# Java 的語法樹直譯器

周敬斐
淡江大學管理科學研究所
博士班研究生

郭肇安
中央研究院計算中心
系統工程師

廖賀田*
淡江大學資訊管理學系
副教授

*htliaw@mail.tku.edu.tw

## 摘要

Java 程式語言具有**物件導向**(object-oriented)、**可攜**(portable)與**多執行緒**(multi-threading)等優點，是當今最風行的程式語言之一。目前所有的 Java 原始檔都必須被編譯成稱為**位元組碼**(bytecode)的**中間語言**(intermediate language)，再交由 **Java 虛擬機**(java virtual machine)進行直譯。但位元組碼與 Java 的原始碼之間並沒有本質上的關聯，位元組碼在直譯上也沒有特別便利之處。

本論文主張以 Java **語法樹**(syntax tree)當成中間語言，並提出對它進行**直譯**(interpreting)的技術。語法樹是製作**剖析器**(parser)時就必須用到的資料結構，它在先天上就和原始碼有最密切的關連。我們直接在語法樹的各種節點上配置適當的**方法**(method)，運用**遞迴呼叫**(recursive call)與**動態繫結**(dynamic binding)來製作直譯器。我們也利用**例外機制**(exception mechanism)來脫離進入多層呼叫後的**控制流程**(control flow)。為配合目前被廣泛使用的位元組碼，我們還提出語法樹與位元組碼之間的銜接機制，讓這兩種不同的直譯機制可以互相呼叫。

本論文所提出的直譯機制非常容易製作，是開發直譯器的優良途徑。我們所實作的直譯器在大型程式上的真實運作已有良好的表現。由於語法樹與原始程式之間的對應既直接又自然，未來在製作**除錯**(debug)機制時也會比較容易。此外，還可以進一步在語法樹上配置圖形展示的能力，讓直譯器在程式語言與資料結構的教學上對學生提供直覺性的幫助。

關鍵字：Java、語法樹、直譯器

# A Syntax-Tree-Based Interpreter for Java

Ching-Fei Chou
Doctoral Student
Department of Management Sciences and Decision Making, Tamkang University

Chao-An Kuo
System Engineer
Computing Center, Academia Sinica

Heh-Tyan Liaw*
Associate Professor
Department of Information Management, Tamkang University

*htliaw@mail.tku.edu.tw

## Abstract

The Java programming language, with advantages such as object-orientation, portability and multi-threading, is one of the most popular programming languages. In current time, the Java source code has to be compiled into an intermediate language, the bytecode, before it can be interpreted by the Java virtual machine. However, there is no intrinsic relationship between bytecode and its source code, and there is no special benefit in interpreting the bytecode either.

In this paper, we recommend adoption of the syntax tree to be the intermediate language for Java, and we propose techniques for interpreting it. The syntax tree is an essential data structure when parsing the source code. Furthermore, it bears strongest relation with the source code. By appending suitable methods into various types of nodes in the syntax tree, the interpreter can be implemented through recursive call and dynamic binding. We also utilize the exception mechanism to quit from the control flow of deep method calls. In order to cooperate with the popular bytecode, we propose a bridge mechanism to interconnect the interpretations of both bytecode and the syntax tree.

It is very easy to build an interpreter based on the syntax tree. Our syntax-tree-based interpreter for Java has been successfully implemented, and it works well on interpreting practical Java software. Since the correspondence between the syntax tree and the source code is direct and natural, it will be easier to implement a debugging mechanism in the future. Furthermore, we can equip graphical rendering functions into the syntax tree, so that the interpreter can provide instinctive assistance in teaching the Java programming language and data structures.

Keyword: Java, Syntax Tree, Interpreter.

# 壹、 緒論

## 一、研究動機與目的

Java 程式語言(Sun Microsystem, Inc., 2007)具有**物件導向**(object-oriented)、**可攜**(portable)與支援**多執行緒**(multi-threading)等特點，近年來已經廣為流行用來開發多種應用程式。目前所有的 Java 原始碼都是經過**編譯器**(compiler)轉換成稱為**位元組碼**(bytecode)的**中間語言**(intermediate language)，編譯後的位元組碼交由 **Java 虛擬機器**(Java Virtual Machine)(Lindholn, and Yellin, 1999)以**直譯**(interpreting)的方式來執行。

語法樹是製作**剖析器**(parser)時就會用到的**資料結構**(Aho, Sethi, and Ullman, 1985)，利用語法樹來進行直譯就可以不必再引用額外的中間語言。以 Java 語言來說，它的中間語言(位元組碼)與高階語法之間並沒有什麼特別的關聯，在直譯上也沒有特別便利之處。開發 Java 語言的昇陽公司(Sun Microsystem, Inc.)聲稱引用位元組碼使 Java 具有可攜性。但其實只要是採用任何一種中間語言(例如 Pascal 語言的 P-code)，並對各個平台製作對應的編譯器與直譯器，則這種中間語言照樣會具有可攜性。對 Java 來說，在所有可能的中間語言之間，只有它的語法樹才是最自然、最合理的中間語言。

程式師在撰寫程式的過程常常需要進行**除錯**(debug)。各種**整合型開發環境**(integrated development environment, IDE)通常都會配置除錯功能(Eclipse Foundation , 2008)。程式師在原始程式的某些位置設定一些**中斷點**(break points)，IDE 則在中間語言的各個對應位置安排中斷的機制。當中間語言與原始檔在結構與順序上有差異時，中斷機制的製作就會變得麻煩。而當我們採用語法樹當做中間語言時，原始碼與中間語言之間的對應既直接又自然，將來要在語法樹的節點配置除錯機制更是輕而易舉。

在程式教學上，若能伴隨著一步步的分解動作將資料結構的圖形逐步顯示出來，對學生將會有很大的幫助。在普通的中間語言上，像這種功能在實作上等於是要另寫一套直譯器。而在採用語法樹當做中間語言時，只要在語法樹的變數節點上添加圖形展示的方法，則直譯器配合除錯機制就自動具備展示資料結構圖形的能力。

普通的直譯器是以**指令週期**(instruction cycle)來執行，過程包含**讀取-解碼-執行**(fetch-decode-execute)。這種直譯程序通常是依中間碼循序進行，只有遇到跳躍指令或呼叫指令才會改變直線順序。本論文所提出的樹形直譯器直接對**語法樹**(syntax tree)進行直譯，這就變成在**巡行**(traverse)一個**樹形資料結構**(tree)。眾所週知：樹形資料結構的巡行可以用遞迴呼叫(recursive call)來做，也可以利用堆疊或額外的指標來引導。用遞迴呼叫來做非常方便，而且可以利用物件導向程式的**動態繫結**(dynamic binding)技術，讓各種節點使用適合於自己的處理方式。這種做法的困難在於處理進入迴圈後的 break 敘述以及 return 敘述。對此，我們發現物件導向語言的**例外機制**(exception mechanism)剛好可以用來做處理。

本論文提出以 Java 語法樹當做中間語言的 Java 直譯器，並討論遞迴式的直譯機制。

### 二、相關研究

BlueJ (BlueJ-The interactive java environment, 2007)是由澳洲墨爾本 Monash 大學的 BlueJ 小組設計與開發，這個工具主要是在課堂上作為 Java 程式的教學輔助工具。該工具提供了整合式的操作環境。程式員可以在圖形編輯器中繪製類別關係圖，讓 BlueJ 將圖形轉換成簡單的程式框架。此外，BlueJ 還可以透過圖形化介面創建物件並觀察物件在執行期的狀態。

BeanShell (BeanShell-Lightweight Scripting for Java, 2007)內建簡單的指令，藉由這些指令可以執行 Java 敘述與算式。它有四種執行模式：命令列、控制台、Applet 與遠端對話伺服器，藉此提供程式員針對簡單的 Java 程式進行偵錯。

DynamicJava (DynamicJava, 2002)是以 Java 語言所開發的互動式 Java 程式碼直譯器(a java source interpreter)，它提供了一個圖形使用者介面讓程式員直接撰寫簡單的 Java **敘述**(statement)來測試程式。

上述的軟體雖然可以執行 Java 原始碼，但它們都須先將原始碼編譯成位元組碼後才能執行，因此無法在**執行期**(runtime)時動態地增加類別或是編修原始碼。

### 三、論文概觀

我們在第二節介紹語法樹的主要單元，包含類別、欄位、函數、建構元與區域變數的類別結構。第三節則介紹如何對**算式**(expression)進行**計值**(evaluation)，其中包含各種運算符號的計值、函數呼叫、欄位、陣列與變數的存取。第四節討論各種**敘述**(statement)的語法樹結構與執行方式。第五節介紹位元碼的直譯機制與樹形碼的銜接機制。第六節討論直譯實例與效能評估，最後則是總結。

