國立交通大學

資訊科學與工程研究所

碩 士 論 文

利用最小成本的文法剖析來產生機器碼

Code Generation with a Least-Cost Parsing Technique

研 究 生：徐德發

指導教授：楊武 教授

中 華 民 國 九 十 九 年 八 月

利用最小成本的文法剖析來產生機器碼

學生：徐德發　　指導教授：楊武博士

國立交通大學資訊科學與工程研究所

摘要

在編譯器領域中有許多自動產生程式碼的技術，一種稱作 BURS 的樹狀樣式比對配合動態規劃的演算法是其中流行的一種做法。但是在本篇論文中，樣式比對改由文法剖析的方式來進行，亦即，利用文法剖析器依據指令文法來針對中間表示法(Intermediate Representation)去進行樣式比對。但指令文法是一個模糊文法，因此要由一個更爲通用的剖析器，稱作 GLR parser 來處理。因爲我們語法剖析器也是採用原本 BURS 演算法中的成本機制來去找尋成本最低的剖析方式，所以若要解決在模糊文法中所存在的兩種衝突：S/R conflict 和 R/R conflict 時，必須保證不會影響 GLR parser 尋找最低成本的結果。因此，我們針對這兩種衝突，提出新的方法來處理，前者是在 runtime 之前處理，藉由分析文法的方式來找出在該文法中每個 S/R conflict 相關的規則，並判斷哪些規則屬於 shift，哪些規則屬於 reduce，然後比較彼此的成本大小；後者則是在 runtime 時處理，針對由相同衝突衍伸出來的剖析堆疊做比較，比較的是其剖析路徑。這兩個解決衝突的技術雖無法解決所有的衝突，但從測試的結果中仍可看出當這兩個技術被執行時，確實可以減少 GLR parser 剖析所需的時間。此外，因爲我們要將 GLR Parser 當作一個指令選取器，來跟 BURS 演算法做個比較，因此我們會將 GLR Parser 整合到一個現存的編譯器，而該編譯器的指令選取機制就是使用 BURS 演算法。但我們並未完全取代該編譯器的後端部分，而是要用 GLR parser 來取代原本的 BURS 指令選取機制，然後比較兩者所產生出來的指令序列是否相同。

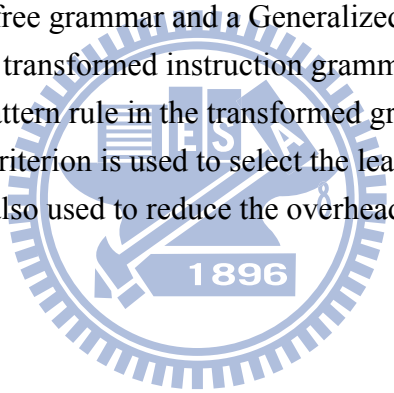
Code Generation with a Least-Cost Parsing Technique

Student: Te-Fa Hsu Advisor: Dr. Wuu Yang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

There are many automatic code generation techniques in a compiler. A tree-pattern matching algorithm with dynamic programming called BURS is one of the most popular. However, in this thesis, pattern matching is performed as a parsing process for code generation. The set of pattern rules in an instruction grammar is transformed into a context-free grammar and a Generalized LR parser is used to do the matching work with the transformed instruction grammar since the grammar is usually ambiguous. Each pattern rule in the transformed grammar is equipped with a numeric cost and this cost criterion is used to select the least-cost result among all the final parsing results and is also used to reduce the overhead of parsing process.



誌謝

這篇論文的完成，首先我要感謝我的指導教授楊武博士。在研究所的時光，多虧老師耐心、細心的指導，並且適時的指點我正確的方向，所以我才能解決論文中的諸多難題，真的很感激老師的許多付出。同時，也要感謝每位口委老師所給予的諸多寶貴意見。

而這段學生生涯，也要感謝實驗室裡的每位學長和同學及學弟妹們，大家共同營造了一個真的是充滿「正面能量」和歡笑的實驗室，也讓我在遇到瓶頸的時候可以放鬆心情，真的很感謝實驗室的每個成員。

當然，我能夠讀到研究所，最重要的是我爸媽在背後的諸多付出，因爲沒有他們的辛勞，不可能有今天的我，同時也要感謝容忍我脾氣的哥哥。當然，還有我的女友，因爲她，讓我人生有了努力奮鬥的目標。

謝謝大家，謝謝老天!

The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 96-2628-E-009-014-MY3, NSC 98-2220-E-009-050, and NSC 98-2220-E-009-051 and a grant from Sun Microsystems OpenSparc Project.

