

Universidad Nacional del Sur

Carrera: Licenciatura en Ciencias de la Computación

Informe de Proyecto

Lenguajes de Programación

Título del Proyecto:

Traducción de código Java a SimpleSem

Integrante:

Emalhao, Lautaro

Docente:

Cobo, María Laura

Federico Martin Schmidt

Materia:

Lenguajes de Programación

Año:

2025

El esquema de las tablas virtuales, las instancias de los objetos y los registros de activación de todos los métodos y constructores del trabajo es el siguiente:

Virtual Tables

VTA	VTB	VTC
VTOject	VTA	VTB
procesarLoteA	procesarLoteB	procesarLoteB
obtenerInformacionA	obtenerInformacionB	obtenerInformacionB
obtenerValorUnitarioA	obtenerValorUnitarioB	obtenerValorUnitarioB
verificarStockA	verificarStockA	verificarStockA
VTPrincipal	manejarPedido1B	manejarPedido1B
	manejarPedido2B	manejarPedido2B
		inicializarProductoPrincipalC
		cumplirOrdenComplejaC

Registro de activación de la clase Main

RA main
Puntero de retorno
Enlace dinámico
int solicitudInicial
int prioridadOrden
C gestorPrincipal

Registros de activación de la clase A

RA constructor de A
Puntero de Retorno
Enlace dinámico

RA obtenerValorUnitario de A
Puntero de Retorno
Enlace dinámico
this

RA procesarLote de A
Puntero de Retorno
Enlace dinámico
this
itemsProcesados
i
stock

RA obtenerInformacion(tipoInfo) de A
Puntero de Retorno
Enlace dinámico
this
tipoInfo

RA obtenerValorUnitario de A
Puntero de Retorno
Enlace dinámico
this

Registros de activación de la clase B

RA constructor de B
Puntero de Retorno
Enlace dinámico

RA obtenerValorUnitario de B
Puntero de Retorno
Enlace dinámico
this

RA obtenerInformacion(tipoInfo) de B
Puntero de Retorno
Enlace dinámico
this
tipoInfo

RA procesarLote de B
Puntero de Retorno
Enlace dinámico
this
resultadoBase
ajuste

RA manejarPedido1B (cantidadSolicitada) de B
Puntero de Retorno
Enlace dinámico
this
cantidadSolicitada

RA manejarPedido2B (cantidadSolicitada, prioridad) de B
Puntero de Retorno
Enlace dinámico
this
cantidadSolicitada
prioridad
costo

Registros de activación de la clase C

RA constructor de C	RA inicializarProductoPrincipal de C
Puntero de Retorno	Puntero de Retorno
Enlace dinámico	Enlace dinámico
ordenId	this

RA cumplirOrdenCompleja(cantidadNecesaria) de C
Puntero de Retorno
Enlace dinámico
this
cantidadNecesaria
itemsFaltantes
vecesProcesado
costoTotal
procesadoAhora

Instancias de objetos y Class Record

Instancia de A	Instancia de B	Instancia de C
Referencia a VTA	Referencia a VTB	Referencia a VTC
int itemId	int itemId	int itemId
int cantidadDisponible	int cantidadDisponible	int cantidadDisponible
int valorUnitario	int valorUnitario	int valorUnitario
	int ubicacionAlmacen	int ubicacionAlmacen
	int tipoProducto	int tipoProducto
		B productoPrincipal
		int nroOrden

Class Record de A
int totalItemsCreados = 0
int limiteProcesamiento = 100

La lógica de la traducción de `cumplirOrdenCompleja` de la clase C tiene como esqueleto la estructura de un ciclo `do-while`, en la cual primero se ejecutan una vez las instrucciones y al final se evalúa la condición para verificar si se vuelve a iniciar el loop o sigue la siguiente instrucción después del ciclo. Como la condición se evalúa en cortocircuito, hay dos jumps condicionales: el primero evalúa la primera condición y, si no se cumple, salta a la siguiente instrucción después del ciclo. Esto evita que se evalúe la segunda condición. Esta última se evalúa y, si se cumple, el PC vuelve al inicio del loop.

Dentro del ciclo `do-while`, hay dos condiciones que se evalúan siempre. La primera se compone de un jump que niega la condición: si se cumple, se ejecuta la siguiente instrucción que sería el bloque dentro del `if`; si no se cumple, salta a la etiqueta que evalúa la condición en el segundo `if`. Esta segunda sentencia se compone de un `if-else`, por lo que se optó por poner un jump condicional que salta a la etiqueta del `else` si no se cumple la condición. En caso de que se cumpla, se ejecuta el bloque correspondiente que está en la instrucción que le sigue al jump, y al final se salta a la etiqueta que suma uno a la variable “`vecesProcesado`” (este jump es útil para no ejecutar las instrucciones del `else`). Si no se cumple la condición, se ejecuta el bloque del `else` que consiste en un `break`, que se traduce como un jump al final del ciclo `do-while`.