## 貳、 語法樹結構

### 一、語法樹程式庫的簡介

本系統先透過**剖析器**(parser)把**原始碼**(source code)中各種程式結構轉換成相對應的**樹狀節點**(tree node)，這些樹狀節點所屬的類別各自配置了直譯所需的**虛擬函數**(virtual function)。我們利用**遞迴呼叫**(recursive call)與**動態繫結**(dynamic binding)來執行各個語法單元，各個單元的執行方式將於後續章節詳細討論。

Java 的程式庫以**套件**(package)來分類。本系統在語法樹的程式庫中，以類別 JUnit 為語法樹中各種單元類別的共同祖先類別，如圖 1 所示。它宣告了一些重要的函數讓

構成語法樹的各個子類別實作。語法樹的程式庫依照 Java 原始碼的結構特性區分為**定義類**(套件 oai.tree.def)、**敘述類**(套件 oai.tree.stmt)、**算式類**(套件 oai.tree.exp)與**文數字類**(套件 oai.tree.lit)四個主要套件。

套件 oai.tree.def 中的類別主要是用來定義**類別**(class)、**函數**(function)、**欄位**(field)、**變數**(local variable)與其他執行期所需的類別。

套件 oai.tree.exp 中定義了語法樹中各種算式節點所對應的類別。一個基本的 Java 算式通常是由**運算符**(operator)和**運算元**(operand)所組成。運算符在語法樹中是屬於**枝型節點**(branch node)，而運算元有可能是枝型節點或是**葉節點**(leaf node)。

套件 oai.tree.lit 中定義了 Java 語言中的各種**文數字**(literal)，這些類別在算式計值的遞迴呼叫中扮演著葉節點的角色，負責將自己所代表的常數值回傳給父節點。

套件 oai.tree.stmt 中的類別都是用來定義各種**敘述**(statement)單元，在這些敘述單元的類別中都有一個函數 execute()負責敘述的執行。各個敘述單元類別會依照自己的功能實作函數 execute()。

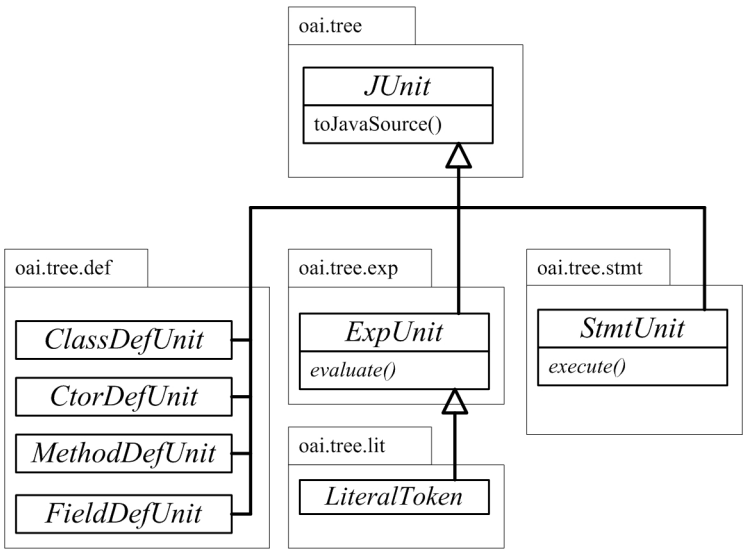



圖 1. 語法樹主要程式庫套件

## 二、類別單元與欄位單元的內容

**類別定義單元**的類別是 TLevClassDefUnit，它的父類別是抽象類別 ClassDefUnit。TLevClassDefUnit 中有若干個欄位各自指著一條陣列，這些陣列分別是用來管理**建構元**(constructor)、欄位、函數、**巢狀類別**(nested class)、**實作介面**(implements interface)與**虛擬函數表**(virtual function table)。虛擬函數表中記錄了該類別實際可呼叫的**實體函數**(non-static method)，並在稍後支援實體函數的執行。此外，TLevClassDefUnit 類別中還有一些欄位用來記載**實體初始化區塊**(non-static initializer)、**靜態初始化區塊**(static initializer)與**靜態欄位值**(static field value)等。

**欄位定義單元**的類別是 TLevFieldDefUnit，它有兩個函數：evaluate(Object obj)與 setValue(Object obj, Object val)，前者用來回傳目前的欄位值，後者則是對欄位進行設值。在類別 TLevFieldDefUnit 中還需要宣告一些欄位用來定義欄位的資料型別、欄位名稱、預設值與欄位的**修飾詞**(modifier)等。靜態欄位值是由 TLevClassDefUnit 負責管理，**實體欄位值**(non-static field value)則是由執行期的實體來管理。

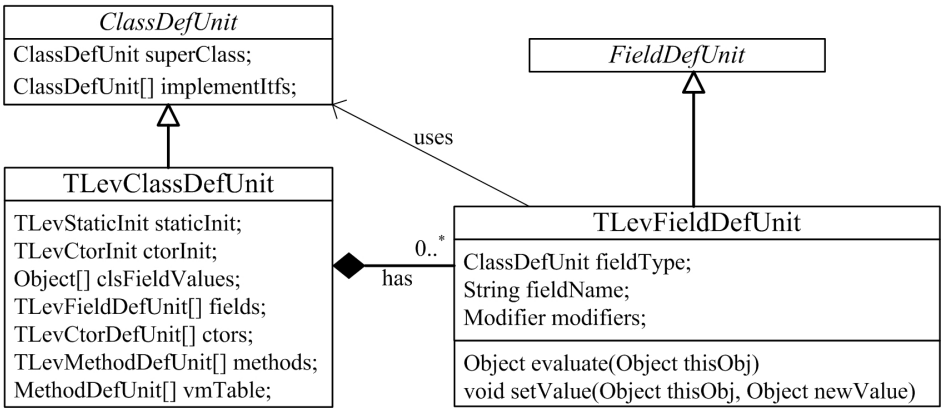



圖 2. 類別與欄位單元的主要結構

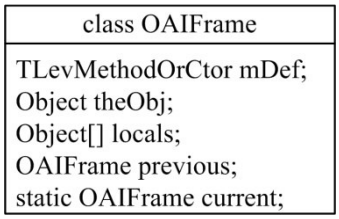
## 三、框架與區域變數

函數或是建構元在被呼叫時都會有一個**框架**(frame)負責管理**參數**(parameter)與**區域變數**(local variable)，在直譯過程中負責管理框架的類別是 OAIFrame，它的結構如圖 3a 所示。在 OAIFrame 中有一個 TLevMethodOrCtor 型別的欄位 mDef，這個欄位用來指著使用這個框架的函數或建構元。Object 型別的欄位 theObj 用來裝實體函數或是建構元的 this 指標，另外還有一個 Object[]陣列 locals 用來記錄區域變數的值。在建立 OAIFrame 時，建構元會依照函數或建構元中的區域變數數量(參數的數量一併計算)將陣列初始化好。

OAIFrame 有一個靜態欄位 current 指著目前使用中的框架，還有一個實體欄位 previous 指著前一個框架。當函數被呼叫時便會建立新的框架，並將 current 的值設到新框架的 previous 中，再把 current 的指標更新成新框架的指標。當函數結束完畢後就把前個框架的指標從 previous 中取回並設回 current 之中。

語法樹中負責管理區域變數的類別是 LocalVarDefUnit，它的結構如圖 3b 所示。在這個類別中的欄位 varType 用來記著這個區域變數的型別、欄位 theName 用來記著這個變數的名稱、欄位 isFinalFlag 用來標示這個變數是否為 final。每個變數都有一個索引編號 varIndex，這個編號指示該變數在 OAIFrame 的 locals 陣列中的存取位置。varIndex 的值是在建立語法樹的過程中，依照變數宣告的順序來指派。

(a)類別 OAIFrame 的結構




(b)類別 LocalVarDefUnit 的結構

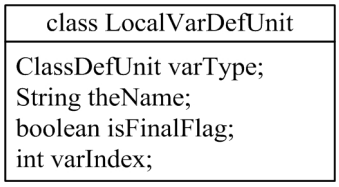



圖 3. 框架與區域變數

## 四、函數單元的內容與執行

**函數定義單元**的類別是 TLevMethodDefUnit，在這個類別中宣告了多個欄位用來記錄函數的修飾詞、**回傳值型別**(return type)、函數名稱、**參數列**(parameter list)與**頂層區塊**(top block)。其中還有一個 int 型別的欄位 vtblIdx 指向所屬類別中的**虛擬函數表**，使得實體函數在執行期時可以完成**動態繫結**。