目錄

圖目錄

表目錄

第一章 緒論

在編譯器領域中有許多自動產生程式碼的技術，一種稱作 BURS(Bottom-Up Rewrite System)的樹狀樣式比對配合動態規劃(tree pattern matching with dynamic programming)的演算法是其中流行的一種做法。但是在本篇論文中，樣式比對的過程是改由文法剖析的方式來進行，亦即，利用文法剖析器(parser)依據指令文法來針對 Intermediate Representation 去進行樣式比對。

一個指令文法可以設計成去代表某個目標機器的指令集，而且指令文法通常是一個模糊的文法，因此要處理這樣的文法就需要一個更爲通用的剖析器，也就是稱作 GLR parser 的語法剖析器。GLR parser 在剖析的過程中會找出所有的剖析樹(parse tree)，因此必須有一套機制去幫助 GLR parser 如何從這些不同的剖析樹當中去挑選一個最好的語法剖析方式。

在 BURS 演算法中所採用的成本機制則移植到此語法剖析器，亦即文法中的每個規則都有一個數值型態的成本，此成本可以代表該規則對應的機器指令所消耗的記憶體使用量或機器周期等。如此 GLR parser 即可計算每個剖析樹的總成本並選出總成本最小的剖析樹，亦即最佳的目標碼。

因爲我們希望將 GLR Parser 當作一個指令選取器(instruction selector)，來跟 BURS 演算法做個比較，因此我們會將 GLR Parser 整合到一個現存的編譯器，而該編譯器的指令選取過程就是使用 BURS 演算法。但我們並未完全取代該編譯器的後端部分，而是要用 GLR parser 來取代原本的 BURS 指令選取機制。因此，真正要取代的是 BURS 機制在執行時剖析的兩個階段：第一階段是以由下往上的追蹤方式，依照 BURS 的有限狀態機表格去爲 Intermediate Representation 的每個節點指定一個狀態，而第二階段則是以由上往下的追蹤方式去萃取每個節點該執行其指令文法中的什麼規則。亦即，BURS 機制在執行時的指令選取過程改由我們 GLR Parser 去剖析 Intermediate Representation 來找出最佳的指令序列。

另外，因位指令文法是個模糊的文法，所以其中存在有許多的 S/R conflict 和 R/R conflict，因此我們希望應用成本機制來解決這樣的衝突，但又不能違背我們 GLR parser 的最終目標—找出總成本最小的剖析樹，因此提出新的方法來解決這兩種衝突。對於 S/R conflict，我們是在產生 GLR parser 的同時，藉由分析文法的方式來找出在該文法中每個 S/R conflict 相關的規則，在判斷哪些規則屬於 shift，哪些規則屬於 reduce，然後比較採取 shift 策略和採取 reduce 策略的成本大小再做決定，但並非每個 S/R conflict 能夠被解決，主要就是可能違背我們 GLR parser 的最終目標。對於 R/R conflict，我們是在 GLR parser 執行剖析工作時，針對由相同衝突衍伸出來的剖析堆疊做比較，比較他們的剖析路徑。這兩個解決衝突的技術經由我們實驗的測試可看出可減輕一些 GLR parser 執行剖析時的負擔。

第二章 相關研究

2.1 模糊文法

如果一個與前後文無關文法(context-free grammar)所產生的句型會有兩個或兩個以上的剖析樹(parse tree)時，則表示此文法是模糊的(ambiguous grammar)。舉一個簡單的與前後文無關文法(context-free grammar)爲例。

●(1)Stmt→if Expr Stmt else Stmt
●(2)Stmt→if Expr Stmt

Figure 2.1 一個模糊文法的範例

「if Expr if Expr Stmt else Stmt」是一個字串，如果使用 Figure 2.1 中的模糊文法來去剖析此字串，則會如 Figure 2.2 中所示有兩種不同的剖析過程。

Stmt → if Expr Stmt else Stmt	Stmt → if Expr Stmt
→ if Expr (Stmt) else Stmt	→ if Expr (Stmt)
→ if Expr (if Expr Stmt) else Stmt	→ if Expr (if Expr Stmt else Stmt)
(a)	**(b)**

Figure 2.2 兩種不同的剖析過程

而這兩種不同的剖析方式代表在剖析的過程中會有兩種不同的剖析樹(parse tree)，如 Figure 2.33，因此 Figure 2.1 中的文法就是一個模糊文法。