Afuera del ciclo hay una condición que evalúa si “`itemsFaltantes`” es mayor a 0. Esta se evalúa negando la condición y haciendo un jump al retorno fuera del `if` si no se cumple “`itemsFaltantes>0`”. Si la condición se cumple, se ejecuta el `return` dentro del bloque del `if` y se salta al final del método para volver al registro de activación del llamador.

Ambos *return* almacenan el valor de retorno en el registro “`D[Actual-1]`”, que es un espacio libre en la memoria de datos antes de la base del registro de activación del método `cumplirOrdenCompleja`, con el propósito de guardar el retorno para que el llamador pueda acceder al valor.

La diferencia entre `cumplirOrdenCompleja` y `procesarLote` de la clase A radica en que, como en el segundo método la instrucción es un ciclo `while`, el bloque dentro del loop puede ejecutarse cero o más veces. En el caso de `cumplirOrdenCompleja` de la clase C, el loop se ejecuta una o más veces.

También es importante destacar que, en `procesarLote` de la clase A, hay una etiqueta que marca el comienzo del bucle, luego se evalúa la condición con un jump `true`: si esta se cumple, se ejecuta la siguiente instrucción que corresponde al bloque dentro del ciclo; caso contrario, salta a la etiqueta fuera del loop. Como la condición se evalúa al principio, la última instrucción del bloque `while` es un jump al comienzo del ciclo.

El mecanismo de selección múltiple del método `procesarLote` de la clase B evalúa primero todas las condiciones antes de ejecutar los bloques correspondientes. Se utiliza un salto condicional (`jumpT`) que dirige la ejecución a la etiqueta de un bloque si se cumple la condición. En caso de que ninguna condición se cumpla, se ejecuta el bloque por defecto. Este bloque contiene una instrucción que modifica una variable, seguida de un salto incondicional al final del `switch`, ya que las instrucciones siguientes corresponden a los demás bloques de los distintos `case`. Siempre que se produce un salto a uno de estos bloques, la última instrucción es un salto

al final del switch, lo que representa el comportamiento de los break, evitando que el flujo de ejecución continúe hacia las instrucciones de los siguientes casos.

En caso de incluir tipos de datos más complejos, deben tenerse en cuenta varias consideraciones al verificar una condición. A diferencia de los valores enteros (como en el caso de la traducción que se implementó), los tipos complejos requieren múltiples operaciones para evaluar la condición. Por ejemplo, al comparar cadenas, es necesario realizar una verificación carácter por carácter; en el caso de objetos, puede requerirse una comparación superficial o profunda, lo que implica implementar o invocar un método *equals()*. Además, se debe contemplar la posibilidad de referencias nulas, lo que agrega mayor complejidad al flujo de control. Para estos casos, se necesita usar registros temporales adicionales, más memoria, e implementar una lógica de control más elaborada, con varios saltos condicionales.

Los operadores con cortocircuito se implementan utilizando múltiples saltos condicionales, a diferencia de los operadores sin cortocircuito, que suelen resolverse con un único salto luego de evaluar todas las condiciones.

En el método `verificarStock` de la clase A se presentan dos condiciones evaluadas con cortocircuito: una con el operador AND y otra con OR. La diferencia entre ambas radica en cómo se gestionan los saltos condicionales durante la evaluación.

Si ninguna de las condiciones de los `if` se cumple, se retorna -1. Al final de cada bloque de código se incluye un salto incondicional hacia la parte de la traducción encargada de retornar al registro de activación del llamador.

En el caso del AND con cortocircuito, se utilizan tantos saltos condicionales como condiciones haya, negando cada una. Esto permite que, si alguna condición no se cumple, se realice un salto que evita la ejecución del bloque correspondiente dentro del `if`.

Por otro lado, en el caso del OR con cortocircuito, se coloca un salto condicional por cada condición, sin negarlas. Si alguna se cumple, se salta directamente al bloque del `if`. Para respetar la estructura general de bloques, la última condición puede negarse para evitar ejecutar el bloque en caso de que ninguna condición se cumpla. Si esta última condición se cumple, el flujo continúa naturalmente hacia el bloque siguiente, sin necesidad de salto.

De este modo, se garantiza que en un AND se omite el bloque si al menos una condición no se cumple, y que en un OR se ejecuta el bloque si al menos una condición se cumple.

Un aspecto importante a aclarar son los saltos que se realizan al invocar un método que accede a la virtual table del padre. Cuando el objeto que realiza la llamada es de tipo C, y se invoca un método heredado de B, el objeto pasado como referencia no coincide con el tipo esperado para acceder directamente a la virtual table del padre. Si se intentara acceder a la virtual table de *this* para luego obtener la del padre y aplicar el desplazamiento correspondiente, se generaría un bucle infinito, ya que siempre se terminaría invocando la etiqueta del método en la clase B.

Para generalizar la solución a este problema, se optó por invocar directamente la etiqueta de la virtual table de B para acceder a la virtual table de A (el padre) y aplicar el desplazamiento necesario. Por esta razón, en lugar de utilizar la expresión $D[D[H[Actual+2]+0]+0]+X$, se utiliza $D[D[VTB]+X]$, donde X representa el offset del método que se desea invocar.