Diseño de Compiladores I

Trabajo Práctico N° 3 y 4

2017

Grupo N°15

Integrantes

Federico Dominguez fedemillo.93@gmail.com
Brian López Muñoz lopezmbrian@gmail.com
Tomás Juárez tomasjuarez.exa@gmail.com

Docente Asignado

Prof. Dr. José Massa

Introducción

En el presente se informará a los docentes las decisiones tomadas a la hora de implementar las soluciones de los temas asignados respecto al trabajo práctico 3 y 4.

El código fue construido en base a la implementación de las entregas anteriores, agregando la lógica necesaria para la generación de código intermedio y posteriormente la generación de código assembler. Para ellos se modificaron reglas de la gramática definidas en el analizador sintáctico y se crearon nuevas clases que encapsulan estos requerimientos.

Generación de Código Intermedio

La generación de código intermedio es una de las tareas más importantes del compilador. Esta consiste en almacenar en una estructura con un tratamiento específico los distintos elementos del lenguaje en base a las reglas del analizador sintáctico. Esto aporta dos ventajas fundamentales en un compilador: portabilidad y posible optimización. La portabilidad se da debido a que si la máquina destino cambia (el código ensamblador) es posible tratar las variantes sobre el código intermedio ya generado en lugar de realizar un nuevo analizador sintáctico. Por otro lado, es posible aplicar optimizaciones sobre el código intermedio antes de la generación de código assembler, aunque esto depende de cuánta sofisticación sea necesaria. En nuestro caso, no se aplicaron optimizaciones ya que no fueron solicitadas por la cátedra.

Diseño e Implementación

A diferencia de las entregas anteriores, la implementación de esta etapa resultó muy sencilla y menos tediosa. Nuevamente se aplicó el paradigma orientado a objetos para facilitar el desarrollo y futuras modificaciones. A continuación se describirán los diferentes módulos involucrados en la etapa de generación de código intermedio.

Modificación en la Tabla de Símbolos

La tabla de símbolos se ha visto modificada respecto a las entregas anteriores para facilitar el desarrollo de la generación de código intermedio. Se agregaron funciones de utilidad sin modificar la lógica de mantenimiento de las estructuras internas. Estas funciones sirven para obtener el tipo de un identificador en cuestión, declarar variables, setear tipos y obtener el último tipo declarado para un identificador (*shadowing*). Estos mecanismos serán explicados más adelante.

Por otro lado, cada símbolo (clase *Symbol*) tienen ahora un *flag* llamado *isAux* para saber si la variable en cuestión es una variable auxiliar o no. Esto sirve únicamente para imprimirlas por pantalla con un mensaje previo que explique esta situación pero no tiene otra función más allá de lo decorativo. Además, se agregó un campo en el símbolo que permite conocer en qué posición de la tabla de símbolos fue insertado. Esto sirve para la

generación de código Assembler del string ya que se le debe asignar un nombre al momento de volcarlo en memoria; esto se explicará en las siguientes secciones.

Representación Intermedia

Nuestra representación de código intermedio es en *tercetos*. Los tercetos tienen, como el nombre lo indica, tres componentes. Esta tupla posee en su primer elemento el operador en cuestión y los dos elementos restantes son operandos. Existen casos en donde el segundo operando no existe y esto es así porque algunas operaciones son unarias.

El diseño de la estructura contenedora de los tercetos es una *fila* (clase *Queue*) o una estructura *FIFO*, debido a que el primer elemento que se inserta es el primero que será tomado una vez generado el código intermedio. Esto permite que los tercetos referenciados puedan resolverse antes de que sean referenciados en futuros tercetos. En este contexto, puede verse que uno o ambos operandos de los tercetos pueden ser un valor concreto, un identificador o una referencia a otro terceto (en el caso de expresiones, por ejemplo).

Para facilitar las operaciones con los tercetos desde el código se implementó una clase *helper* (*IntermediateCodeManager*) con el fin de administrar la cola de los tercetos despejando el código del analizador sintáctico ya que la administración se realiza en la clase [figura 1]. Esta clase posee una cola con tercetos y la administra cada vez que se insertan, se quitan o se modifican valores de los tercetos, ya que existen casos especiales como el de la regla *if*, *if-else* y *switch*, las cuales se detallarán a continuación.

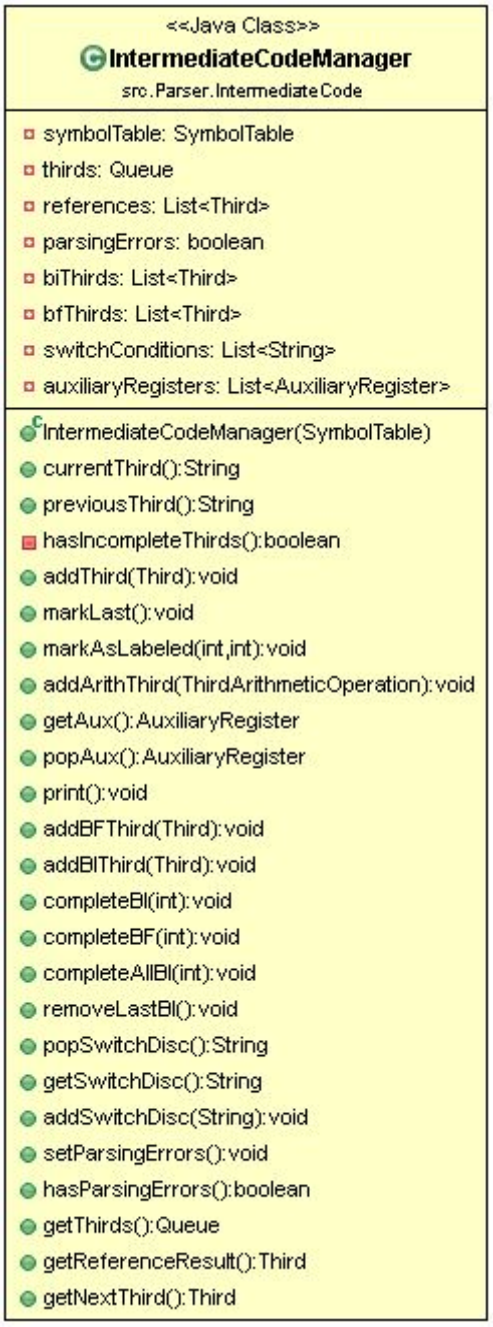

Figura 1: *administrador de tercetos.*

Además, para facilitar la generación de código de la próxima etapa se delega la responsabilidad a cada terceto: cada terceto sabe cómo generar su propio código assembler. Esto se lleva a cabo con el patrón de diseño *Strategy* simplemente agregando en la clase madre (*Third*) un método abstracto llamado *generateAssembler*, el cual deben implementar todas las clases que la extienden [figura 2].

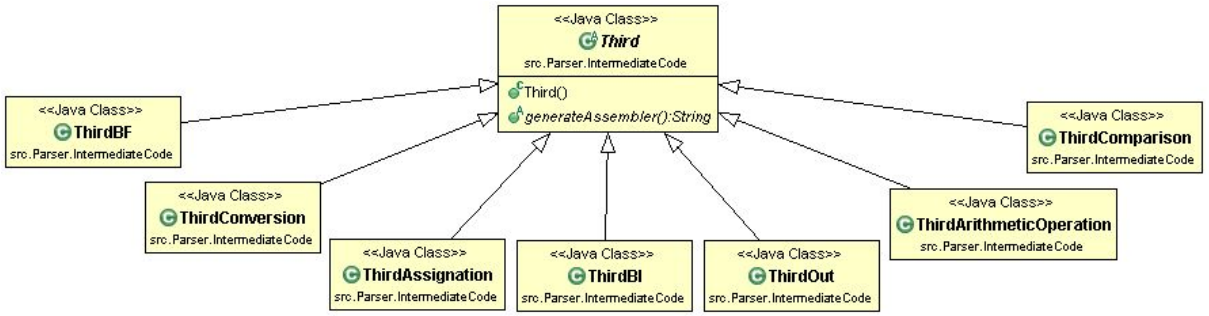

Figura 2: *clase Third con el método generateAssembler y las herencias de los tercetos particulares.*

Regla if

Los tercetos involucrados en esta regla deben agregarse a la cola de tercetos al igual que en cualquier otra regla. Sin embargo, existe un tratamiento especial y es que luego de la condición se crea un terceto BF, es decir, de salto por falso respecto a la condición. El problema es que la regla aún no sabe cuándo terminará el bloque posterior al *then*, con lo cual, una vez finalizada la regla de if, hay que completar el operando del tercetos *BF* creado previamente. Esto se puede resolver de forma muy simple utilizando las referencias de Java. Para ello se creó un método en la clase *IntermediateCodeManager* para agregar un terceto de tipo *ThirdBF* el cual lo agrega a la cola de tercetos, pero almacena su referencia en una fila adicional llamada *biThirds* y cuando se invoque el método *completeBF* se debe pasar por parámetro a la posición de los tercetos a saltar en caso de que este exista, el cual ya se sabrá dado que ese método se invoca una vez que finaliza la regla. El hecho de utilizar una cola para almacenar referencias de tercetos *BF* se debe a que puede ocurrir que existan sentencias de selección anidadas, entonces la solución más sencilla fue utilizando esta estructura, con lo cual siempre se opera con el terceto *BF* del tope al salir de la recursión de la regla.