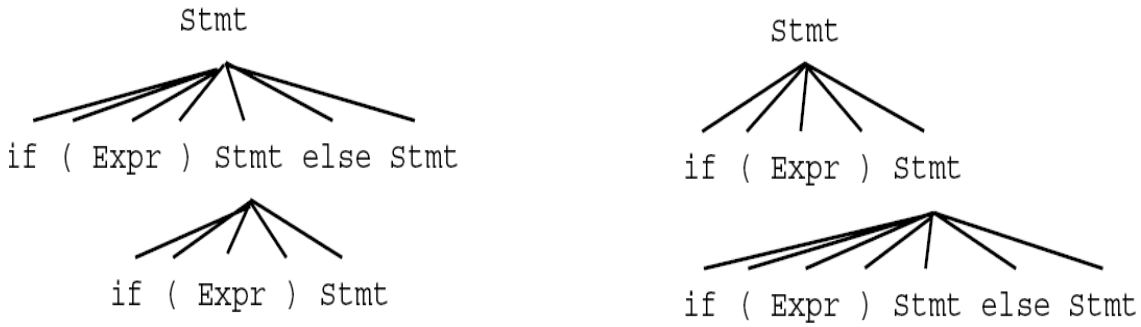


Figure 2.3 兩種不同的剖析樹

一個文法是一個模糊文法，表示它存在衝突的情況，而所謂的衝突可以包括 shift/reduce 衝突或 reduce/reduce 衝突。以下就介紹這兩種衝突。

2.1.1 shift/reduce conflict

Figure 2.4 是一個含有 shift/reduce 衝突的範例文法，假設其中的 ICONST_32 和 IADD32 都是 terminal，其餘則是 non-terminal。當一個剖析器的剖析堆疊的內容是「IADD32」，此時又讀取到一個新的標誌是 ICONST_32，這時可以選擇使用第一條規則進行縮減策略，但同樣的也可以採取移動策略，因此在這樣的情況下就形成了 shift/reduce conflict。Bison 對於這樣的衝突所採取的解決方法就是固定選擇移動策略，所以在這個例子中，Bison 就會選擇「reg16: IADD32 ICONST_32 reg32」這條規則。

●(1)reg32→ICONST_32
●(2)reg32→IADD32 ICONST_32 reg32
Figure 2.4 含有 shift/reduce 衝突的模糊文法

2.1.2 reduce/reduce conflict

Figure 2.5 是一個含有 reduce/reduce conflict 的範例文法，假設其中的 IADD32 是 terminal，其餘則是 non-terminal。當一個剖析器的剖析堆疊的內容是「IADD32 reg16 reg16」，這時可以選擇使用第一條規則進行縮減策略，但同樣的也可以採用第二條規則進行縮減策略，因此這時候就形成了 reduce/reduce 衝突。Bison 對於這樣的衝突所採取的解決方法就是固定選擇縮減策略中的第一條規則，因此在這個例子就會選擇「reg16: IADD32 reg16 reg16」這條規則。

●(1)reg16→IADD32 reg16 reg16
●(2)reg32→IADD32 reg16 reg16
Figure 2.5 含有 reduce/reduce 衝突的模糊文法

2.2 Generalized LR Parser(GLR parser)

2.2.1 GLR parser 的觀念

GLR parser 的概念最先是被學者 Tomita[10]所提出，用來當作自然語言方面的剖析器，以處理在自然語言文法中容易出現的模糊情形，並且保留在剖析過程中所有可能的剖析方式。之後，GLR parser 的應用也從原本處理自然語言的範疇擴展到處理程式語言。GLR parser 的運作機制其實很類似一般的 LR parser，但前者可以處理模糊文法，並且使用一個圖形資料結構來紀錄剖析過程所有的可能性。

2.2.2 Graph-Structure Stack(GSS)

當 GLR parser 去處理模糊文法時，所使用的剖析堆疊結構和一般 LR parser 中所使用的剖析堆疊結構會有所不同。前者所使用的剖析堆疊結構其實是被稱做 Graph-Structure StackSS(GSS)的一個圖形結構。Figure 6 是一個 GSS 的例子，包含了三個各自獨立的剖析堆疊，其中每個節點都會有一個狀態和一個指標去指向其父節點。這三個剖析堆疊會有共用父節點的情況，主要是希望減輕 GSS 所形成的負擔。

GLR parser 是一個通用的剖析器，也就是其使用模糊文法去剖析一個句子時，若遇到 shift/reduce 或 reduce/reduce 衝突的存在，其剖析過程的堆疊會產生分裂，形成兩個各自獨立的堆疊，這跟 LR parser 這樣的剖析器就有所不同。