類別 TLevMethodDefUnit 中有實體函數 exactInvoke(Object theObj, Evaluable[] actualArgs)。當**函數呼叫式**(function call expression)被計值時便會轉呼叫 exactInvoke()。函數在被呼叫後會建立一個框架用來管理參數、區域變數與 this 指標。當函數被呼叫時，便將函數定義與呼叫者的參數帶入 OAIFrame 的建構元，並把參數與區域變數指派給新的框架，然後將框架推入(push)到**框架堆疊** (frame stack)中。此時 exactInvoke()就會呼叫頂層區塊敘述的 execute()函數來執行函數本體內的敘述，最後再將函數值回傳給呼叫者。執行函數本體的細節將在第肆章討論。

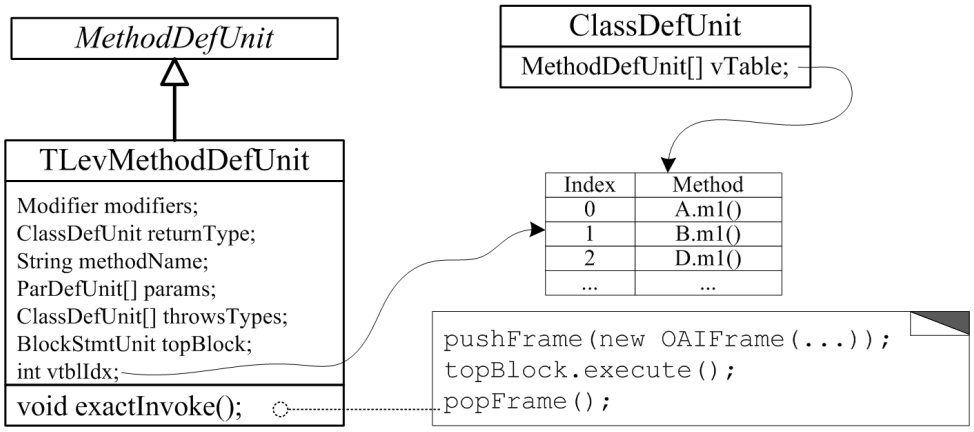



圖 4. 函數單元的主要結構

### 五、建構元單元的內容與執行

在 Java 語言中初始化類別的單元有**靜態初始化區塊**(static initializer)、**實體初始化區塊**(non-static initializer)與**建構元**(constructor)。

類別 TLevStaticInit 負責執行類別的靜態初始化區塊，它有一個頂層區塊。當類別被載入時，直譯器便呼叫 TLevStaticInit 的函數 invoke()來執行頂層區塊，執行流程和函數單元大致相同，只是 TLevStaticInit 沒有參數也沒有回傳值。負責實體初始化區塊的類別是 TLevCtorInit，它會在創建類別實體時自動被呼叫。TLevCtorInit 的內容與執行過程和 TLevStaticInit 相同，差別在於 TLevCtorInit 在執行時會將自己的指標作為參數傳給 OAIFrame。

建構元可以視為是沒有函數值的函數，因此在建構元定義類別 TLevCtorDefUnit 中大部分的欄位和 TLevMethodDefUnit 相同。在創建類別實體時，TLevClassDefUnit 會先造出類別實體，然後再呼叫 TLevCorDefUnit 的 bodyInvoke()執行建構元本體來初始化實體。建構元是由算式單元 new 所發動，我們將在第參章討論。

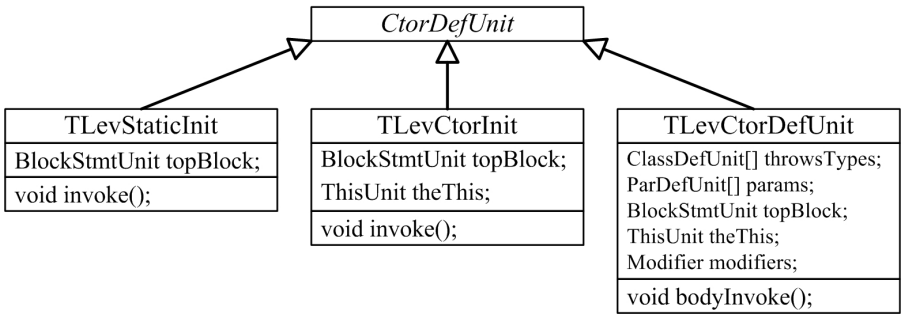



圖 5. 建構元單元的主要結構

## 參、 算式的計值

### 一、計值與設值的 Java 介面

**算式**(expression)的運算單元皆可以被**計值**(evaluate)，有些運算單元還可以被**設值**(assign)。在語法樹中，我們設計了一個 Java 介面 Evaluable 讓那些可以被計值的算式單元類別實作，這個介面中有一個函數 evaluate()，它的回傳值型別是 Object。這個函數是以遞迴的方式對子樹計值，然後將結果回傳給呼叫者。

當一個運算單元可以被設值時，它必定也可以被計值，因此介面 LValue 繼承介面 Evaluable。LValue 中有一個沒有回傳值的函數 setValue(Object value)讓其他算式可以對這個運算單元進行設值。那些可以被設值的運算單元類別(例如：區域變數、陣列元素等)皆實作介面 LValue。

算式中最基本的計值單元為**文數字類別**(literal class)，這些文數字類別代表著 Java 語言的**原始資料型別**(primitive data type)、**字串型別**(string type)與**空型別**(null type)。這些文數字類別皆實作介面 Evaluable，在被計值時就把它所代表的值裝在**外覆類別**(wrapper class)中回傳給呼叫者。因此在算式計值時的遞迴呼叫中扮演著葉節點的角色。

某些運算單元中可能會有記載資料型別的欄位，例如 instanceof 運算符。因此我們還設計了一個 Java 介面 Declarable 讓那些具有宣告能力的類別定義單元實作，這個介面中有一系列的函數用來支援執行期時的型別判定與轉換。

## 二、運算符的計值

### (一)一元運算符

Java 語言中的**一元運算符**(unary operator)有 “+”、 “-”、 “!”與 “~”，這些運算符在語法樹中各有相對應的類別，這些類別中都有一個 Evaluable 型別的欄位 op 儲存**運算元**(operand)。以類別 NegUnit 為例，它代表了變號運算符 “-”。在計值時它會先對欄位 op 進行計值，然後對結果值進行負運算後回傳給呼叫者。

運算符 “++”、 “--” 還分為**前置運算符**(prefix operator)與**後置運算符**(postfix operator)，它們在計值後會改變運算元的值，因此它們的類別中會有一個 LValue 型別的欄位 theObj 用來代表運算元。以前置加法運算符 “++op”為例，它在被計值時先將運算元 op 的值加 1，再把 op 的計值結果回傳。若是後置加法運算符 “op++”，則是先將 op 的計值結果存放在臨時變數中再對 op 加 1，最後再將存放在臨時變數中的值回傳給呼叫者。

### (二)二元運算符

在二元運算符單元的類別中有兩個 Evaluable 欄位 op1 與 op2 用來代表運算符的兩個運算元，Java 的二元運算符有 “+”、 “-”、 “*”、 “/”、 “%”、 “>”、 “>=”、 “<”、 “<=”、 “==”、 “!= ”、 “<<”、 “>>”、 “>>>”、 “&”、 “|”、 “^”、 “&&”、 “||” 與 “instanceof”。以乘運算符類別 MulUnit 為例，它在計值時分別先對欄位 op1 與 op2 計值，再將兩個結果作運算，然後將所得的值回傳給呼叫者。

另外還有一些負責設值的二元運算符，包括 “=”、 “+=”、 “-=”、 “/=”、 “%=”、 “*=”、 “<<=”、 “>>”、 “>>>=” ”、 “&=”、 “|=” 與 “^=”。在它們所對應的類別中會有一個 LValue 型別的欄位 theObj 代表等號左邊被設值的那一方，另外一個 Evaluable 型別的欄位 rhs 則代表等號右邊被計值的那一方。當運算符被計值時，它會將 rhs 欄位的計值結果經對應的運算後設值給 theObj。以運算符 “*=” 所對應的類別 MulByUnit 為例，它會將 rhs 的計值結果與 theObj 的計值結果相乘，然後把相乘後的結果設值給 theObj 並回傳給呼叫者。

關鍵字 instanceof 也是二元運算符，在類別 InstanceOfUnit 中有一個 Evaluable 型別的欄位 exp 記著所要判斷的算式，它還有一個 Declarable 型別的欄位 type 記著所判定型別。當 InstanceOfUnit 被計值時，會把欄位 exp 計值後結果的型別資訊和 type 欄位作比較，並將"是否相符"以布林值回傳給呼叫者。

