliar.

- *realizarAsignacion:*
- FSTP *varAux*

Es importante mencionar que las instrucciones específicas de la FPU incluyen FLD para cargar datos en la pila de la FPU, FADD para realizar la suma, y FSTP para almacenar el resultado. Además, las instrucciones de comparación (FCOM) y las instrucciones de salto condicional (JA, JB, etc.) se utilizan para gestionar casos de desbordamiento. Estas instrucciones están diseñadas para trabajar con números de punto flotante y manejar situaciones especiales que puedan surgir durante la ejecución de operaciones aritméticas.

Por otro lado, la instrucción FSTSW (Floating Point Store Status Word) se emplea para adquirir el estado actual de la FPU, que engloba condiciones como cero, negativo, positivo, cero o infinito, y desbordamiento aritmético. Al almacenar este estado en el registro AX, se permite la realización de comparaciones condicionales basadas en dicho estado. En esta función, tras efectuar la suma con FADD, se recurre a FSTSW seguido de SAHF (Store AH into Flags) para transferir el byte de estado del registro AX al registro de flags de la CPU. Luego, se ejecutan comparaciones condicionales (JA, JB, etc.) empleando el estado almacenado en los flags para determinar si la suma ha superado los límites admisibles para números de punto flotante.

Generación de las etiquetas destino de las bifurcaciones

En esta sección, se aborda la tarea de generar etiquetas relacionadas con las bifurcaciones resultantes de las sentencias de control, específicamente en los casos del condicional IF y el bucle WHILE. Estas etiquetas desempeñan un papel crucial en el control del flujo de ejecución del programa. El proceso se lleva a cabo mediante técnicas como backpatching y el uso de una pila, estrategias que ya fueron mencionadas en la fase de generación de código intermedio. En los tercetos, el primer valor, correspondiente al operador, puede ser BF (Bifurcación por Falso) o BI (Bifurcación Incondicional). En otros casos, cuando este primer operador señala el inicio de una porción de código a la cual se debe saltar, se tratara de un terceto que únicamente contiene la etiqueta *"label"* en el campo del operador.

```
if (tercetos[i].operador.find("label") == string::npos){
    string op;
    string ftOp;
    string scOp;
    if (tercetos[i].operador == "BF"){
        op = tercetos[getNro(tercetos[i].operando1)].operador+tercetos[getNro(tercetos[i].operando1)].tipo;
        ftOp = reemplazarCaracter(tercetos[getNro(tercetos[i].operando2)].operador+claveTS,':','_');
    } else if (tercetos[i].operador == "BI"){
        op = tercetos[i].operador;
        ftOp = reemplazarCaracter(tercetos[getNro(tercetos[i].operando1)].operador+claveTS,':','_');
```

Imagen 17: Fragmento de código que indica el manejo de etiquetas BF y BI.

En el caso de que el operador no contenga la palabra *label*, existe la posibilidad de estar ante la etiqueta *BF* o *BI*, a continuación se explica cómo se procede en esta situación.

Bifurcación por Falso (*BF*):

- Se verifica si el operador del terceto actual es una bifurcación por falso (BF).
- En caso afirmativo, se obtienen los operadores (*operando1* y *operando2*) y el tipo de los tercetos referenciados por dichos operandos.
- Se construye la clave (*op*) para acceder al hashmap *codigos* donde se almacenan los llamados a las funciones generadores de Assembler. Esta clave se conforma por el operador que contiene el terceto referenciado en el *operando1* y su tipo.
- Se arma el primer operando que recibirá por parámetro la función (*ftOp*). Este se genera reemplazando los caracteres ':' por '_' en el operador del segundo terceto referenciado concatenado con la clave del ámbito actual (*claveTS*).

Bifurcación Incondicional (*BI*):

- Si el operador del terceto es una bifurcación incondicional (BI), se construye la clave (*op*) del salto incondicional con el mismo operador del terceto actual.
- La etiqueta de destino (*ftOp*) se genera reemplazando los caracteres ':' por '_' en el operador del primer terceto referenciado y concatenándolo con la clave del ámbito actual (claveTS).

Luego de esto, en ambos casos se obtiene la función correspondiente mediante el metodo *getFuncion(op)* y luego se ejecuta con los parámetros *ftOp, scOp, aux* y *error.*

```
archivoASMCODE << EstructurasAssembler::getFuncion(op)(ftOp, scOp, aux,error) << endl;
```

Imagen 18: Invocación a la función que genera el Assembler correspondiente según el operador y el tipo.

En el caso de una BF la función getFuncion(op) puede devolver alguna de las siguientes funciones:

```
{"==SHORT", EstructurasAssembler::getCompEqualShort},
{"!!SHORT", EstructurasAssembler::getComDifShort},
{"<SHORT", EstructurasAssembler::getCompLessShort},
{">SHORT", EstructurasAssembler::getCompGreaterShort},
{"<=SHORT", EstructurasAssembler::getCompLessEqShort},
{">=SHORT", EstructurasAssembler::getCompGreaterEqShort}
{"==ULONG", EstructurasAssembler::getCompEqualUF},
{"!!ULONG", EstructurasAssembler::getComDifUF},
{"<ULONG", EstructurasAssembler::getCompLessUF},
{">ULONG", EstructurasAssembler::getCompGreaterUF},
{"<=ULONG", EstructurasAssembler::getCompLessEqUF},
{">=ULONG", EstructurasAssembler::getCompGreaterEqUF},
{"==FLOAT", EstructurasAssembler::getCompEqualUF},
{"!!FLOAT", EstructurasAssembler::getComDifUF},
{"<FLOAT", EstructurasAssembler::getCompLessUF},
{">FLOAT", EstructurasAssembler::getCompGreaterUF},
{"<=FLOAT", EstructurasAssembler::getCompLessEqUF},
{">=FLOAT", EstructurasAssembler::getCompGreaterEqUF},
```

Imagen 11: Fragmento de hashmap que denota todas las comparaciones posibles que utiliza luego una BF para saltar.