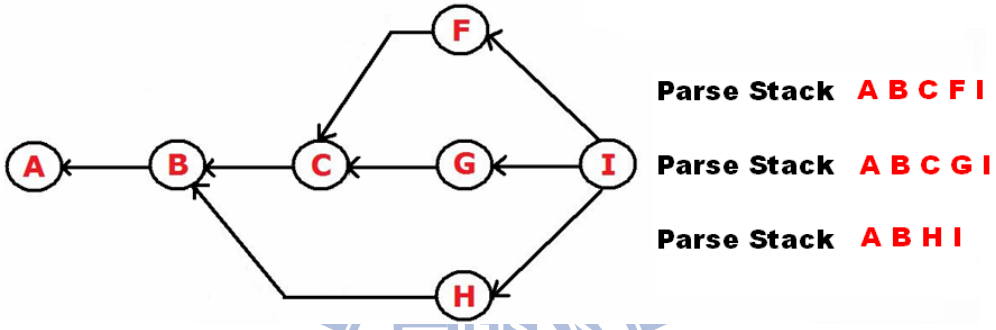


Figure 2.6 一個 GSS 的範例
在此 GSS 中，含有三個不同的剖析堆疊

維持整個 GSS 會有一個問題就是需花費許多的時間與空間，這是因爲當一個剖析堆疊要採取 shift 策略時，一個新的節點會被推入到堆疊中，而當要依照某個規則去採取 reduce 策略時，該規則的右側(right-hand side)有多少個元素，則堆疊就必須推出多少個節點。在這些節點被推入與推出圖狀堆疊的過程中，指標必須不斷的在這些節點之間來回移動，因此就會浪費不少的時間與空間。

2.2.3 兩種常見的實作技術

維持整個 GSS 會浪費不少的時間與空間，因此有兩種主要的改進技術應用在 GLR parser 的實作上。

第一個技術就是讓在 GSS 中，從相同剖析堆疊衍伸出來的所有不同的剖析堆疊都會使用指標去指向其共同的剖析堆疊，如此一來這些不同的剖析堆疊就可以共享相同的一連串節點。也就是圖型堆疊是使用鏈結串列的方式來實作，每個節點都擁有指標去指向其父節點，因此當一個剖析堆疊要分裂時，新加入的節點會用指標去指向其父節點。

第二個技術就是去合併擁有相同頂端狀態的剖析堆疊。也就是在剖析過程中，兩個各自獨立發展的剖析堆疊在擁有相同的頂端狀態時，會共享一個節點，

如此一來也可以節省 GSS 所需要的節點數了。

2.3 一個 Java 虛擬機器：CVM

2.3.1 CVM 的介紹

本篇論文因爲希望能夠比較兩種指令選取的機制：以 BURS 的樹狀樣式比對配合動態規劃的演算法爲基礎的指令選取機制和以我們 GLR parser 爲基礎的指令選取機制，因此我們需要一個在指令選取的部分就是使用 BURS 演算法的編譯器。我們選擇了一個應用在嵌入式環境，稱作 CVM(CDC Virtual Machine)的 Java 虛擬機器所含有的一個即時編譯器(just-in-time compiler)，此即時編譯器在指令選取的部分就是使用 BURS 演算法。

在 CVM 中，不只含有一個傳統的直譯器(interpreter)，也有一個動態編譯器(dynamic compiler)，也就是即時編譯器(just-in-compiler)。當一個 Java 程式碼被當成 CVM 的輸入時，會被轉成一連串的 Java byte-codes，然後這些 bytecodes 會被執行，執行時 CVM 有一個 profile 機制去辨認出哪些 Java 程式碼相對應的 bytecodes 執行的次數是比較多的，而在超過一個門檻值時，就會認爲這些程式碼是比較常執行的，而這樣比較常被執行的程式碼就稱作「Hot method」。

並非經常被執行的程式碼都會交由直譯器來處理，而經常被執行的「Hot method」則是交由即時編譯器來處理：將相關的 bytecodes 翻譯成目標機器的原生指令(native machine codes)，如此一來就不需要每次執行那些 Hot method 時都還要重新翻譯，也就是可以直接執行該 bytecodes 編譯好的原生機器碼來取代原本 bytecodes 直譯的過程。

CVM 這樣的 Java 虛擬機器使用直譯器和即時編譯器的主要理由在於執行一段 bytecodes 編譯好的原生機器碼會比原本 bytecodes 直譯再執行的過程還要快，但如果以 bytecodes 的編譯時間及執行時間和 bytecodes 直譯再執行的過程相比，則前者反而比較慢，因此就出現了一個取捨的問題，哪些程式碼會經常執行或者僅執行一兩次而已。因此 CVM 使用一套測量機制來去辨認出哪些程式碼執行的次數是比較多的，也就值得花時間由即時編譯器去進行編譯，否則由直譯器在程式碼要被執行時再來直譯就可以。