Regla if-else

Esta regla es similar a la regla *if*, solo que se introduce la rama de salto por falso *else*. Para ello se utilizan los tercetos *BI*, o bien de salto incondicional al final de la regla cuando la condición haya sido verdadera y se haya ejecutado la rama del *then*. Se sigue la misma metodología al utilizar una fila adicional para los tercetos de tipo *BI*.

Regla Switch

La sentencia de control *switch* puede verse como el anidamiento de sentencias *if-else*. De esta forma, se utilizan los mismos tercetos que en la regla *if-else*: *ThirdBI* y *ThirdBF*.

Modificación de Reglas del Analizador Sintáctico

Si bien la estructura gramatical se definió en la entrega anterior y reconocía correctamente las distintas sentencias asignadas al grupo, en distintos casos fue necesario modificar algunas reglas con el fin de simplificar la generación de código intermedio. Estas

modificaciones consisten en separar las reglas en indirecciones para dar tratamientos específicos en esa instancia.

Las modificaciones se dieron en el caso de las reglas *if*, *if-else* y *switch*.

Regla if

La regla *if* era originalmente como se muestra a continuación:

```
<if_statement> ::=
        IF '(' <condition> ')' THEN <executional_statements> END_IF '.'
      | IF '(' <condition> ')' THEN <explicit_delimited_execution_block> END_IF
```

sin embargo, la creación del terceto *BF* se da luego del paréntesis de la condición y la completitud del terceto *BF* se da luego del bloque o sentencia de ejecución posterior el terminal *THEN*. Para realizar estas operaciones, la regla se reescribe como sigue:

```
<if_statement> ::=
         IF '(' <if_cond> ')' THEN <then_if_execution_statement> END_IF '.'
      |  IF '(' <if_cond> ')' THEN <then_if_execution_block> END_IF '.'

<then_if_execution_statement> ::= <executional_statements>
<then_if_execution_block> ::= <explicit_delimited_execution_block>
<if_cond> ::= <condition>
```

Como puede observarse, las modificaciones no son más que indirecciones a las reglas para aplicar tratamientos específicos. Las acciones semánticas se describirán posteriormente.

Regla if-else

La regla if-else era originalmente como se muestra a continuación:

```
<if_else_statement> ::=
        IF '(' <condition> ')' THEN <executional_statements> ELSE
<executional_statements> END_IF '.'
      | IF '(' <condition> ')' THEN <executional_statements> ELSE
<explicit_delimited_execution_block> END_IF '.'
      | IF '(' <condition> ')' THEN <explicit_delimited_execution_block> ELSE
<executional_statements> END_IF '.'
      | IF '(' <condition> ')' THEN <explicit_delimited_execution_block> ELSE
<explicit_delimited_execution_block> END_IF '.'
```

Sin embargo, al igual que con la regla *if* resulta difícil generar el código intermedio con esta gramática. Es necesario establecer indirecciones en la regla para poder aplicar en cada una de ellas acciones particulares. En este caso, el terceto *BF* se crea luego de la condición y en la etapa de las sentencias de ejecución posteriores al terminal *THEN* se completa el terceto *BF* y se crea el terceto *BI* de forma incompleta para ser completado luego del bloque de ejecución de la rama del else. La regla se reescribe como sigue:

```
<if_else_statement> ::=
```

```
            IF '(' <if_cond> ')' THEN <then_ifelse_execution_statement> ELSE
<else_execution_statement> END_IF '.'
      |     IF '(' <if_cond> ')' THEN <then_ifelse_execution_statement> ELSE
<else_execution_block> END_IF '.'
      |     IF '(' <if_cond> ')' THEN <then_ifelse_execution_block> ELSE
<else_execution_statement> END_IF '.'
      |     IF '(' <if_cond> ')' THEN <then_ifelse_execution_block> ELSE
<else_execution_block> END_IF '.'

<then_ifelse_execution_statement> ::= <executional_statements>
<then_ifelse_execution_block> ::= <explicit_delimited_execution_block>
<else_execution_block> ::= <explicit_delimited_execution_block>
<else_execution_statement> ::= <executional_statements>
```

Las acciones semánticas serán detalladas posteriormente.

Sentencia switch

La última regla modificada fue la descripción de la sentencia *switch*. Esto se debe a que los tercetos de esta sentencia son similares a los que serían generados por una sentencia de *if-else* anidados con un discriminante y comparaciones por igualdad.
Por ejemplo, los siguientes *snippets* tienen la misma semántica:

```
a,b,c:ulong.

switch(a) {
    case 1:
        b = c.
    case 2:
        c = b.
}.
```

```
a,b,c:ulong.

if (a == 1) then
    b = c.
else
    if (a == 2) then
        c = b.
    end_if.
end_if.
```

Como puede observarse, cada *case* es una equivalencia con un *if-else* anidado en la rama del else del *case* anterior y en el último case se genera un if sin un else. Todos los *BI* generados se dirigen hacia la la misma dirección: el *end_if* del *if* más externo; salen de los condicionales para ejecutar las próximas sentencias. Por otro lado, cada *BF* se dirige a la próxima condición del próximo *if* anidado. Para llevar a cabo esta tarea con los tercetos, es necesario modificar la gramática previamente definida, la cual era como se muestra a continuación:

```
<switch_statement> ::= SWITCH '(' ID ')' '{' <cases_rule> '}' '.'
<cases_rule> ::=
        <cases_rule> <case_rule>
      | <case_rule>
<case_rule> ::=
        CASE NUMERIC_CONST ':' <explicit_delimited_execution_block>
      | CASE NUMERIC_CONST ':' <executional_statements>
```

La modificación cambia la definición de la sentencia *switch* como sigue:

```
<switch_statement> ::= SWITCH '(' <switch_id> ')' '{' <cases_rule> '}' '.'
<switch_id> ::= ID
<cases_rule> ::=
          <cases_rule> <case_rule>
       |  case_rule

<number_switch_case> ::=
       NUMERIC_CONST

<case_rule> ::=
          CASE <number_switch_case> ':' <explicit_delimited_execution_block>
       |  CASE <number_switch_case> ':' <executional_statements>
```

Las acciones semánticas serán descritas posteriormente.

Operadores Posicionales

Los operadores posicionales de los elementos de la gramática fueron una de las herramientas más útiles en el compilador, ya que permitían acceder a los campos requeridos que eran completados en la recursión del elemento en cuestión y pueden ser obtenidos al finalizar la regla. Estos operadores se aplican a elementos terminales y no terminales de la gramática, pero son más provechosos cuando se los utilizan en elementos no terminales. Supongamos la siguiente regla:

```
foo: bar1 ‘<’ bar2 {
       $$.campo1 = $1.campo1;
       $$.campo2 = $3.campo1.
}
```

Puede verse que los *$1* y *$3* permite referenciar a *bar1* y *bar2*, respectivamente. El elemento *$$* se refiere a la misma regla y puede verse como el análogo al “retorno” de la regla, es decir, se setean los valores en los campos que corresponda y estos podrán ser utilizados en otras reglas que las utilicen al referenciarla posicionalmente. De esta forma, al entrar en la recursión de las distintas reglas, estas podrán resolver sus propios valores en base a su composición. Cabe soslayar que los distintos campos (en este caso *campo1* y *campo2*) son agregados a la clase *ParserVal* en donde se declaran los tipos y los valores iniciales de ser necesario. Estas no son más que variables públicas de la clase que se pueden acceder referenciando al objeto posicionalmente como se explicó previamente.