En el contexto de bifurcaciones por falso, se realiza una comparación previa expresada en un terceto referenciado por el primer operando, mientras que en el segundo operando se almacena el número del terceto correspondiente al *label* al que se debería saltar. La dirección de este salto se captura en la variable *ftOp*, la cual se corresponde con el parámetro formal denominado *operando1* dentro de la función. Aquí se presentan algunas de las funciones que implementan este mecanismo, como *getCompLessEqUF*, *getCompGreaterEqUF* y *getCompEqualUF*. Cabe destacar que todas estas funciones comparten el objetivo común de dirigir la ejecución hacia la etiqueta especificada, después de analizar el resultado de la comparación.

```
string EstructurasAssembler::getCompLessEqUF(string operando1, string operando2, string & varAux,bool error[]){
    return JA+operando1;
}
```

Imagen 19: Función que genera la instrucción Assembler de salto Jump Above

```
string EstructurasAssembler::getCompGreaterEqUF(string operando1, string operando2, string & varAux,bool error[]){
    return JB+operando1;
}
```

Imagen 20: Función que genera la instrucción Assembler de salto Jump Below

```
string EstructurasAssembler::getCompEqualUF(string operando1, string operando2, string & varAux, bool error[]){
    return JNE+operando1;
}
```

Imagen 21: Función que genera la instrucción Assembler de salto Jump Not Equal

Por otro lado, para el caso de una *BI* se ejecuta el siguiente código:

```
string EstructurasAssembler::getJump(string operando1, string operando2, string & varAux,bool error[]){
    return JMP+operando1;
}
```

Imagen 22: Función que genera la instrucción Assembler de salto incondicional Jump

En la generación de código intermedio, específicamente en el contexto de una bifurcación incondicional, se utiliza la función *getJump* de la clase "*EstructurasAssembler*". Esta función tiene como objetivo producir la instrucción JMP en código assembler. La instrucción JMP se emplea para dirigir el flujo de ejecución hacia otra sección del programa sin realizar ninguna condición de prueba. En este escenario, el destino del salto se determina mediante el primer operando, denotado como *operando1*, el cual representa la etiqueta (*label*) o dirección a la cual se realizará el salto incondicional.

Por último, como se ve en la imagen, en el caso de que el campo operador del terceto si contenga la palabra *label* entonces, se agrega la etiqueta al archivo .CODE para poder delimitar una nueva porción de código que continua.

```
} else {
    archivoASMCODE << reemplazarCaracter(tercetos[i].operador+claveTS,':','_')+":" << endl;
}
```

Imagen 23: Fragmento de código que se ejecuta en caso de un terceto con una etiqueta "label"

Estas etiquetas permiten controlar el flujo de control y son los destinos que buscan las bifurcaciones tanto BF como BI para proseguir con las instrucciones correctas.

Conclusiones

En el transcurso de este proyecto de diseño de compiladores, se abordaron con éxito tres etapas esenciales, cada una representando un hito crucial en el desarrollo del compilador. Comenzando con el análisis léxico, donde se implementó un analizador basado en autómatas finitos y matrices de transición de estados para lograr la identificación precisa de tokens y gestionar eficientemente la tabla de símbolos.

La fase sintáctica se basó en la herramienta YACC, que facilitó la especificación gramatical y la generación automática del analizador sintáctico. Aspectos clave, como yyparse() y yylval, desempeñaron un papel fundamental en la ejecución del análisis semántico.

La tercera etapa correspondió a la generación de código. Está a su vez se fraccionó en varias fases más pequeñas. La fase más destacada fue el análisis semántico y la generación de código intermedio mediante tercetos. La introducción de tercetos permitió una representación abstracta y eficiente de las operaciones, allanando el camino para la traducción posterior a instrucciones específicas de la arquitectura en la etapa de código Assembler.

Finalmente, en la fase de generación de código Assembler, se implemenaron funciones específicas para operaciones aritméticas y de control de flujo. La gestión de variables auxiliares, la detección de errores y la generación precisa de instrucciones Assembler fueron aspectos cruciales en esta etapa.

Este proyecto de diseño de compiladores no solo ha sido una inmersión profunda en la construcción técnica de un software esencial, sino también una reflexión sobre la importancia a menudo subestimada de los compiladores en el mundo de la programación. Aunque los compiladores trabajan en las sombras, convirtiendo el código fuente en instrucciones ejecutables, son los pilares fundamentales que permiten la ejecución de cualquier programa. Su desarrollo, a menudo arduo y meticuloso, subraya la conexión entre la teoría de lenguajes de programación y la aplicación práctica en la creación de software funcional. Este proyecto nos ha proporcionado una apreciación más profunda de la intersección entre la teoría y la implementación en el diseño de compiladores, recordándonos que, detrás de cada línea de código, hay un proceso complejo que permite la ejecución coherente de programas.