2.3.2 CVM 的即時編譯器

CVM 的即時編譯器產生原生機器碼的元件可以分成兩部分，也就是前端和後端。前端部分的元件會將一段 Java bytecodes 轉換成一種樹狀的 Intermediate Representation，而許多關於 bytecodes 的資訊，例如資料型態和數值等都隱藏在這個 Intermediate Representation(IR)中。前端元件也會處理其他的事情，像是驗證和安全檢查的工作、針對 IR 去進行多種最佳化的工作。後端部分主要是利用

由 Java Code Select(JCS)這個剖析器產生器(parser generator)所產生出來的一個目的碼產生器(code generator)去剖析由前端所產生出來的 IR。在剖析的過程中會執行相關的函式去產生目標機器的原生指令。

後端部分還有三個重要的元件：暫存器管理者(Register Manager)、常數池管理者(Constant Pool Manager)和堆疊管理者(Stack Manager)。暫存器管理者主要負責的是持續追蹤在編譯過程中所有暫存器的使用情況，也使用一個資料結構來去紀錄已評估過的計算表示式(evaluated expressions)目前是儲存在記憶體或者是暫存器中。常數池管理者則是用來管理所產生出來的原生機器碼所參考道的 32 位元或 64 位元的常數。堆疊管理者則負責管理在編譯時會被推入到 Java expression stack 中的一些方法的參數。後端部分的元件可以利用堆疊管理者和暫存器管理者所負責的事項來去追蹤在 Java 堆疊中被當作參數使用的一些數值。

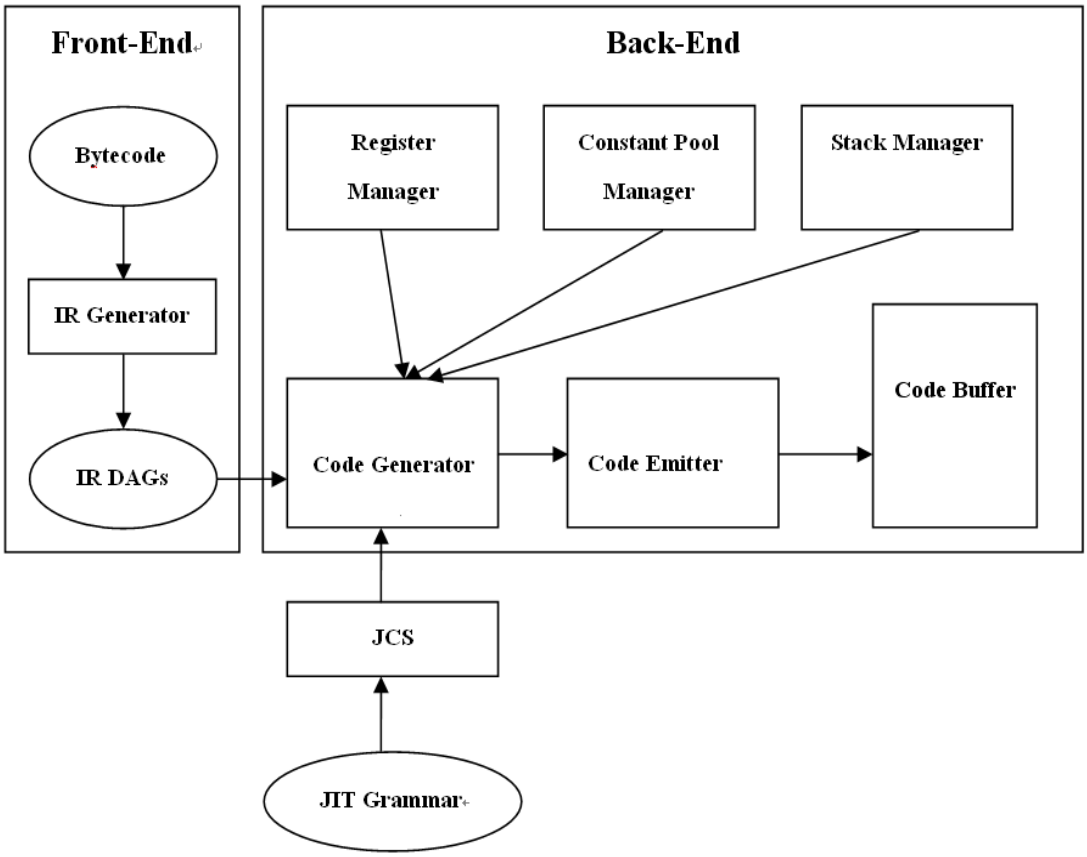


Figure 2.7 即時編譯器的架構

接下來則是要分別介紹在 CVM 的即時編譯器中和我們研究相關的元件，包括 IR 產生器(IR generator)，Java Code Select 以及目的碼產生器。