En nuestro caso, se amplió la clase *ParserVal* en los siguientes campos:

```
public Object firstOperand;
public Object secondOperand;
public StackReference stackPosition;
public String temporalType;
public boolean isConverted = false;
public int symbolTablePosition = 0;
//TP2.
```

```
public int begin_line;
public int end_line;
```

Los campos *operand1* y *operand2* sirven para facilitar el armado de los tercetos; como su nombre lo indica representan los operandos del teceto en cuestión, los cuales pueden ser una referencia a otro terceto en la pila (clase *StackReference*) o bien un identificador o una constante. A su vez, cada regla retorna también una posición en la pila de tercetos para que las reglas que la utilicen puedan saber en qué posición se encuentra. Esto se realiza al finalizar la regla: se inserta en la pila de tercetos y se devuelve la posición en la que fue insertado mediante la creación de un objeto *StackReference*. Luego, las reglas que la utilicen preguntan si su valor no es *null*; si es nulo, entonces es una constante o un identificador, ya que no se ha insertado en la pila de tercetos, pero si no es nulo, entonces es una expresión o un terceto que necesita ser referenciado.

El campo *temporalType* el cual sirve para validar la semántica respecto a los tipos en las operaciones. Esto es así ya que en la recursión, puede ocurrir que el tipo de una expresión, un identificador o una constante sea modificado. Si es una expresión, el tipo de ésta no puede consultarse en la tabla de símbolos ya que no está en ella, pero si es una constante o un identificador, si bien puede consultarse en la tabla de símbolos, también puede ocurrir que se haya realizado una conversión explícita, lo cual no representa un cambio en la tabla de símbolos. Para ello, se modifica su tipo sólo en la recursión de las reglas utilizando este campo y las otras reglas que lo utilicen podrán modificar este campo o utilizarlo para consultar el tipo de cada elemento.

Además, se agregó el atributo el booleano *isConverted* que sirve para saber si la expresión en cuestión ha sido convertida explícitamente según *UL_F*. Esto es así para ayudar a la generación de código intermedio, ya que cambia la utilización de los operandos.

Finalmente, se agregó el campo *symbolTablePosition*, el cual sirve para almacenar la posición en la que el token fue almacenado en la tabla de símbolos para poder generar el código correspondiente en lenguaje ensamblador, el cual se explicará previamente.

Acciones Semánticas

En las secciones previas se trataron puntos específicos respecto a la implementación. Sin embargo, la semántica asociada a cada sentencia y la resolución de la misma se discutirá en esta sección.

Más allá de los aspectos de implementación, la creación y mantenimiento de la pila de tercetos se muestra a continuación [Tabla 1].

Regla gramatical	Generación de Código Intermedio
`if_cond`	`ThirdBF third = new ThirdBf();` `codeManager.addBFThird(third);`
`then_if_execution_statement`	`codeManager.completeBF();`
`then_if_execution_block`	`codeManager.completeBF();`

then_ifelse_execution_statement	ThirdBI third = new ThirdBI();
	codeManager.addBIThird(third);
	codeManager.completeBF();

then_ifelse_execution_block	ThirdBI third = new ThirdBI();
	codeManager.addBIThird(third);
	codeManager.completeBF();

else_execution_block	codeManager.completeBI(); codeManager.markLast();
else_execution_statement	codeManager.completeBI(); codeManager.markLast();
assignation_statement	Third third = new ThirdAssignation(operand1, operand2, symbolTable, operand2.temporalType); codeManager.addThird(third);
out_statement	Third third = new ThirdOut(operando1, posicionEnTabladeSimbolos(operando1)); codeManager.addThird(third);

case_rule	Third third = new ThirdBI();
	codeManager.addBIThird(third);
	codeManager.completeBF();

number_switch_case	Third comparison = new ThirdComparison("==", switchDiscriminant, operand2, symbolTable, operand1.temporalType);
	codeManager.addThird(comparison);
	Third thirdBF = new ThirdBF();
	codeManager.addBFThird(thirdBF);

switch_statement	codeManager.completeAllBIThirds();
condition	Third third = new ThirdComparison(operator, operand1, operand2, symbolTable, operand1.temportalType); codeManager.addThird(third);
UL_F '(' expression_rule ')'	Third third = ThirdConversion(operand1); codeManager.addThird(third);
expression_rule	ThirdArithmeticOperation third = new ThirdAtithmeticOperation(operator, operand1, operand2, symbolTable, operand1.temporalType); codeManager.addThird(third);
term	ThirdArithmeticOperation third = new ThirdAtithmeticOperation(operator, operand1, operand2, symbolTable, operand1.temporalType); codeManager.addThird(third);

Tabla 1: *Creación de tercetos en la generación de código intermedio.*

Como puede verse, cada terceto tiene una clase específica que ayudará a facilitar la generación de código assembler. Una vez creado el terceto, se inserta en la fila de tercetos, contenida en la clase *IntermediateCodeManager* (*codeManager* en la tabla 1). Esta posee

utilidades como *addThirdBF*(), *addThirdBI*(), *addArithmeticThird()* entre otras, para mantener las referencias y poder completarlas previamente. Esto se consigue invocando el método *completeBI*, *completeBF* y *completeAllBI*. Éste último sirve para la utilización de la sentencia *switch*, ya que todos los saltos incondicionales apuntan a la finalización de la sentencia (sale de la estructura de control). Finalmente, el método *markLast()* marca el último terceto para luego facilitar la generación de etiquetas en assembler, ya que los saltos lo requieren. Para mayor detalle puede verse el código completo en el archivo *recursos/gramatica.y*.

Las cuestiones relativas a la implementación se verá en las subsiguientes secciones, en donde se detallarán los mecanismos utilizados para llevar a cabo la semántica de cada sentencia.

Declaración de Variables

La declaración de variables se lleva a cabo insertando los identificadores en una lista durante la recursión de las reglas declarativas y al finalizar, se declaran todas estas variables en la tabla de símbolos con el tipo dado. Esto es así ya que al momento de recorrer la lista de identificadores no se tiene aún el tipo, entonces es conveniente almacenarlas en una lista y una vez obtenido el tipo, se declaran iterando la estructura. Esto se refleja en la siguiente porción de código, escrito en la gramática:

```
declarative_statements:
	variables_declaration_statement
	{
		$$.begin_line = $1.begin_line;
		$$.end_line = $1.end_line;
		if (currentRuleError == 0) {
			//Marco como declaradas las variables en la tabla de símbolos.
			this.declareVars($1.temporalType); //Se itera la lista.
		}
		this.currentRuleError = 0;
	}
	…
	;

variables_declaration_statement:
		var_list ':' type '.'
		{
			$$.begin_line = $1.begin_line;
			$$.end_line   = $4.end_line;

			$$.temporalType = $3.sval;

		}
	…
	;


var_list:
		ID
		{
```

```
                $$.begin_line = $1.begin_line;
                $$.end_line   = $1.end_line;

                this.declaredVars.add($1.sval);
        }
|       var_list ',' ID
        {
                $$.begin_line = $1.begin_line;
                $$.end_line   = $3.end_line;

                this.declaredVars.add($3.sval);
        }
...
;
```

Posteriormente, se marcan en la tabla de símbolos como declaradas y se les asigna el tipo correspondiente. Una vez realizado esto, pueden utilizarse en lo que resta del código.

Detección de Tipos

Para detectar los tipos de los distintos elementos de la gramática, terminales o no terminales, se agrega un nuevo campo en la clase *ParserVal*, como se explicó en apartados anteriores. Sin embargo, en la primera ocurrencia de las constantes e identificadores se setea el valor dado desde la tabla de símbolos, y eventualmente este cambia en reglas superiores. Una vez seteado el valor, las reglas que utilizan las expresiones, constantes e identificadores pueden consultar su tipo directamente desde el valor del campo *temporalType* según su posición en la regla.

Identificadores

En todas las sentencias dadas, se verifica si los identificadores utilizados han sido declarado previamente o no. Si fueron declarados son utilizados pero en caso contrario se arroja error por la utilización de una variable no declarada. Esto se aplica para todas las sentencias descritas más adelante, con lo cual no hace falta volver a aclararlo.

Esto consiste únicamente en consultar en cada regla que tenga el terminal *ID* a la tabla de símbolos si un identificador con tal nombre ha sido declarado o no utilizando la función *isDeclared(String id)*.