(三)三元運算符

Java 語言中還有一個三元運算符 "?:"，它在語法樹中所對應的類別為 ConditionalExpUnit，它有三個 Evaluable 型別的欄位 op、tval 與 fval。在計值時，若 op 的計值結果為"true"則回傳 tval 欄位的計值結果，"否"就回傳 fval 欄位的計值結果。

(四)關鍵字 this 與 super 的計值

關鍵字 this 與 super 在語法樹中所對應的類別為 ThisUnit 與 SuperUnit，當 ThisUnit 被計值時，它先取得目前的框架(OAIFrame)，並將框架中所記載的 this 指標回傳給呼叫者。另外當 SuperUnit 被計值時，也是從目前的框架中取回 this 指標，再透過 this 指標查找父類別來回傳父類別的指標。

## 三、欄位的存取

在算式中定義**靜態欄位**(static field)與**實體欄位**(non-static field)的類別分別是由 DotStaticFieldUnit 類別與 DotThisFieldUnit 類別來記載。這兩個類別皆繼承了 DotFieldUnit 類別，而 DotFieldUnit 也實作了介面 LValue，因此具有計值與設值的特性。

DotFieldUnit 中有一個 FieldDefUnit 欄位 fDef 指著將要存取的欄位定義，它的子類別 DotThisFieldUnit 擁有一個 Evaluable 型別的欄位指著發動它的指標(例如區域變數)；而 DotStaticFieldUnit 則是擁有一個 ClassDefUnit 型別的欄位指著存取欄位所屬的類別。

**靜態欄位**(static field, 又稱 class variable)的欄位值是由宣告它們的類別所管理，因此當靜態欄位被計值時，DotStaticFieldUnit 會從欄位定義 fDef 取回欄位值。當欄位被設值時，DotStaticFieldUnit 便將新值透過欄位定義 fDef 設回給宣告類別。

**實體欄位**(non-static field, 又稱 instance variable)的欄位值是由類別 UnivTLevObj 管理，在 UnivTLevObj 中有一個陣列 fieldValues 記錄了各個實體欄位的值。當實體欄位被計值時，DotThisFieldUnit 先對發動者計值，在取得 UnivTLevObj 實體後，依照欄位定義 fDef 中的索引值從 UnivTLevObj 的 fieldVales 陣列中取回欄位值。同理，若對實體欄位設值，就是依照 fDef 的索引值把新值設回 UnivTLevObj 的 fieldValues 陣列中。

## 四、參數、區域變數與陣列的存取

定義**參數**與**區域變數**的類別分別為 ParDefUnit 與 LocalVarDefUnit，其中 ParDefUnit 繼

承了 LocalVarDefUnit。類別 LocalVarDefUnit 都有一個 int 型別的欄位 varIndex 表示這個變數在 OAIFrame 區域變數陣列中的某個位置，欄位 varIndex 的值是在編譯期依照變數宣告的順序來指派(從參數先開始)。

LocalVarDefUnit 實作了介面 LValue，因此它覆寫了計值函數 evaluate()與設值函數 setValue()。在對變數進行計值時，LocalVarDefUnit 依照自己的 varIndex 從 OAIFrame 中的 locals 陣列取值。同理，在對變數設值時，也是將新值設入陣列 locals 中 varIndex 的位置。

陣列的存取是由類別 EntryUnit 來表示(圖 6a)，這個類別有兩個欄位。第一個欄位 arrayEV 是 Evaluable 型別，它用來指著**陣列指標算式**(array reference expression)；第二個欄位 indexEV 也是 Evaluable 型別，這個欄位用來指代表**索引算式**(index expression)。在建立語法樹的過程中，若 arrayEV 所指的實體不是陣列，或者 indexEV 計值後的資料型別不是 int，都將發生例外。在處理多維陣列時，欄位 arrayEV 會指向下一個維度的 EntryUnit，以此類推，如圖 6b 所示。

在存取陣列時，EntryUnit 會先對 arrayEV 計值取回陣列物件。接著再對 indexEV 計值並檢查 indexEV 的索引值是否在範圍之內，如果索引值小於零或是大於陣列長度就會發生 java.lang.ArrayIndexOutOfBoundException。一旦取得陣列物件與索引值後，就透過 java.lang.reflect.Array 類別的 get()與 set()函數來存取陣列。

(a) EntryUnit 結構　　(b) 二維陣列的結構

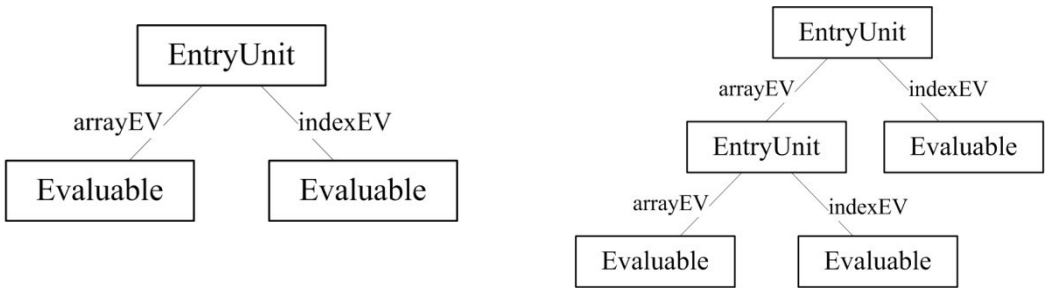



圖 6. 語法樹中的陣列結構

## 五、函數呼叫與計值

在語法樹中用來表示函數呼叫的算式為 CallUnit 類別，這個類別有兩個欄位，分別是 DotMethodUnit 型別的欄位 function 與 Object[]型別的 actualUnits。function 欄位用來指著所要呼叫的函數(可能是靜態函數或是實體函數)，而 actualUnits 則是呼叫函數所需參數陣列。

類別 DotMethodUnit 用來表示函數呼叫算式中不含參數的部分，它有一個 MethodDefUnit 型別的欄位 theMethod 指著所要呼叫的函數。函數呼叫還要區分**靜態函數**(static function，又稱 class method)的呼叫與**實體函數**(non-static function，又稱 instance method)的呼叫，因此 DotMethodUnit 的子類別 DotStaticMethodUnit 與類別 DotThisMethodUnit 分別表示靜態函數呼叫與實體函數呼叫。DotStaticMethodUnit 中有一個 ClassDefUnit 型別的欄位 theClass 指著靜態函數的所屬的類別。DotThisMethodUnit 則是有一個 Evaluble 型別的欄位 invoker 指著函數呼叫的發動者，這個發動者可能是變數、欄位、函數值等。

圖 7 是一個實體函數呼叫 var.setXY(3, 4)的語法樹實例。類別 CallUnit 的欄位 function 指著實體函數呼叫的算式，另一個欄位 actualUnits 指著函數呼叫的參數陣列。參數陣列中有兩個 IntLiteral 型別的元素就是呼叫函數的參數"3"與"4"。函數 setXY()是一個實體函數，因此 CallUnit 的欄位 function 所指的算式單元為 DotThisMethodUnit，這個單元的欄位 invoker 指著這個函數的發動者(也就是變數 var)，另外一個欄位 theMethod 指著實體函數 setXY()的定義。

當 CallUnit 被計值時，它會將 actualUnits 作為參數轉呼叫 function 欄位的函數 invoke()，再把 invoke()的函數值作為自己的函數值回傳給呼叫者。

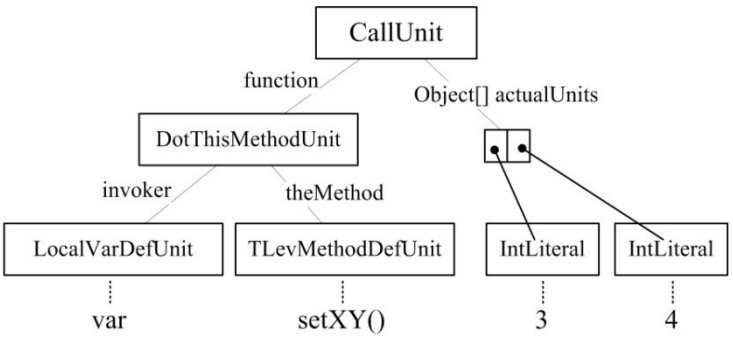



圖 7. 函數呼叫的語法樹實例

## 六、類別實體與陣列的創建

語法樹中用來表示關鍵字 new 的類別為 NewUnit，它有一個型別為 Declarable 的欄位 theClass 與一個型別為 Evaluable 的陣列 actualUnits。欄位 theClass 是這個實體的資料型別，另外一個欄位 actualUnits 則是要傳入建構元的參數。