2.3.2.1 即時編譯器的 IR

Java 虛擬機器是一個以堆疊爲基礎的架構(stack-based architecture)，因此 Java

bytecodes 在設計上就有堆疊的概念在裡頭。在評估一個計算表示式時會使用一個堆疊，有些值會被推入到堆疊中，等到要計算時再被推出來，然後結果計算完了，該結果又會被推回到堆疊中，而一些方法的參數和傳回值也都會被傳遞到堆疊中以供後續的使用。這樣以堆疊概念所設計的架構適合用在直譯的情況下，但若應用在以暫存器爲基礎的架構中則顯得較沒效率。因此，CVM 的即時編譯器的目標就是要將堆疊導向的 bytecodes 翻譯成原生機器碼。

在翻譯的過程中，前端元件會將 bytecodes 翻譯成與硬體平台無關的 intermediate representation，而所有隱含的 Java 語意(Java semantics)都會顯現在這樣的表示法中。這樣樹狀的表示法架構很適合用來表示將堆疊導向的 bytecodes 轉換成以暫存器爲基礎的底層機器架構的原生碼。這樣的 IR 比起堆疊導向的 bytecodes 來說，也是更爲適合去分析和操控程式碼，亦即，進行最佳化的處理或是移除一些重複的計算過程。IR 其實是一連串的表示樹(expression tree)所組成的，而這些表示樹如果以後序追蹤的方式來看的話，其實就容易看出所代表的 bytecodes 計算式。

由 IR 產生器最終所產生出來的 IR 會被傳給目的碼產生器，然後再由目的碼產生器轉成原生機器碼。而目的碼產生器是以後續追蹤(post-order traversal)的方式來走訪整個樹狀 IR。

每拜訪一個 IR 的節點，就會去讀取其中的資訊來知道該執行怎樣的指令，而計算完的結果也可以傳遞給其他 IR 中相關的節點以便更進一步的使用。當目的碼產生器走訪完整個樹狀的 IR 時，Java 原始程式的翻譯過程就算是完成了，也就是 Java 原始程式轉成位元組碼再轉成原生機器碼的過程就完成了。

2.3.2.2 Java Code Select 和目的碼產生器

Java Code Select (JCS)是一個產生剖析器的工具，其功能類似 Yacc 或 Bison，但不同的是 Yacc 或 Bison 所建立出來的剖析器是以字串樣式比對(pattern matching with streams)爲基礎，JCS 所產生出來的剖析器則是以樹狀樣式比對爲基礎(pattern matching with tree-based data structures)。

JCS 所接受的輸入是一個稱作 JIT grammar 的指令文法(instruction grammar)，文法中的每個規則都會有一個數值成本，然後，JCS 就能依此產生一個目的碼產生器，稱作 JIT code generator。這個目的碼產生器其實就是個剖析器，以由下往上且由左而右的樣式比對方式去針對 Intermediate Representation 來進行剖析。

由於 JCS 是以 BURS(Bottom-Up Rewrite System)這樣的樹狀樣式比對演算法爲基礎，所以在產生目的碼產生器的同時，其實也產生了一個 BURS 的 tree automaton(tree automaton)以供使用。而在建造 tree automaton 的時候，已經利用成本的觀念來去決定每個情況下最佳的指令序列，因此目的碼產生器在執行時剖析 IR，只需要兩個階段就可以找出該 IR 最佳的指令序列，然後再去執行相關的

函式以產生原生機器碼。

目的碼產生器在真正產生原生機器碼之前會設定一些細節或者是限制。然後，目的碼產生器就會呼叫在目的碼發出器(code emitter)中的相關函式去真正產生原生機器碼。目的碼發出器是位在即時編譯器的最底層，能爲特定的硬體架構去產生其相關的目標機器指令。

2.4 BURS 理論

BURS(Bottom-Up Rewrite System)演算法是一種樹狀樣式比對配合動態規劃的演算法，是應用在編譯器領域中自動產生程式碼的技術，而此技術最主要的特點在於會根據所給定的輸入文法去產生一個目的碼產生器和一個 tree automaton 資料。BURS 演算法因爲在執行時間之前的編譯-編譯時間(compile-compile time)就利用動態規劃的方式作了大量的計算，考慮所有可能的情況，然後根據成本的觀念去找出最佳的解，因此其所建立出來的目的碼產生器在真正執行時就不需要太多的計算，所以速度相當的快。但也因爲大量的計算集中在建立 tree automaton 時，能夠有效率去產生 BURS tree automaton 的過程就相當重要了。