Operaciones Aritméticas

Las operaciones aritméticas son realizadas únicamente entre identificadores, expresiones o constantes del mismo tipo, en cualquier combinación. En principio, una operación aritmética sólo puede realizarse mediante expresiones del mismo tipo dado que en nuestro grupo no poseemos conversiones implícitas. Para ello, se deben comparar los tipos de los dos operandos según el campo *temporalType*. De no coincidir, se arroja un error de tipos incompatibles, en caso de ser iguales, se procede a instanciar un terceto de operaciones aritméticas según la clase *ThirdArithmeticOperation* en donde se pasa por parámetro al constructor el operador (‘/’,’*’,’+’ o ‘-’), los operandos (referencias a otros tercetos o bien lexema de constante/identificador), la tabla de símbolos y finalmente el tipo de los operandos (*ulong* o *float*). Para agregar a la fila de tercetos un terceto con una

operación aritemética, se utiliza la función *addArithmeticThird()* de la clase *IntermediateCodeManager* con el fin de crear una variable auxiliar para almacenar el resultado del terceto. La lógica para la generación del código assembler en base a su tipo se explicará más adelante ya que no es parte de la discusión del código intermedio.

Conversión Explícita

Nuestro grupo obtuvo como uno de los temas particulares la conversión de expresiones de tipo *ulong* a *float* mediante un operador de conversión *UL_F*. Al llegar a esta regla, simplemente se cambia el tipo temporal, ya que no es correcto cambiar el tipo de la variable/constante en la tabla de símbolos. Esto se debe a que el cambio de tipo se realiza únicamente en el contexto de la operación en curso y en posteriores reglas, el identificador/constante debe tener el mismo tipo que cuando se declaró, salvo en una instrucción en particular que se verá más adelante. Por otro lado, en caso de que sea una expresión, también cambia su tipo temporal cubriendo así todos los posibles casos.

Comparaciones

Las comparaciones forman parte de la estructura sintáctica de nuestro lenguaje ya que es utilizado en sentencias de control como *if*, *if-else* y *switch*. Este terceto es necesario ya que expresa una comparación entre dos operandos que pueden ser constantes, identificadores o expresiones (referencias a otro terceto en la pila). Este terceto tendrá una función específica a la hora de generar código assembler. A la hora de crear un terceto de comparación, dado por la clase *ThirdComparison* es necesario pasar por parámetro al constructor el operador de comparación (‘<’, ‘>’, ‘<=’,’>=’,’==’,’<>’), los operandos, la tabla de símbolos y el tipo de los operandos, ya que esto influye en el juego de instrucciones a utilizar a la hora de generar el código assembler.

Nuevamente, las comparaciones sólo pueden realizarse entre operandos de mismo tipo, con lo cual de no coincidir se arroja un error de incompatibilidad de tipos.

Sentencia de Control *if*

La sentencia *if* requiere de un único terceto, el cual es representado por la clase *ThirdBF*. Este sirve para generar un salto por falso, haciendo que todas las reglas dentro de la rama del *then* no sean ejecutadas si la comparación resulta en falso.

Sentencia de Control *if-else*

La sentencia *if-else* requiere de dos tercetos fundamentales: *ThirdBF* y *thirdBI* los cuales representan el salto por falso y el salto incondicional al final del else cuando se deben ejecutar las instrucciones de la rama del then, respectivamente. El mecanismo para apilar y desapilar los tercetos BF y BI se explicó previamente, y no hay otra acción semántica para discutir en este respecto.

Sentencia de Control *switch*

Esta sentencia es, semánticamente hablando, similar a *if-else* anidados en otras ramas *else*. Sin embargo, existen algunas consideraciones a tener en cuenta, ya que a diferencia de las otras reglas, los *cases* del *switch* poseen una constante numérica, cuyo tipo debe ser contrastada con el tipo del discriminante (variable de control del *switch*) y

deben ser iguales. Es necesario tener en cuenta que pueden existir switches anidados, con lo cual resulta provechoso utilizar una pila de tipos de condiciones en la clase *IntermediateCodeManager*, entonces al entrar en recursión por reglas anidadas, siempre se apila el tipo del discriminante y se lo quita al finalizar el switch. De esta forma, siempre que se solicite el tipo del discriminante, se obtendrá el último ingresado y así se puede comparar en cada regla *case* con la constante en cuestión.

Sentencia de Asignación Simple

Este tipo de sentencias es muy sencillo de cubrir ya que el identificador del lado izquierdo debe tener el mismo tipo que la expresión en al lado derecho, o de lo contrario se debe informar un error de incompatibilidad de tipos. De coincidir los tipos, se genera un terceto representado por la clase *ThirdAssignation* en donde se pasan los dos operandos, identificador de izquierda y expresión derecha (la cual puede ser una referencia a otro terceto), la tabla de símbolos y el tipo de alguno de los operandos.

Sentencia de Asignación con *let*

Esta sentencia, a diferencia de la asignación simple tiene una lógica más engorrosa. En este caso existen dos propiedades a llevar a cabo en el lenguaje ya que la semántica de la sentencia *let* puede desglosarse en dos partes: inferencia de tipos y *shadowing*.

La técnica de shadowing consiste en “oscurecer” la última declaración de la variable en cuestión. Entonces, utilizando la sentencia *let*, es posible redefinir la variable y de ahí en más utilizar la última declaración con el nuevo tipo, sin afectar a instrucciones anteriores. Supongamos, por ejemplo la siguiente porción de código:

```
a,b:ulong.
a=b+a. [válido por ser ambos de tipo ulong.]
LET a = 5,. [válido por shadowing. Ahora a es de tipo float por inferencia de tipos.]
a=a+8,. [válido por ser ambos de tipo float.]
a=a+b. [inválido porque b es de tipo ulong y a es de tipo float]
```

Puede verse que una vez que se utiliza let en una asignación, la variable en cuestión es redefinida con un nuevo tipo. Ahora bien, esto hace necesaria la inferencia de tipos, lo cual es algo realmente simple: el identificador del lado izquierdo será ahora del tipo de la expresión del lado derecho, con lo cual se debe preguntar cuál es su campo *temporalType* según la posición de la regla en la gramática y ese será el tipo inferido.

Por otro lado, deben realizarse algunas verificaciones adicionales. Si una variable es declarada con un tipo en cuestión, no puede tener una sentencia *let* cuyo tipo inferido sea del mismo anterior ya que arrojará un error. Ahora bien, si el último tipo definido es distinto, esto es válido. Los siguientes códigos de ejemplo dan cuenta de ello:

a:ulong. LET a = 5,. [bien.] LET a = 5. [bien.] LET a = 1,84, [bien.]	a:ulong. LET a = 5. [mal. Ya era ulong.]

`LET a = ,58 [mal. ya era float.]`	

Todos estos cambios deben verse reflejados en la tabla de símbolos ya que, efectivamente se está redefiniendo una variable. El término “oscurecer” nos sugiere que la variable con el tipo anterior no debe desaparecer de la tabla de símbolos, sino simplemente ocultarse en futuras instrucciones. Esto se resuelve fácilmente ya que nuestra tabla de símbolos mantiene una estructura cuyo identificador puede tener múltiples símbolos según *Map<String, List<Symbol>>*. Entonces, al encontrarse con una sentencia *let*, simplemente se agrega a la lista (que funciona como una estructura *FIFO*) un nuevo símbolo con el nuevo tipo. Una vez realizadas estas tareas se crea un terceto de tipo *thirdAssignation* al igual que en la asignación simple, excepto que esta vez el tipo dado será el del tipo inferido (el de la expresión de la derecha).

Casos de Prueba

Sólo se prueban programas válidos, ya que de ocurrir al menos un error de parsing no se genera código intermedio.

Número de archivo	Sub-índice	Caso de Prueba	Arroja error	Resultado
1	1	Sentencias declarativas de múltiples variables.	No.	No se generan tercetos. Se genera código assembler con el volcado de la tabla de símbolos.
1	2	Sentencias declarativa con redefinición de variables.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por redefinición de variables.
2	1	Asignaciones simples.	No.	Genera los tercetos y el código assembler correctamente.
2	2	Asignaciones simples con dinstintos tipos.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por tipos diferentes.
3	1	Asignaciones con la sentencia LET.	No.	Genera los tercetos y el código assembler correctamente.
3	2	Asignaciones con la sentencia LET.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por redefinición de variables ya que el shadowing funciona si previamente la última definición de la misma no tiene el mismo tipo inferido.