當 NewUnit 被計值時，首先判斷 theClass 是否有建構元，若沒有預設建構元就轉呼叫 TLevClassDefUnit 的函數 newInstance()。如 theClass 有建構元，則依照 actualUnits 的參數規格搜尋合適的建構元，然後將 actualUnits 作為參數帶入 TLevCtorDefUnit 的函數 newInstance()來創建類別實體。

類別 UnivTLevObj 用來管理執行期的**類別實體**(instance of class)，在這個類別中有一個 TLevClassDefUnit 型別的欄位 cls 指這個實體的型別定義(如圖 8a 所示)。此外，Object 型別的欄位 superObj 用來存放這個實體的父類別指標，欄位 fieldValues 則是存放著各個實體欄位值供 TLevFieldDefUnit 存取。

執行期的陣列物件由類別 UnivTLevArray 來管理，在這個類別中有一個 TLevClassDefUnit 型別的欄位 cls 指著這個陣列的型別(如圖 8b 所示)，另外還有一個 Object 型別的欄位 arrayObj 用來指著由 NewArrayUnit 或是 ArrayValueInitUnit 所創建的陣列指標。

陣列的創建分為無初始值與有初始值兩種狀況，無初始值的陣列創建是由類別 NewArrUnit 負責。在 NewArrUnit 中有欄位 fullType 指示陣列的型別(例如：int[][])，另外還有一個 Evaluable[]型別的欄位 size 記錄各個維度的陣列長度。當 NewArrUnit 被計值時，它便依照 fullType 的型別資訊建立維度為 size.length 的陣列，並賦予原始資料型別的陣列元素預設值(例如：元素若為 int 型別則預設值為 0)。有初始值的陣列創建是由類別 ArrayValueInitUnit 負責，它也有一個欄位 fullType 指示陣列的型別，此外還有 Evaluable[]型別的欄位 val 用來儲存初始值。以 “int[] ia={1, 2} 為例，ArrayValueInitUnit 在計值時會先建立一個 int[]型別的陣列實體，然後再將 vals 的計值結果依次設到陣列的各個元素中。

(a) UnivTLevObj 類別結構

| class UnivTLevObj |
| --- |
| TLevClassDefUnit cls;<br>Object superObj;<br>Object[] fieldValues; |

(b) UnivTLevArray 結構

| class UnivTLevArray |
| --- |
| TLevClassDefUnit cls;<br>Object arrayObj; |

圖 8. 執行期物件管理類別

## 肆、 敘述的執行

Java 語言的敘述分為**簡單敘述**(simple statement)與**複合敘述**(compound statement)，簡單敘述沒有子敘述，例如：**算式敘述**(expression statement)、**return 敘述**、**break 敘述**、**continue 敘述**、**assert 敘述**與 **throw 敘述**等。複合敘述通常會包含一個以上的子敘述，例如：**區塊敘述**(block statement)、**if-else 敘述**、**switch 敘述**、**try-catch 敘述**、**while 敘述**、**for 敘述**與 **synchronized 敘述**等。

我們將一切敘述都用**抽象**(abstract)的祖類別 StmtUnit 來管理。StmtUnit 有一個抽象函數 execute()讓各個敘述的子類覆寫，另外還有一個 String 型別的欄位 label 用來記錄敘述的**標籤**(label)。

以下依直譯方式分項加以討論：

### 一、簡單敘述

**算式敘述**(expression statement)是一種簡單敘述，內含一個 Evaluable 型別的欄位 exp。當算式敘述被執行時，它所**覆寫**(override)的函數 execute()將會呼叫 exp 欄位的函數 evaluate()來對算式進行計值。

### 二、區塊敘述與區域變數宣告敘述

**區塊敘述**(block statement)常被用作其他敘述的子敘述，它自己還內含有許多的子敘述。區塊敘述的 execute()會依序地呼叫執行本體內的各條敘述的 execute()。

"**區域變數宣告敘述**"內含於區塊敘述，它是由兩大部分組成：(1)**變數定義**(local variable definition)，這包含了變數所宣告的資料型別與名稱，另外還有一個可有可無的(2)**初始值**(initial value)算式，這個初始值算式會在敘述被執行時計值，再把計值結果設入變數。

在建構語法樹的過程中，區塊敘述會為區域變數宣告敘述中的變數編派它們在 OAIFrame 中的索引值。這些索引值從**頂層區塊**(top block)開始，每增加一條變數宣告敘述，索引值便會遞增。最後將統計在建構元或是函數中的變數數量，以便於在 OAIFrame 中建立足夠長度的陣列來儲存變數。

### 三、流程控制類敘述

(一)if-else 敘述

**if 敘述**(if statement)一共有三個部分所構成：條件算式、條件為 true 時的敘述與條

件為 false 時的敘述。這三個部份分別由 Evaluable 型別的欄位 exp、StmtUnit 型別的 tPart 與 fPart 欄位所組成。If 敘述的 execute()首先呼叫 exp 欄位的 evaluate()。evaluate()的函數值若為 true 則呼叫 tPart 欄位的 execute()，否則就呼叫 fPart 欄位的 execute()(若 fPart 為空，則忽略不執行)。

(二)switch 敘述

**switch 敘述**(switch statement)是複合敘述，它在語法樹中的類別為 SwitchStmtUnit。SwitchStmtUnit 繼承了 BlockStmtUnit，在這個類別中有一個 Evaluable 型別的欄位 var 用來記載判斷條件的算式、一條 CaseStmtUnit 型別的**串列**(list)指著若干條 **case 敘述**(case-statement)與一個 CaseStmtUnit 型別的欄位 defaultCase 指向預設的 case 敘述。

我們將 case 敘述視為一種特殊的標籤，這些標籤會安插在 switch 敘述本體中用來指示 case 敘述的起點。switch 敘述的 execute()會把 var 的計值結果拿來和 CaseStmtUnit **串列**(list)中的值相比較，若執行比較結果落在某條 case 中時，則 switch 敘述將從此條 case 所的下一條敘述開始執行。如果沒有找到合適的 case 條件，但有 default 標籤就直接執行 default 標籤之後的敘述，否則結束執行。case 敘述中通常會含有 break 敘述，所以 switch 敘述還必須負責捕捉 break 敘述所丟出的例外。(break 敘述在下節討論)。

(三)while 敘述與 for 敘述

**while 敘述**(while statement)與 **do-while 敘述**(do-while statement)是由迴圈執行條件與敘述本體所構成，定義它們的類別分別是 WhileStmtUnit 與 DoStmtUnit。這兩個類別各自都擁有一個記錄執行條件的算式欄位 exp，還會有一個記錄敘述本體的欄位 body。while 敘述的 execute()首先會對欄位 exp 計值，若計值結果為“true”就呼叫 body 的 execute()，之後再對 exp 計值，若是“false”就會結束執行，否則將繼續重複這個循環；而 do-while 敘述的 execute()則是先執行 body 所指的敘述，再判斷 exp 的計值結果，若結果為“true”就執行 body，執行完後會再判斷 exp 的計值結果，若為 false 就結束離開否則就重複這個循環。

**for 敘述**(for statement)是由“迴圈初始敘述”、“終止條件”、“更新敘述”與“敘述本體”所構成，初始條件還可以是變數宣告敘述或是多條個算式敘述，更新敘述同樣也可能包含多條算式敘述。在類別 ForStmtUnit 中，欄位 initStmts 是一條敘述型別的陣列，它可能指著多條用來初始化迴圈條件的敘述。欄位 exp 記錄了執行終止條件、欄位 updates 也是一條算式敘述型別的陣列作為更新敘述，最後還有一個欄位 body 指著敘述本體。

當 for 敘述被執行時，首先依次地執行 initStmts 陣列中所指住的每一條敘述，接著對 exp 欄位計值，若計值結果為“true”就執行 body 欄位所指的敘述，執行完畢之後再依次執行 updates 中的算式敘述，然後對 exp 欄位計值，判斷計值結果決定重複循環或是結束。

while, do-while, for 的 body 經常會是一個區塊敘述，這時 body 的 execute()會再向下依序執行區塊內的各個敘述(如前面之討論)。