對於一些典型的機器架構來說，由於直接產生 BURS 的 tree automaton 所需的狀態和狀態轉移表是很耗費空間的，所以顯得沒有效率。一些學者如 Proebsting [9]就提出藉由許多不同的優化技術來去減輕 BURS 狀態轉移表所佔的空間，因此可以讓以 BURS 爲基礎所設計的目的碼產生器的產生器(code-generator generator)顯得更有效率一些。

2.4.1 建立 BURS tree automaton

假設在一個指令文法中有 6 個規則(Figure 2.8)，然後假設每個規則的成本都是 1。現在以這個例子來說明 BURS 演算法建立 tree automaton (tree automaton)時的過程。

tree automaton 其實也是一種狀態機，但處理的是樹狀架構而非傳統的狀態機所處理的字串。因此在 BURS 演算法中所使用的指令文法就含有樹狀架構的概念，更精確的說是二元樹的概念，也就是其中的 terminal 可分成零維、一維和二維，而 BURS 在建造 tree automaton 時，就分別爲零維、一維和二維的 terminal 來建造它們的狀態轉移表，如此合起來就形成一個完整的 BURStree automaton。因此我們假設在 Figure 2.8 的文法中，LOCAL32、INEG32、IADD32、ISUB32 和 ASSIGN 都是 terminal，且 LOCAL32 是零維，INEG32 是一維， IADD32、ISUB32 和 ASSIGN 則是二維。在說明如何建立自動機的轉移表時，要先說如何建立所需要的狀態。

- (1)statement→ASSIGN LOCAL32 reg32 **1**
- (2)reg32→LOCAL32 **1**
- (3)reg32→INEG32 reg32 **1**
- (4)reg32→IADD32 reg32 reg32 **1**
- (5)reg32→ISUB32 reg32 reg32 **1**
- (6)reg32→IADD32 reg32 INEG32 reg32 **1**

Figure 2.8 一個指令文法的範例

在 tree automaton 中，一個狀態會包含一些資訊：能夠完全比對的規則(full match rule)、部分比對的規則(partial match rule)和使用包覆(closure)運算涵蓋進來的規則，以及相關規則的成本。所謂的包覆運算是指當一個狀態的完全比對規則的左半部 non-terminal(left-hand-side non-terminal)會出現在哪些規則的右半部中，就將那些規則涵蓋進來。以下分別說明建造每個狀態的過程。

2.4.1.1 建立 tree automaton 的每個狀態

在建造狀態 1 時，會先考慮維度是零的 terminal，而在此零維的就只有 LOCAL32，也就是要考慮「LOCAL32」。能夠和「LOCAL32」達成完全比對的就是「reg32: LOCAL32」這條規則，達成部分比對的則是「statement: ASSIGN LOCAL32 reg32」這條規則，此外，因爲「reg32: LOCAL32」是完全比對規則，其左半部的 non-terminal 是 reg32，因此可以根據包覆運算將右半部含有 reg32 這個標誌的規則給涵蓋進來。Figure 2.9 顯示了狀態 1 所包含的相關資訊。

State 1

(1)full match rule

- reg32→[LOCAL32]

(2)partial match rule

- statement→ASSIGN [LOCAL32] reg32

(3)closure

- statement→ ASSIGN LOCAL32 [reg32]
- reg32→INEG32 [reg32]
- reg32→IADD32 [reg32] reg32
- reg32→IADD32 reg32 [reg32]
- reg32→ISUB32 [reg32] reg32
- reg32→ISUB32 reg32 [reg32]
- reg32→IADD32 [reg32] INEG32 reg32
- reg32→IADD32 reg32 INEG32 [reg32]

Figure 2.9 狀態 1 的內容

在建造狀態 2 時，因爲維度是零的 terminal 已經處理完了，所以考慮維度是 1 的 terminal，而在此一維的就只有 INEG32，也就是要考慮「INEG32 reg32」。能夠和「INEG32 reg32」達成完全比對的就是「reg32: INEG32 reg32」這條規則，達成部分比對的則是「reg32: IADD32 reg32 INEG32 reg32」這條規則，此外，因爲「reg32: INEG32 reg32」是完全比對規則，其左半部的 non-terminal 是 reg32，因此又可以根據包覆運算將右半部含有 reg32 這個標誌的規則給涵蓋進來。Figure 2.10 顯示了狀態 2 所包含的相關資訊。