4	1	Operaciones aritméticas varias.	No.	Genera los tercetos y el código assembler correctamente.
4	2	Operaciones aritméticas varias entre tipos diferentes.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por tipos diferentes.
5	1	Sentencia if con y sin anidamientos.	No.	Genera los tercetos y el código assembler correctamente.
5	2	Sentencia if-else con y sin anidamientos.	No.	Genera los tercetos y el código assembler correctamente.
5	3	Sentencia switch con y sin anidamientos.	No.	Genera los tercetos y el código assembler correctamente.
5	4	Sentencias de control con comparaciones entre tipos distintos.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por tipos diferentes.
6	1	Conversión explícita UL_F en varios contextos.	No.	Reconoce las reglas correctamente. El parsing finaliza con 0 errores.
6	2	Conversión explícita UL_F de una expresión que ya es float.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por tipos diferentes.
7	1	Programa que utiliza variables sin declarar.	Sí.	No se generan tercetos ni código assembler. Se arroja un error semántico por variables no definidas.

Consideraciones

1. Los tercetos *BF* guardan como primer componente la referencia a la condición que le antecede. Sin embargo, esta referencia no es utilizada en la implementación y existe pura y exclusivamente para imprimir por pantalla el terceto como se solicita en la cátedra.
2. Las sentencias *switch* poseen un último *BI* innecesario. Si bien esto es redundante, no afecta al flujo del programa ya que se ejecuta correctamente.

Generación de Assembler

La etapa de generación de código Assembler fue una de las etapas más complejas del trabajo. Si bien el lenguaje assembler no es complicado en sí mismo, la lectura de los tercetos y la generación del código a partir de ellos, con las combinaciones posibles que puedan surgir hace que esta tarea se vuelva tediosa.

El código generado es compatible con la arquitectura de procesadores *x86* y *x87* ya que en nuestro caso operamos con elementos de punto flotante teniendo que ampliar el juego de instrucciones de los procesadores *8086*.

Diseño e Implementación

Para implementar este requerimiento se delega la responsabilidad a cada terceto: cada terceto sabe cómo generar su propio código assembler, como se explicó en la sección de diseño e implementación del código intermedio. En este contexto, la generación de código assembler se reduce a meras plantillas [tabla 2], es decir, porciones de códigos de assembler almacenados en *strings* con las variantes necesarias según el caso. Esta lógica es gestionada por cada terceto.

Sentencia de alto nivel	Tipos	Código Assembler	Explicación
Sentencias declarativas			
`a,b:ulong.`	a y b ulong.	`_UI_a DD ?` `_UI_b DD ?`	Se solicita espacio en memoria para las variables a y b, sin un valor conocido (*?*). Su tipo es *DD* dado que son palabras de 32 bits. Las variables y constantes *ULONG* son antecedidas con la palabra *_UI_*.

a,b:float.	a y b float.	_FP_a DD ? _FP_b DD ?	Se solicita espacio en memoria para las variables a y b, sin un valor conocido (*?*). Su tipo es *DD* dado que son palabras de 32 bits aunque también podrían ser de tipo *REAL4* el cual las define como un tipo real (punto flotante) de 4 bytes. Las variables y constantes *FLOAT* son antecedidas con la palabra *_FP_."*
OUT(“Hola mundo!”).		ST_STRING_ID_1 db "Hola mundo!", 0	Se almacena en memoria un string cuyo nombre se antecede por *ST_STRING_ID* concatenado con la posición en la que fue insertado en memoria el token de string. Esto es así para identificarlo de alguna manera ya que el lexema puede contener caracteres especiale so ser muy largo.
Asignaciones			
a=b	a y b ulong.	MOV ebx, _FP_b MOV _FP_a, ebx	Se mueve el segundo operando a un registro auxiliar *ebx* (*MOV* no es una instrucción *mem-to-mem*) y luego se almacena en la sección de memoria correspondiente a la variable del lado izquierdo el valor del registro.
a=b	a y b float.	FLD _FP_b FST _FP_a	Se carga la variable b en la pila del co-procesador y luego se almacena ST(0), es decir, el valor de la variable b en la variable a.
Conversiones			
UL_F(a)	a de tipo ulong.	FILD _UI_a	Apila la variable a en el co-procesador para futuras operaciones. Las operaciones subsiguientes deben tener en cuenta que uno de sus operandos ya está en la pila.

Operaciones Aritméticas			
a+b	a y b de tipo float.	FLD _FP_a FLD _FP_b FADD FCOM FP_max FSTSW ax SAHF JA sum_overflow FST aux1	Se cargan los operandos en la pila del co-procesador utilizando *FLD* dado que ambos operandos son de punto flotante. Se suma *ST(0)* y *ST(1)* utilizando *FADD* y posteriormente se compara con el máximo de punto flotante para verificar si hubo overflow. Si no hay overflow, se almacena el resultado de la suma en la variable auxiliar *aux1*.

	a y b de tipo ulong.	MOV eax, _UI_a
		ADD eax, _UI_b
		MOV aux1, eax
		JC sum_overflow
			Se carga la variable a en el registro *eax* y luego se suma *eax* y la variable b utilizando la instrucción *ADD*. Esto es así ya que *ADD* no opera con operandos en memoria, sino que al menos uno de ellos tiene que ser un registro. El resultado es almacenado en *eax y* posteriormente se mueve el resultado a la variable auxiliar *aux1* y se evalúa el overflow mediante *JC*.

	a convertida de ulong a float y b float.	FLD _FP_b FADD FCOM FP_max FSTSW ax SAHF JA sum_overflow FST aux1	Como previamente se cargó la variable entera a en stack del co-procesador utilizando el comando *FILD* en el código generado por la conversión, ahora no se debe volver a apilar ya que se lo utilizará implícitamente. Posteriormente, se apila la variable b y se procede a sumar y a verificar el overflow.

	a float y b convertida de ulong a float.	FLD _FP_a FADD FCOM FP_max FSTSW ax SAHF JA sum_overflow FST aux1	Como previamente se cargó la variable entera b en stack del co-procesador utilizando el comando *FILD* en el código generado por la conversión, ahora no se debe volver a apilar ya que se lo utilizará implícitamente. Como el orden de los factores no influye en una suma, no hay que invertir el orden de éstos, entonces *a+b=b+a* por propiedad conmutativa de la suma. Posteriormente, se apila la variable a y se procede a sumar y a verificar el overflow.
	a y b convertidas de ulong a float.	FADD FCOM FP_max FSTSW ax SAHF JA sum_overflow FST aux1	Como las variables enteras a y b ya fueron apiladas en el co-procesador por la instrucción *FILD* desde la conversión, no hace falta volver a apilarlos. Nuevamente, en este caso no importa el orden de la suma. Luego se verifica el overflow y se almacena el resultado *ST(0)* en la variable auxiliar *aux1*.
a-b	a y b de tipo float.	FLD _FP_a FLD _FP_b FSUB FST aux1	Se cargan los operandos en la pila del co-procesador utilizando *FLD* dado que ambos operandos son de punto flotante. Se realiza la operación *ST(0) = ST(1) - ST(0)*. Luego se almacena el resultado en *aux1*.

	a y b de tipo ulong.	MOV eax, _UI_a
		SUB eax, _UI_b
		XOR edx, edx
		CMP eax, edx
		JL ui_negative_result
		MOV aux1, eax
			Se carga la variable a en el registro *eax* y luego se resta *eax* y la variable b de la forma *eax = eax - b* utilizando la instrucción *SUB*. Esto es así ya que *SUB* no opera con operandos en memoria, sino que al menos uno de ellos tiene que ser un registro. Posteriormente se

			mueve el resultado a la variable auxiliar *aux1* y se evalúa si quedó un resultado negativo utilizando el salto *JL (*los resultados en *ulong* no pueden ser negativos*)*.

	a convertida de ulong a float y b float.	`FLD _FP_b`
		`FSUB`
		`FST aux1`
		`MOV ebx, aux1`
			Como previamente se cargó la variable entera a en el stack del co-procesador utilizando el comando *FILD* en el código generado por la conversión, ahora no se debe volver a apilar ya que se lo utilizará implícitamente. Posteriormente, se apila la variable b y se procede a restar utilizando *FSUB* y luego moviendo el resultado en *ST(0)* a la variable auxiliar *aux1*.