(四)try-catch-finally 敘述

**try-catch 敘述**(try-catch statement)是由一個 try 區塊與若干個 **catch 敘述**(catch statement)所組成，最後還可以有一個可有可無的 finally 區塊。定義 try-catch 敘述的類別是 TryStmtUnit，它有三個欄位分別為 tryBlk、catchBlks 陣列與 finallyBlk(如圖 9 所示)。其中 tryBlk 與 finallyBlk 的型別為 BlockStmtUnit，而 catchBlks 為 CatchStmtUnit 型別的陣列。CatchBlockStmtUnit 兩個欄位 xptVar 與 blk，它們的型別分別是 LocalVarStmtUnit 與 BlockStmtUnit。xptVar 負責記載 catch 敘述中的變數宣告，另外的 blk 是 catch 敘述的本體。

try-catch 敘述的 execute()首先會呼叫 tryBlk 的 execute()。在 tryBlk 的執行過程中，可能發生**例外**(exception)。為了在語法樹中模擬例外機制，我們將 **throw 敘述**(throw-statement)所產生的例外包裝在我們自訂的例外類別 TLevThrowXpt 中。若在執行 tryBlk 的過程中發生例外，try-catch 敘述的 execute()將只負責捕捉 TLevThrowXpt 的例外，好讓其它例外(uncaught exception)上轉到父敘述中。

若 try-catch 敘述中有 finally 敘述，則不論 tryBlk 是否發生例外，try-catch 敘述最後都將執行 finallyBlk 欄位所指的敘述。

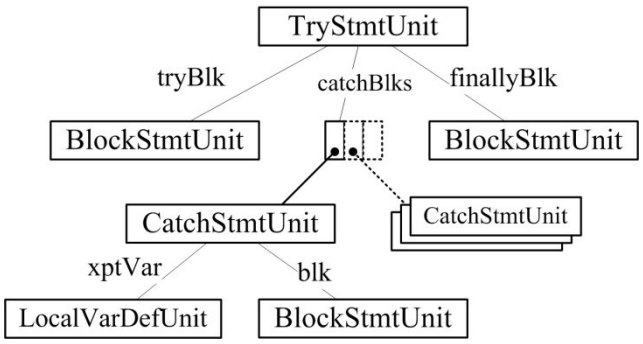



圖 9. try-catch 敘述在語法樹的結構

**四、脫離類敘述**

(一)return 敘述

如前所述，我們是以從上而下(top-down)的遞迴呼叫來完成語法樹的直譯機制，這個呼叫的過程自然會**巡行**(traverse)整個語法樹的資料結構。

**return 敘述**(return statement)可能在函數末端，但也可能在中間。當 return 敘述單元被執行時，必須終止整個函數的直譯並回到函數呼叫點。

為了讓控制流程脫離一層層執行中的 execute()，我們利用**例外機制**(exception mechanism)讓遞迴呼叫得以從深層自動脫離。

當 return 敘述單元類別的函數 execute()被執行時，它會產生一個 ReturnXpt 型別的例外，這個例外會中斷函數本體中內各層 execute()的執行，並在 TLevMethodDefUnit 的 exactInvoke()中被**捕捉**(catch)。由於 return 敘述可能擁有**回傳值**(return value)。因此在 return 敘述單元中還擁有一個 Evaluable 型的欄位 retValue 負責記載回傳值的算式。當 return 敘述單元被執行時，是將 retValue 的計值結果存入 ReturnXpt 中，這使得 exactInvoke() 能夠從 ReturnXpt 例外的實體中取得函數的回傳值。

(二)break 敘述與 continue 敘述

**break 敘述**(break statement)與 **continue 敘述**(continue statement)的執行同樣也是利用例外機制來進行脫離，當 break 敘述或是 continue 敘述被執行時，它們分別會產生 BreakXpt 與 ContinueXpt 例外讓外層敘述捕捉並處理。

迴圈類敘述在執行時會捕捉執行敘述時所可能產生的 BreakXpt 與 ContinueXpt。若是捕捉到 BreakXpt 就要比較 BreakXpt 所攜帶的標籤是否和本敘述的標籤相等，一旦相等就直接結束迴圈的執行，否則再將這個例外扔給外層敘述繼續處理。當捕捉到 ContinueXpt 時，同樣也會將 ContinueXpt 所攜帶的標籤與本敘述做比較。若標籤相等，while 敘述就會重新執行迴圈，而 for 敘述會先執行 updates 中的算式敘述再重新開始。如果兩標籤不相等，就再將此例外再扔給外層敘述。

(三)throw 敘述

**throw 敘述**(throw statement)有一個 Evaluable 型別的欄位 xptValue 負責記載例外值的算式，throw 敘述的 execute()是將 xptValue 的計值結果--也就是**真實例外實體**(actual exception instance)—作為參數代入 TLevThrowXpt 的建構元。當 try-catch 敘述攔截到 TLevThrowXpt 型別的例外時，便將 TLevThrowXpt 所攜帶的真實例外實體解開，然後和 catchBlks 中的敘述比對，如果 catchBlks 中有處理該例外的敘述時，便把真實例外實體作為參數傳給 catch 敘述執行。否則就讓 TLevThrowXpt 繼續向外層傳播出去。

### 五、同步敘述

**同步敘述**(synchronized statement)也屬於複合敘述，它有兩個部分：第一個部分是請求**鎖定**(lock)的物件(例如：區域變數或是欄位)，第二個部分就是區塊敘述。因此在同步敘述的類別中有一個 Evaluable 型別的欄位 obj 將指著一條算式，當敘述被執行時，會先對 obj 欄位計值以取得鎖定實體，一旦取得鎖定後就會執行區塊敘述，並在執行區塊敘述的 finally 敘述中解除鎖定(依據 Java 語言規格書的規定)(Gosling, Joy, Steele Jr., and Bracha, 2005)。

## 伍、 位元碼的直譯與樹形碼的銜接

Java 的**執行環境**(runtime environment)中有大量已經編譯成**位元組碼**(bytecode)的程式庫，我們將這些位元組碼編組為 **B 級**(byte-code level)子樹並透過 Java 的 Reflection 技術(Arnold, Gosling, and Holmes, 2005)來執行。原有語法樹的部份則稱為 **T 級**(tree-code level)子樹。

### 一、位元碼的直譯機制

(一)B 級類別的使用

直譯器所讀入的 Java 原始碼可能會用到其他已經編譯好的程式庫，這些編譯過後的類別是由類別 BLevClassDefUnit 來管理。在類別 BLevClassDefUnit 中有一個 java.lang.Class 型別的欄位 theClass 用來指著某個 B 級類別。透過 Java Reflection 所提供的 API(Application Programming Interface)，我們將可以取得這個類別的完整資訊。Java 的**原始資料型別**(primitive data type)在語法樹中是由 BLevClassDefUnit 管理，並在直譯器啟動時就準備好。

在創建類別實體時，類別 NewUnit 便會把參數帶入 BLevClassDefUnit 的函數 newInstance()中，這個函數接著轉呼叫 theClass 的 newInstance()來創建 B 級類別實體，並將創建好的實體回傳給呼叫者。

(二)B 級欄位的存取

負責定義 B 級**欄位**(field)的類別是 BLevFieldDefUnit，在這個類別中有一個 java.lang.reflect.Field 型別的欄位 theField 負責指著某個 B 級欄位，我們透過 theField 便可取得 B 級欄位的資訊，例如：修飾詞、資料型別、欄位名稱等。BLevFieldDefUnit 還宣告了 evaluate(Object obj)與 setValue(Object obj, Object value)函數以便提供 B 級欄位的取值與設值，這兩個函數分別會轉呼叫 theField 的 get(Object obj)與 set(Object obj, Object value)進行取值與設值。

(三)B 級函數的呼叫

類別 BLevMethodDefUnit 用來定義 B 級函數，它有一個 java.lang.reflect.Method 型別的欄位 theMethod，我們藉由 theMethod 來取得 B 級函數的相關資訊並用來呼叫 B 級函數。

在語法樹中，B 級函數的發動也是由 CallUnit 負責。當 CallUnit 被計值時，便會轉呼叫 MethodDefUnit 的函數 invoke()，然後透過動態繫結的機制轉呼叫 BLevClassDefUnit 的函數 exactInvoke(Object invoker, Object[] actual)。參數 invoker 為實體函數的發動者，若是靜態函數則參數值為 null；另外參數 actual 為實際函數呼叫的參數陣列。接著轉呼叫 theMethod 的函數 invoke(Object obj, Object[] args)來執行 B 級函數，最後再將函數值傳回給呼叫者。

## 二、T 級類別繼承 B 級類別時的銜接

當 T 級類別**繼承**(inherit)B 級類別時，我們必須在兩者之間建立新的溝通機制使 T 級與 B 級之間的虛擬機制得以正常運作。以圖 10a 的 T 級類別 SubClassT 為例，它繼承了 B 級類別 SuperClassB。其中 SuperClassB 宣告了函數 m1()與 m2()，而 SubClassT 覆寫了 m1()。假設 SuperClassB 的函數 m2()將呼叫函數 m1()，則當 SubClassT 的實體 objSubT 呼叫 m2()時，依動態繫結的原則應該要呼叫 SubClassT 的 m1()。為了使動態繫結的機制可以穿越 T 級與 B 級類別，本系統的編譯器會自動產生**橋接類別**(bridge class)的位元組碼，再利用橋接類別來銜接 T 級與 B 級。