	a float y b convertida de ulong a float.	`FLD _FP_a`
		`FXCH ST(1)`
		`FSUB`
		`FST aux1`
			Como el orden de la resta sí importa, no es lo mismo realizar *a-b* que *b-a*. En este caso, el segundo operando es una conversión excplícita de b que ya fue insertado en el stack del co-procesador. En este contexto, al apilar la variable a, la resta se realizaría de la forma *b-a*, pero lo deseado es a-b por lo que al apilar a, hay que intercambiar los elementos de *ST(0)* a *ST(1)* y viceversa.

	a y b convertidas de ulong a float.	`FSUB` `FST aux1`	Los dos operadores ya han sido insertados en el stack del co-procesador, con lo cual sólo deben utilizarse implicitamente con *FSUB*. Luego se procede a mover el resultado a la variable auxiliar *aux1*.

a*b	a y b de tipo float.	FLD _FP_a FLD _FP_b FMUL FCOMP FP_max FSTSW ax SAHF JA mult_overflow FST aux1	Se cargan los operandos en la pila del co-procesador utilizando *FLD* dado que ambos operandos son de punto flotante. Se realiza la operación *ST(0) = ST(0) * ST(1)* utilizando *FMUL* y posteriormente se compara con el máximo de punto flotante para verificar si hubo overflow. Si no hay overflow, se almacena el resultado de la suma en la variable auxiliar aux1.

	a y b de tipo ulong.	MOV eax, _UI_a
		MUL eax, _UI_b
		MOV aux1, eax
		JC mul_overflow
		MOV ebx, aux1
			Se carga la variable a en el registro *eax* y luego se multiplica *eax* y la variable b utilizando la instrucción *MUL*. Esto es así ya que *MUL* no opera con operandos en memoria, sino que al menos uno de ellos tiene que ser un registro. El resultado es almacenado en *eax y* posteriormente se mueve el resultado a la variable auxiliar *aux1* y se evalúa el overflow mediante *JC*.

	a convertida de ulong a float y b float.	FLD _FP_b FMUL FCOMP FP_max FSTSW ax SAHF JA mult_overflow FST aux1	Como previamente se cargó la variable entera a en stack del co-procesador utilizando el comando *FILD* en el código generado por la conversión, ahora no se debe volver a apilar ya que se lo utilizará implícitamente. Posteriormente, se apila la variable b y se procede a multiplicar y a verificar el overflow.

	a float y b convertida de ulong a float.	FLD _FP_a FMUL FCOMP FP_max FSTSW ax SAHF JA mult_overflow FST aux1	Como previamente se cargó la variable entera b en stack del co-procesador utilizando el comando *FILD* en el código generado por la conversión, ahora no se debe volver a apilar ya que se lo utilizará implícitamente. Como el orden de los factores no influye en una multiplicación, no hay que invertir el orden de éstos, entonces *a*b=b*a* por propiedad conmutativa de la multiplicación. Posteriormente, se apila la variable a y se procede a multiplicar y a verificar el overflow.
	a y b convertidas de ulong a float.	FMUL FCOMP FP_max FSTSW ax SAHF JA mult_overflow FST aux1	Como las variables enteras a y b ya fueron apiladas en el co-procesador por la instrucción *FILD* desde la conversión, no hace falta volver a apilarlos. Nuevamente, en este caso no importa el orden de la multiplicación. Luego se verifica el overflow y se almacena el resultado *ST(0)* en la variable auxiliar *aux1*.
a/b	a y b de tipo float.	FLDZ FCOMP _FP_b fstsw ax sahf jz division_by_zero FLD _FP_a FLD _FP_b FDIV FST aux1	Se carga el valor en el stack utilizando *FLDZ* y se compara con el divisor, es decir, la variable b. Si coinciden, se salta al tag de error por divisón por cero mediante la instrucción *JZ*. Si la división es válida, se cargan los operandos en la pila del co-procesador utilizando *FLD* dado que ambos operandos son de punto flotante. Se divide *ST(0)* y *ST(1)* utilizando *FDIV* y posteriormente se almacena el resultado de la

			suma en la variable auxiliar aux1.

	a y b de tipo ulong.	`MOV eax, _UI_a`
		`XOR edx, edx`
		`CMP edx, _UI_b`
		`JE division_by_zero`
		`DIV _UI_b`
		`MOV aux1, eax`
			Se carga la variable a en el registro *eax.* Luego se setea el registro edx en cero mediante la operación *xor* para compararlo con el segundo operando, la variable b. Si coinciden, se está en precensia de una división por cero, con lo cual se salta al *tag* que alerta una división por cero. Si el divisor es distinto de cero, se efectúa la división de *a/b* mediante el operando *DIV*, moviendo el resultado obtenido en *eax* a la variable auxiliar *aux1*.

	a convertida de ulong a float y b float.	`FLDZ`
		`FCOMP _FP_b`
		`FSTSW ax`
		`SAHF`
		`JZ division_by_zero`
		`FLD _FP_b`
		`FDIV`
		`FST aux1`
			Se carga el valor en el stack utilizando *FLDZ* y se compara con el divisor, es decir, la variable b. Si coinciden, se salta al tag de error por divisón por cero mediante la instrucción *JZ*. Si la división es válida, se carga solo el divisor en la pila del co-procesador utilizando *FLD* dado que el primer operando fue introducido previamente por la conversión explícita. Se divide utilizando *FDIV* y posteriormente se almacena el resultado de la suma en la variable auxiliar *aux1*.

	a float y b convertida de ulong a float.	`FLDZ`
		`FCOMP`
		`FSTSW ax`
		`SAHF`
		`JZ division_by_zero`
		`FLD _FP_a`
		`FXCH ST(1)`
		`FDIV`
		`FST aux1`
			Se carga el valor en el stack utilizando *FLDZ* y se compara con el divisor, es decir, la variable b. Si coinciden, se salta al tag de error por divisón por cero mediante la instrucción *JZ*. Si la división es válida, se carga solo el dividendo en la pila del co-procesador utilizando *FLD* dado que el segundo operando fue introducido previamente por la conversión explícita. Ahora bien, si se realizara la división se estaría realizando *b/a*, con lo cual hay que intercambiar *ST(0)* y *ST(1)* utilizando *FXCH*. Luego Se divide utilizando *FDIV* y posteriormente se almacena el resultado de la suma en la variable auxiliar *aux1*.

	a y b convertidas de ulong a float.	`FLDZ`
		`FCOMP`
		`FSTSW ax`
		`SAHF`
		`JZ division_by_zero`
		`FDIV`
		`FST aux1`
			Se realiza el mismo chequeo del divisor que en los casos anteriores. Si es distinto de cero, simplemente se dividen utilizando *FDIV* y posteriormente se asigna el resultado ubicado en *ST(0)* en la variable auxiliar *aux1*.

Comparaciones			
a>b	a y b de tipo float.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JBE tag	Como los dos operandos son de tipo float, la comparación debe realizarse utilizando el co-procesador. Para ello basta con cargar el primer operando en el stack y luego contrastarlo con el segundo operando mediante *FCOM*. Luego se se setean los bits de *status* en el registo *ax* y eventualmente el flujo del programa salta a la

			etiqueta llamada *tag*. La comparación en este ejemplo se da utilizando *JBE*, la cual significa *Jump if Below or Equal* y es utilizada debido a que el número puede tener signo.
	a y b de tipo ulong.	`CMP _UI_a, _UI_b` `JLE label3`	Este caso es mucho más simple ya que ambos operandos son enteros. Aquí se comparan las dos variables mediante *CMP* y se utiliza *JLE (Jump if Less or Equal)* para tomar la decisión del salgo a la etiqueta correspondiente si los flags seteados por la comparación coincide.

	a convertida de ulong a float y b float.	`FLD _FP_b`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JBE tag`
			Similar a la primer solución con los dos operandos de punto flotante, solo que esta vez el primer operados ya estaba insertado en la pila por la instrucción de conversión explícita.

	a float y b convertida de ulong a float.	`FLD _FP_a`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JBE label4`
			Idem al caso anterior.

	a y b convertidas de ulong a float.	`FXCH ST(1)`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JBE label5`
			En este caso el orden de los operandos debe invertirse, con lo cual se utiliza *FXCH* ya que ambos fueron insertados previamente por la instrucción de conversión explícita.

a<b	a y b de tipo float.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JAE tag	Estos casos son similares a las soluciones explicadas previamente. La única diferencia radica en la utilización de *JAE (Jump if Above or Equal)* por ser un número signado y *JGE (Jump if Greater or Equal)* en el caso de operandos enteros.
	a y b de tipo ulong.	`CMP _UI_a, _UI_b` `JGE label3`	

	a convertida de ulong a float y b float.	FLD _FP_b FCOM FSTSW ax SAHF JAE tag	
	a float y b convertida de ulong a float.	FLD _FP_a FCOM FSTSW ax SAHF JAE label4	
	a y b convertidas de ulong a float.	FXCH ST(1) FCOM FSTSW ax SAHF JAE label5	
a>=b	a y b de tipo float.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JB tag	Estos casos son similares a las soluciones explicadas previamente. La única diferencia radica en la utilización de *JB (Jump if Below)* por ser un número signado y *JL (Jump if Less)* en el caso de operandos enteros.
	a y b de tipo ulong.	CMP _UI_a, _UI_b JL label3	
	a convertida de ulong a float y b float.	FLD _FP_b FCOM FSTSW ax SAHF JB tag	
	a float y b convertida de ulong a float.	FLD _FP_a FCOM FSTSW ax SAHF JB label4	
	a y b convertidas de ulong a float.	FXCH ST(1) FCOM FSTSW ax SAHF JB label5	
a<=b	a y b de tipo float.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JA tag	Estos casos son similares a las soluciones explicadas previamente. La única diferencia radica en la utilización de *JA (Jump if Above)* por ser un número signado y *JG (Jump if Greater)* en el caso de operandos enteros.
	a y b de tipo ulong.	CMP _UI_a, _UI_b JG label3	

	a convertida de ulong a float y b float.	FLD _FP_b FCOM FSTSW ax SAHF JA tag	
	a float y b convertida de ulong a float.	FLD _FP_a FCOM FSTSW ax SAHF JA label4	
	a y b convertidas de ulong a float.	FXCH ST(1) FCOM FSTSW ax SAHF JA label5	
a<>b	a y b de tipofloat.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JZ tag	Estos casos son similares a las soluciones explicadas previamente. La única diferencia radica en la utilización de *JZ (Jump if Zero)* por ser un número signado y *JE (Jump if Equal)* en el caso de operandos enteros. La utilización de *JZ* se debe a que *CMP* en realidad realiza una resta entre los números para realizar la comparación y en el caso que de cero es porque los operandos son iguales.
	a y b de tipo ulong.	CMP _UI_a, _UI_b JE label3	
	a convertida de ulong a float y b float.	FLD _FP_b FCOM FSTSW ax SAHF JZ tag	
	a float y b convertida de ulong a float.	FLD _FP_a FCOM FSTSW ax SAHF JZ label4	
	a y b convertidas de ulong a float.	FXCH ST(1) FCOM FSTSW ax SAHF JZ label5	
a==b	a y b de tipo float.	FLD _FP_a FCOM _FP_b FSTSW ax SAHF JNE tag	Estos casos son similares a las soluciones explicadas previamente. La única diferencia radica en la utilización de *JNE (Jump if Not Equal).*
	a y b de tipo ulong.	CMP _UI_a, _UI_b JNE label3	

	a convertida de ulong a float y b float.	`FLD _FP_b`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JNE tag`	

	a float y b convertida de ulong a float.	`FLD _FP_a`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JNE label4`	

	a y b convertidas de ulong a float.	`FXCH ST(1)`
		`FCOM`
		`FSTSW ax`
		`SAHF`
		`JNE label5`	

Impresión de cadenas de texto			
`OUT("Hola mundo")`	N/A	`invoke MessageBox, NULL, addr ST_STRING_ID_1, addr ST_STRING_ID_1, MB_OK`	Imprime un mensaje por pantalla. La cadena de texto debe estar en memoria ya que MessageBox debe recibir parámetros con direcciones de memoria.

Tabla 2: *plantillas de código Assembler.*

Si bien cada terceto sabe cómo generar su propio código, existe un administrador que se encarga de leer los tercetos, solicitarles su código y concatenarlos; la clase *GenerateAssembler*. Además, esta clase también agrega otras porciones de código necesarias, como el preámbulo del programa y las etiquetas de salto útiles para los chequeos en tiempo de ejecución solicitados. Es importante destacar que esta tabla es representativa de las plantillas de assembler utilizadas, pero existen variantes en nuestro código como por ejemplo a la hora de manipular constantes, entre otras configuraciones. Para ver en detalle el assembler, se debe generar el archivo correspondiente a cada programa.

Preámbulo

Esta parte del código Assembler es la que establece la arquitectura del procesador sobre el cual va a correr el programa (y por tanto el juego de instrucciones) y las inclusiones de las bibliotecas necesarias.

En principio, se incluyen dos tipos de información importantes: *.386* y *.model* el cual establece la compatibilidad con los procesadores *x86* y el modelo de memoria, respectivamente. Esta información es estática y es simplemente un string que antecede a cualquier variante del programa final. A continuación se inserta el código modelo:

```
.386
.model flat, stdcall
```

```
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
include c:\masm32\include\masm32rt.inc
```

Segmento de datos

Este segmento es literalmente el volcado de la información de la tabla de símbolos y de ciertos datos importantes para llevar a cabo los requerimientos solicitados. En principio, toda la información que volcamos en esta sección permite establecer un alias a una región de memoria que tendrá el ancho definido por el programador, dependiendo del tipo información que se almacene en dicha partición. En principio, nuestro grupo obtuvo los tipos de datos *float* y *ulong*, ambos de 32 bits con lo cual se utiliza el tipo *DD*. Ahora bien, en el caso de información de tipo *float* podría utilizarse el tipo *REAL4*, el cual fuerza a insertar un valor de punto flotante de 32 bits, pero a fines prácticos es indistinto.

Toda información almacenada en este segmento se ve antecedida por dos posibles casos: *_UI_* y *_FP_* si es de tipo *ulong* o *float*, respectivamente. Esto es así para permitir una clara diferenciación de los datos utilizados al momento de generar código a partir de los tercetos, así como para aportar legibilidad a la hora de leer el programa ya que en el fondo ambos son de tipo *DD* indiscriminadamente.

Para generar esta sección del programa, se lee la tabla de símbolos y se procede a generar el código necesario para cada entrada, como se explicó previamente y como puede verse en la plantilla de sentencias declarativas en la tabla 1. Sin embargo, también se inserta utilidades como por ejemplo, el máximo número *ulong* y *float* para chequear *overflow* en operaciones aritméticas, el cual fue un tema a tratar por el grupo. Además, las constantes son almacenadas en memoria para ser referenciadas de esta forma con algunas modificaciones: se reemplazan los caracteres ‘,’, ‘-’ y ‘+’ en el lexema de la constante por el símbolo ‘_’ para darle un nombre a la porción de memoria que almacena la constante y finalmente, se da un valor inicial al declararla el cual también tiene reemplazos: si el número comienza con una ‘,’ se le anteceden un 0 y si termina en una ‘,’ se le asigna un 0 luego. En cualquier caso se reemplaza la ‘,’ por un ‘.’ ya que este es el formato válido exigido por *MASM*.

Finalmente, se declaran las variables auxiliares definidas por cada operación aritmética registrada. Estas no poseen un valor inicial y serán antecedidas por *_FP_* o *_UI_* según sea su tipo. Para saber si la entrada en la tabla de símbolos es una variable auxiliar o no, se consulta a cada símbolo leído si está marcado como tal. Esta utilidad de la tabla de símbolo sirve pura y exclusivamente como decorativa para separar el código y darle una mejor estética. Además, se declaran las cadenas de texto en memoria ya que deben ser referenciadas luego por las alertas en el código. Para ello se vuelcan las cadenas de texto de la tabla de símbolos y se le solicita a cada token la posición en la que fue insertado en la tabla de símbolos para identificarlo de alguna manera en el futuro. Eso es así ya que un string puede contener caracteres especiales y el reemplazo por expresiones regulares puede acarrear ciertos problemas. Las cadenas de texto son antecedidas por *ST_STRING_ID_* y luego se concatena la posición en la tabla de símbolos.

Segmento de código

En esta sección es donde se encuentra toda la lógica escrita por el programador en el lenguaje de alto nivel. Todas las sentencias ejecutables son generadas en assembler desde la lectura de los tercetos, solicitandoles el código necesario con lo cual la clase *GenerateAssembler* no juega otro rol más que concatenar estas cadenas de texto.