銜接的情況有兩種：(1)B 級類別叫用了 T 級類別的實體函數與(2)T 級類別叫用 B 級類別的實體函數。以 SuperClassB 的實體函數 m1()來說明，第一種情況的作法是在橋接類別中覆寫實體函數 m1()，當橋接類別中的 m1()被呼叫時，它將透過欄位 subCls(這個欄位會指著 T 級的子類別，請參考圖 10b 負責直譯 SubClassT 中的函數 m1()。

第二種情況的作法是在橋接類別中宣告“上轉函數”負責轉呼叫父類別的實體函數。以類別 SuperClassB 的 m1()為例，本系統的編譯器將在橋接類別中宣告相同規格的函數，但須將函數名稱將改名為“_super_m1()”，這是為了避免函數名稱衝突。當 T 級類別透過關鍵字 super 呼叫 B 級父類別的函數時，直譯器會透過欄位 superCls(這個欄位會指著橋接類別)轉呼叫上轉函數。

(a) T 級類別繼承 B 級類別示意圖。　　(b) T 級類與 B 級類別銜接示意圖。

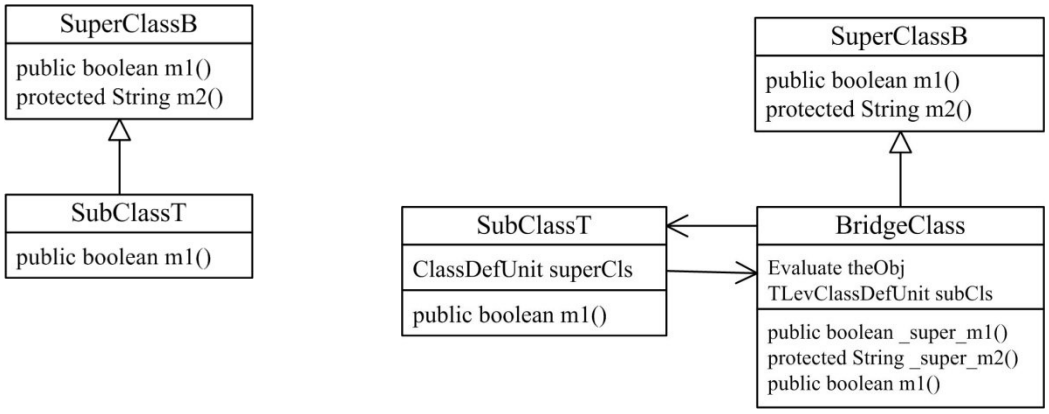



圖 10. 樹形碼與位元碼銜接示意圖

## 陸、 直譯實例與效能評估

### 一、交談式直譯實例

我們以改進後的 OAI (Open Access Interface) (廖賀田、陶彥樺，2001)作為操作介面，使用者可以在本系統的操作介面中直接輸入 Java 敘述，OAI 會將 Java 原始碼轉換成語法樹讓使用者依據需求進行直譯或是除錯。OAI 的操作介面中含有兩種模式：一種是**命令模式**(command mode)，此模式在視窗中的提示符號為 "#"。另外一種模式為**敘述模式**(statement mode)，使用者可以直接在視窗中輸入敘述進行直譯。

表 1 的範例是一個小型的繪圖程式，這個程式可以在視窗上利用滑鼠拖曳來繪製圖形。表 2 是 OAI 主控視窗在操作過程中的畫面，首先我們利用#load 指令將 Draw.java 讀入直譯器成為 T 級類別，接著在敘述模式中製造一個視窗並以類別 Draw 的實體作為視窗的 Content Pane (Walrath, Campione, Huml, and Zakhour, 2004)，交談過程列於表 2，執行時用滑鼠在視窗中劃線如圖 11。在操作過程中，滑鼠事件被轉接到 T 級類別中的反應函數(response function)，但使用上仍然順暢自然。

表 1. Draw.java

```
import java.awt.event.MouseEvent;
import tw.fc.gui.GraphicPanel;
pubilc class Draw extends GraphicPanel {
    int lastX, lastY;
    public void mousePressed(MouseEvent e) {    // response function
        lastX=e.getX(); lastY=e.getY();
    }
    pubilc void mouseDragged(MouseEvent e) { // response function
        final java.awt.Graphics g = this.getGraphics();
        final int x = e.getX(), y = e.getY();
        g.drawLine(lastX, lastY, x, y);
        lastX = x; lastY = y;
    }
}
```

表 2. OAI 主控視窗畫面

```
* :)# load Draw.java
* :)# import javax.swing.JFrame
* :) JFrame f = new JFrame("Draw");
_res -> (java.lang.Object) <null>
* :) f.setBounds(0, 0, 300, 200);
_res -> (void) <null>
* :) f.setContentPane(new Draw());
_res -> (void) <null>
* :) f.setVisible(true);
_res -> (void) <null>
```

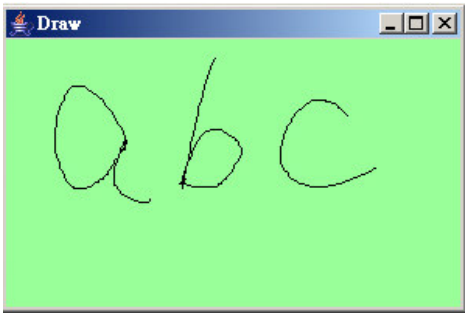



圖 11. 執行畫面

## 二、直譯整個程式

我們以一套 “互動式 3D 導覽系統” (向賢偉，2008)進行測試。使用者可以經由這套 3D 導覽系統與系統所展示的環境進行互動，就如同親臨現場遊走參觀。這套導覽系統在“第十二屆全國大專院校資訊管理實務競賽”中獲得 AP5 資訊應用組第一名。

這套系統有 110 個類別共計約一萬行的原始碼，其中包含了複雜的繪圖計算。本論文的直譯器在讀入原始碼後能夠正確執行該系統(如圖 12 所示)，執行時的畫面與操作也都很流暢。

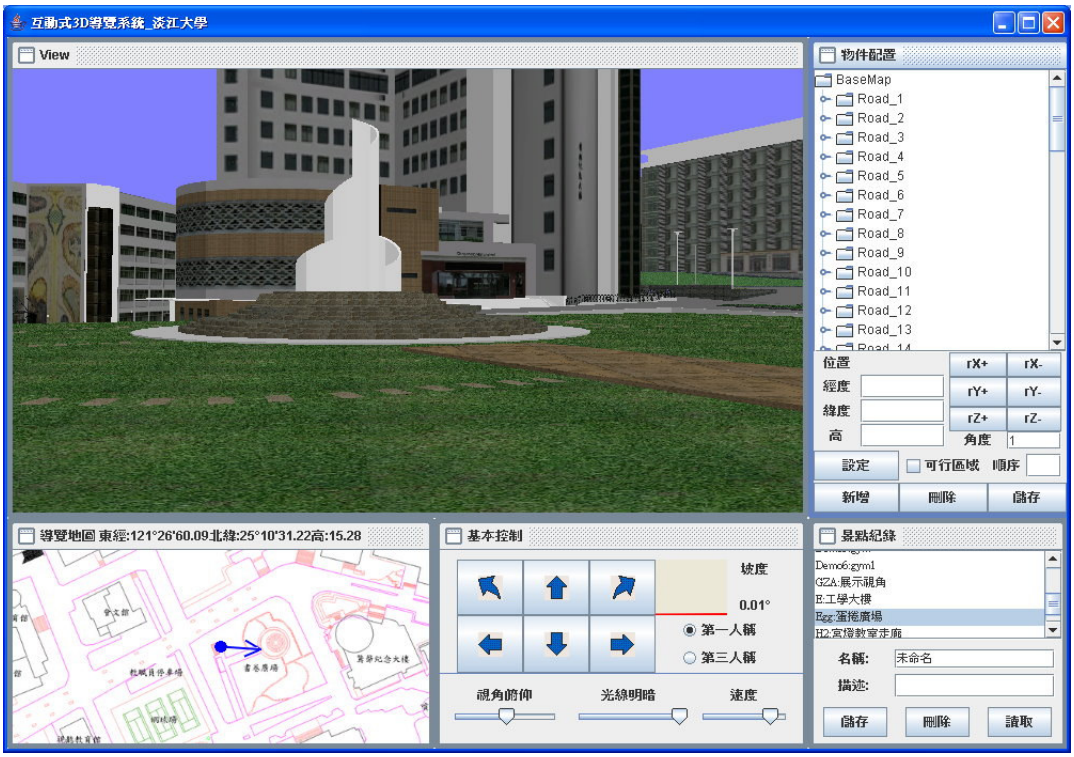



圖 12. 互動式 3D 導覽系統執行畫面

## 三、效能評估

本節將針對樹形直譯機制與傳統編譯後執行進行效能的的比較。測試程式碼將會測試空迴圈、靜態函數呼叫、實體函數呼叫、算式設值、一元算式計值、多層判斷敘述與陣列存取。

實驗是在 Intel Core 2 Duo T7100 1.8 GHz 配有 2GB 主記憶體的電腦上進行，作業系統為 Microsoft Windows XP Professional (Microsoft Corporation, 2008)。Java 的執行環境為 j2se 6.0 (Sun Microsystems, Inc., 2007)。