Asignaciones

El terceto de asignaciones tiene una lógica simple que verifica si la expresión de la derecha existe o no, consultando si el valor correspondiente es *null*. Si es valor es *null* entonces se trata de una expresión que previamente fue insertada en el stack del co-procesador por tratarse de una conversión explícita (*a=U_F(b)*). En este caso sólo la variable del lado izquierdo es tipo *float* y sólo se ejecuta la instrucción *FST* a la variable a. En el caso de que la expresión de la derecha no sea *null*, pueden ocurrir tres casos: que sea un registro, que sea una constante o un identificador volcado en el segmento de datos (un operando en memoria) o bien que se trate de una referencia a otro terceto, en donde se pide la variable auxiliar de este resultado. En cualquiera de estos casos, se verifica que el resultado sea un registro del procesador y en caso contrario se mueven los valores de memoria al registro *ebx* y posteriormente, se mueve el valor de *ebx* a la variable a. Esto es así porque no es posible utilizar la instrucción *MOV* con dos operandos en memoria. El método *prepareConstants()* de la clase *ThirdAssignation* es un claro ejemplo de cómo la constante que está en memoria se mueve al registro *ebx* si es primer operando o *ecx* si es segundo operando para poder utilizarse luego. Es importante comentar que el registro *eax* debe ser utilizado con cuidado porque ahí se alojan resultados implícitamente según el tipo de instrucción utilizado, entonces para evitar conflictos no se lo utilizó salvo que fuera sumamente necesario.

Finalmente, luego de generar la cadena de texto correspondiente con el código assembler de la asignación, se consulta si el terceto está marcado para generar una etiqueta de salto. Si está marcada se concatena un string con un número de etiqueta dado al momento de la marca.

Operaciones aritméticas

Las variantes de la generación de código de operaciones aritméticas fueron de las más complejas de implementar y se mostraron las porciones de código válidas para los distintos casos en la tabla 1.

En principio, las funciones de generación de código se diferencian las de tipo de punto flotante y las de tipo entero largo sin signo. Las más sencillas de implementar fueron las operaciones con operandos de tipo *ulong* ya que lo único a tener en cuenta es que las constantes deben moverse de memoria a los registros *ebx* y *ecx* dependiendo de si son el primer o el segundo operando. Las instrucciones de punto flotante deben tener en cuenta los casos de conversiones explícitas en sus operandos ya que estos fueron insertados en la pila del co-procesador por una instrucción previa. Para saber si algún operando fue convertido o no, se evalúa la variable boolean *isConverted* que es pasada por parámetro desde la gramática, justamente con el campo homónimo agregado en *ParserVal*.

Sea cual sea el caso, los resultados son almacenados en un registro auxiliar asignado al momento de creación del terceto desde la gramática. La clase codeManager asigna una variable auxiliar a un terceto en particular al momento de agregar el terceto que representa una operación aritmética, como se explicó en este mismo informe. Entonces, una vez realizada la operación se mueve el resultado a la variable auxiliar, la cual se le solicita a la clase *CodeManager* mediante *getResultName()*. Ahora bien, si el resultado surge de una operación *ulong* se encontrará en el registro eax con lo cual se realiza la instrucción *MOV* y si el resultado es de punto flotante se utiliza la instrucción *FST*.

Finalmente, luego de generar la cadena de texto correspondiente con el código assembler de la asignación, se consulta si el terceto está marcado para generar una etiqueta de salto. Si está marcada se concatena un string con un número de etiqueta dado al momento de la marca.

Comparaciones

De igual forma que con los tercetos de operaciones aritméticas, existen distintas variantes respecto a los operandos: pueden ser ambos *ulong*, ambos *float* o de distinto tipo en sus dos posibles configuraciones. Por otro lado, debe tenerse en cuenta también que cuando se convierta un operando de *ulong* a *float*. Las distintas variantes pueden verse en la tabla 1.

Los tercetos de comparaciones (*ThirdComparison*) generan en su código la carga de la comparación en sí misma utilizando *CMP* si los operandos son de tipo *ulong* o *FCOM* cuando los operandos son de punto flotante. Estos tercetos sólo se encuentran en el contexto de una sentencia de control, con lo cual sirve exclusivamente para determinar si se toma un salto o no. Para ello, los tercetos también generan el código de salto por un criterio ya que esto depende del operador de comparación. Todos los casos de comparación se evalúan por caso contrario; si se desea comparar con el operador ‘>’, entonces el salto se realizará por menor o igual puesto que siempre se evalúa primero el salto por falso, es decir, un terceto de comparación siempre será precedido por un terceto de salto por falso (*ThirdBF*). Sin embargo, el comparador no sabe a qué label debe saltar, con lo que sólo se remite a generar el salto con un criterio correspondiente y el siguiente terceto de tipo BF concatena la etiqueta a saltar en caso de ser cumplirse la condición (contraria a la escrita por el programador en la sentencia de alto nivel).

Saltos condicionales

Los saltos condicionales surgen desde los tercetos *BF* (*ThirdBF*). Como se comentó previamente, son siempre antecedidos por una comparación, con lo cual es necesario que se concatene la etiqueta a la cual se debe saltar, la cual es la concatenación de la cadena “label” con el número de terceto al cual se debe saltar, marcado previamente en la etapa de generación de código intermedio.

Saltos incondicionales

Los saltos incondicionales son generados desde la clase *ThirdBI*. Consiste en utilizar la instrucción *JMP* a una posición de tercetos marcada en la etapa anterior.

Impresión de mensajes

La generación del terceto *ThirdOut* es trivial ya que solo se imprime el mensaje invocando a la rutina *MessageBox*, como puede verse en la tabla 1 simplemente agregando como parámetro las direcciones de memoria de la cadena previamente definida. Luego se genera la etiqueta de salto en caso de que esté marcada.

Casos de Prueba

Se utilizan los mismos casos de prueba válidos (los que no arrojan errores) de la etapa de generación de código intermedio, ya que generan todas las estructuras válidas del lenguaje. Por otro lado, a modo de mostrar el correcto funcionamiento de la detección de *overflow* en multiplicación y suma, detección de valores negativos en restas con operandos de tipo *ulong* y detección de división por cero en ambos tipos, se adjuntan nuevos códigos que den cuenta de ello en la siguiente tabla [tabla 3].

Número de archivo	Subíndice de archivo	Chequeo
1	1	Overflow en suma con operandos de punto flotante.
1	2	Overflow en suma con operandos ulong.
2	1	Overflow en multiplicación con operandos de punto flotante.
2	2	Overflow en multiplicación con operandos ulong.
3	1	Resultado negativo en restas con operandos ulong.
4	1	División por cero con operandos ulong.
4	2	División por cero con operandos de punto flotante.

Tabla 3: *chequeos en tiempo de ejecución.*

Consideraciones

1. En los casos de comparaciones y operaciones aritméticas, se tiene en cuenta cuando un operando *ulong* es convertido a *float* explícitamente mediante *UL_F*. Esto debería haber sido de otra manera, ya que nuestro grupo obtuvo el tema de variables auxiliares. Sin embargo, las soluciones se implementaron teniendo en cuenta justamente que una vez que el terceto de conversión realiza la instrucción *UL_F*, en lugar de verter ese valor en una variable auxiliar, se utiliza ese valor directamente desde la pila del co-procesador. Esto puede verse como una optimización del compilador que si bien no fue solicitada, el grupo la implementó de

igual forma, lo cual fue permitido por el docente con una explicación previa de lo que esto implica.

Conclusión

Esta segunda entrega del trabajo permitió al grupo comprender cómo los distintos lenguajes de programación genera un ejecutable a partir de un código de alto nivel. Esto es importante ya que nos permite conocer por qué los lenguajes que utilizamos a lo largo de la carrera tienen ciertas restricciones o problemas. Por otro lado, también nos deja ver cuán compleja es la realización de la etapa de generación de código assembler sabiendo que los compiladores de lenguajes comerciales generan distintos códigos para cada arquitectura soportada. Finalmente, es importante aclarar que al realizar las dos últimas etapas de compilador, entendimos que las optimizaciones en los lenguajes de alto nivel pueden implicar menor cantidad de instrucciones en el código de procesador y así funcionar más rápido con la misma semántica.

A modo de comentario, la etapa de generación de código assembler fue una de las más tediosas debido a la utilización del juego de instrucciones *x87* y las variantes que surgieron con sus operandos. Una autocrítica válida es que el grupo no siguió al pie de la letra la utilización de variables auxiliares, complicando más el desarrollo (aunque logrando un código assembler más optimizado). Esto nos sugiere que muchos de los requerimientos por parte de la cátedra no son al azar, sino que lejos de eso, sirven para facilitar el desarrollo al alumno al mismo tiempo que se brinda conocimiento.