表 3 的測試結果顯示本論文的直譯器在執行時的效能不及傳統編譯機制，這是因為本論文的直譯器是透過 Java 的虛擬機器(Lindholn, and Yellin, 1999)來執行，在效能上會比編譯後執行慢。未來若將本系統以 C++(Stroustrup, 2000)來實作，效能應該會和編譯後執行不相上下。

表 3. 測試項目的原始碼與結果

| 測試項目 | 測試項目原始碼 | 執行時間 (秒) | 執行時間 T 級：B 級 |
|---|---|---|---|
| 空迴圈 | int count = 50,000,000;<br>while (count-- > 0) {<br>;<br>} | 3.8 秒 | 7：1 |
| 靜態方法呼叫 | int count = 5,000,000;<br>while (count-- > 0) {<br>staticCall( ); //函數本體為空<br>} | 3.2 秒 | 18：1 |
| 非靜態方法呼叫 | int count = 5,000,000;<br>while (count-- > 0) {<br>nonStaticCall( ); //函數本體為空<br>} | 3.3 秒 | 26：1 |
| 設值算式 | int count = 5,000,000;<br>while (count > 0) {<br>count = count - 1;<br>} | 1.2 秒 | 20：1 |
| 一元算式 | int count = 5,000,000;<br>while (count > 0) {<br>count--;<br>} | 1.2 秒 | 15：1 |
| 多層判斷式 | int x=0;<br>int count = 5,000,000;<br>while(count-- >0){<br>if(x > 0){<br>if(x > 1){<br>if(x > 2){ x = 0; }<br>else{ x++; }<br>}<br>else{ x++; }<br>}else{ x++; }<br>} | 2.3 秒 | 13：1 |

| | | | |
|---|---|---|---|
| 陣列存取 | int[ ] num = new int[1];<br>int count = 5,000,000;<br>while(count-- >0){<br>num[0]=count;<br>} | 4.2 秒 | 30：1 |

## 四、遞迴式直譯的極限

本系統是利用 Java 語言製成的語法樹直譯器，在執行時是對語法樹進行由上而下的遞迴呼叫，因此有必要探究遞迴呼叫的極限能力。

就語法上來說，Java 語言並沒有限制遞迴呼叫的層數。但由於參數傳遞、呼叫與回返的資料結構會耗用主記憶體，太過多層的呼叫仍會造成記譯體耗盡的狀況。

考慮以下的遞迴函數：

```
long sum(long n) {
    if (n == 0) { return 0; }
    return (sum(n - 1) + n);
}
```

以這個函數在我們的平台上作測試，發現經由 Java 虛擬機可以順利執行的最大參數值是 n=7134。

我們的直譯器目前仍是經由 Java 虛擬機運作。當然也會受到這種執行期的限制。以連續做加法的式子來進行測試，我們可以正常執行 1+2+3+4+5+....+4673+4674。若再加上一項(到 4675)就會耗盡記憶體。這個式子的語法樹含有 4674 個**終端節點**(terminal node)和 4673 個**非終端節點**(non-terminal node)。由連續加法的**左結合性**(left associative)規則，這是一個向右傾斜的樹。它在直譯時一共要進行 9347 個遞迴呼叫。由於終端節點的計值是立刻完成，真正等著執行的遞迴函數最高是堆到 4674 層。

相對於上一個用來測試的遞迴函數可堆到 7134 層。連加式的語法樹可堆的層數少到只剩一半左右。這是因為我們的直譯器被載入(load)後已經耗用了一些主記憶體。

就實務上來說，程式的行數增加時，語法樹通常只是就寬度在增長。深度超過 100 的語法樹已經是極為罕見的不自然情況。

## 柒、 結論

本論文提出以 Java **語法樹**(syntax tree)作為**中間語言**(intermediate language)進行直譯(interpreting)，並利用**遞迴呼叫**(recursive call)與**動態繫結**(dynamic binding)的機制來製作直譯器。

以 Java 目前的中間語言 -- **位元組碼**(bytecode)來說，它和 Java 的原始碼之間並沒有特別的關聯，在直譯上也沒有較便利之處。語法樹在先天上就如同高階語法一般，它和**原始碼**(source code)之間有最密切的關連。因此本論文主張以語法樹當作中間語言，並提出對它進行直譯的技術。

語法樹是製作**剖析器**(parser)時就會用到的資料結構(Aho, Sethi, and Ullman, 1985)，本論文以從上而下(top-down)的遞迴呼叫來對語法樹進行直譯。我們直接在語法樹的各種節點中配置相對應的函數，透過動態繫結的機制，在各種節點以適當的處理方式進行直譯。我們也利用**例外機制**(exception mechanism)來處理迴圈、return 與 throw 敘述在**控制流程**(control flow)上的脫離。為配合目前被廣汎使用的位元組碼，我們還提供了語法樹與位元組碼之間的銜接機制，讓兩種不同的直譯機制可以互相呼叫。

本系統目前的直譯器為了配合 Java 現有的的程式庫，暫時是以 Java 語言實作，因此在執行效能上會比較慢，但還在可接受的範圍內。未來若將本系統改以 C++(Stroustrup, 2000)配合組合語言實作，效能應該會和現有的直譯器不相上下。另外，我們還可將語法樹以**樹形碼**(tree-code)的形式輸出成為另一種**類別檔**(class file)，當成發佈(release) Java 軟體的另一種仍具有可攜性的選擇。

從**軟體工程**(software engineering)的角度來觀察，利用語法樹做中間語言讓系統開發者比較容易實作直譯器，也不易發生錯誤。此外，日後在開發**除錯**(debug)功能時，原始碼與語法樹之間的對應既直接又自然，因此在實作上將更為容易。若進一步在語法樹的節點上添加圖形展示的功能，則直譯器便能夠同時配合除錯機制與展示能力，在程式語言與資料結構的教學上對學生提供直覺性的幫助。

## 參考文獻

向賢偉，“以 OpenGL 建構的 3D 導覽系統”, 淡江大學資訊管理學系碩士班碩士論文，*淡江大學資訊管理學系*，2008 年 1 月。

廖賀田、陶彥樺，“Java 程式之探索器”, *Communication of IICM*, 4 (3), pp.83-94, Sep., 2001.

Aho, A. V., Sethi, R., and Ullman, J. D., *Compiler Principles, Techniques, and Tools*, Addison-Wesley, 1985.

Arnold, K., Gosling, J., and Holmes, D., *The Java Programming Language 4th edition*, Addison-Wesley, August 27, 2005.

BeanShell-Lightweight Scripting for Java, *http://www.beanshell.org/*, Accessed on Dec. 2007

BlueJ-The interactive java environment, *http://www.bluej.org/*, Oct. 2007

DynamicJava, *http://koala.ilog.fr/djava/*, Version 1.1.5, Jun. 2002, Accessed on Dec. 2007

Eclipse Foundation, "The Java Implementation of the platform debug component," *http://www.eclipse.org/eclipse/debug/index.php*, Accessed on Feb. 2008.

Gosling, J., Joy, B., Steele Jr., G. L., and Bracha, G., *The Java Language Specification Third Edition*, Addison-Wesley, June 24, 2005.

Lindholn, T. and Yellin, F., *Java Virtual Machine Specification 2nd Edition*, Addison-Wesley, 1999.

Microsoft Corporation, *http://www.microsoft.com/windows/products/windowsxp/default.mspx/*, Accessed on Feb. 2008.

Stroustrup, B., *The C++ Programming Language 3rd Edition*, Addison-Wesley , Feb. 11, 2000.

Sun Microsystems, Inc., *Java 2 Platform Standard Edition*, *http://java.sun.com/javase/index.jsp*, Accessed on Dec. 2007.

Walrath, K., Campione, M., Huml, A., and Zakhour, S., *The JFC Swing Tutorial: A Guide to Constructing GUIs 2nd Edition*, Addison-Wesley, March 5, 2004.