State 2

(1)full match rule

- reg32→[INEG32 reg32]

(2)partial match rule

- reg32→IADD32 reg32 [INEG32 reg32]

(3)closure

- statement→ ASSIGN LOCAL32 [reg32]
- reg32→INEG32 [reg32]
- reg32→IADD32 [reg32] reg32
- reg32→IADD32 reg32 [reg32]
- reg32→ISUB32 [reg32] reg32
- reg32→ISUB32 reg32 [reg32]
- reg32→IADD32 [reg32] INEG32 reg32
- reg32→IADD32 reg32 INEG32 [reg32]

Figure 2.10 狀態 2 的內容

在建造狀態 3 時，因爲維度是一的 terminal 已經處理完了，所以考慮維度是二的 terminal，而在此二維的就有 IADD32、ISUB32 和 ASSIGN，因此分別考慮這三個 terminal 的情況。在此先處理 IADD32，也就是要考慮「IADD32 reg32 reg32」。能夠和「IADD32 reg32 reg32」達成完全比對的就是「reg32: IADD32 reg32 reg32」這條規則，但卻沒有任何規則可以達成部分比對，此外，因爲「reg32: IADD32 reg32 reg32」是完全比對規則，其左半部的 non-terminal 是 reg32，因此又可以根據包覆運算將右半部含有 reg32 這個標誌的規則給涵蓋進來。Figure 2.11 顯示了狀態 3 所包含的相關資訊。

State 3

(1)full match rule

- reg32→[IADD32 reg32 reg32]

(2)partial match rule

(3)closure

- statement→ ASSIGN LOCAL32 [reg32]
- reg32→INEG32 [reg32]
- reg32→IADD32 [reg32] reg32
- reg32→IADD32 reg32 [reg32]
- reg32→ISUB32 [reg32] reg32
- reg32→ISUB32 reg32 [reg32]
- reg32→IADD32 [reg32] INEG32 reg32
- reg32→IADD32 reg32 INEG32 [reg32]

Figure 2.11 狀態 3 的內容

在建造狀態 4 時，就來考慮 ISUB32 這個二維的 terminal，因此考慮「ISUB32 reg32 reg32」。能夠和「ISUB32 reg32 reg32」達成完全比對的就是「reg32: ISUB32 reg32 reg32」這條規則，但也如同狀態 3 一般沒有任何規則可以達成部分比對，此外，因爲「reg32: ISUB32 reg32 reg32」是完全比對規則，其左半部的 non-terminal 是 reg32，因此又可以根據包覆運算將右半部含有 reg32 這個標誌的規則給涵蓋進來。Figure 2.12 顯示了狀態 4 所包含的相關資訊。

State 4

(1)full match rule

- reg32→[ISUB32 reg32 reg32]

(2)partial match rule

(3)closure

- statement→ ASSIGN LOCAL32 [reg32]
- reg32→INEG32 [reg32]
- reg32→IADD32 [reg32] reg32
- reg32→IADD32 reg32 [reg32]
- reg32→ISUB32 [reg32] reg32
- reg32→ISUB32 reg32 [reg32]
- reg32→IADD32 [reg32] INEG32 reg32
- reg32→IADD32 reg32 INEG32 [reg32]

Figure 2.12 狀態 4 的內容

在建造狀態 5 時，則是處理 ASSIGN 這個二維的 terminal，因此考慮「ASSIGN

LOCAL32 reg32」。能夠和「ASSIGN LOCAL32 reg32」達成完全比對的就是「reg32: ASSIGN LOCAL32 reg32」這條規則，但也如同狀態 3 和狀態 4 一般沒有任何規則可以達成部分比對，此外，因爲「reg32: ASSIGN LOCAL32 reg32」是完全比對規則，其左半部的 non-terminal 是 statement，因此又可以根據包覆運算將右半部含有 statement 這個標誌的規則給涵蓋進來，但並未有這樣的規則存在。Figure 2.13 顯示了狀態 5 所包含的相關資訊。

State 5

(1)full match rule

- statement→[ASSIGN LOCAL32 reg32]

(2)partial match rule

(3)closure

Figure 2.13 狀態 5 的內容

這時候就剩規則 6 還沒被處理，所以考慮 IADD32 另外一個子樣式：[IADD32 reg32 INEG32 reg32]。能夠和「IADD32 reg32 INEG32 reg32」達成完全比對的就是「reg32: IADD32 reg32 INEG32 reg32」這條規則，但也如同狀態 3、狀態 4 和狀態 5 一般沒有任何規則可以達成部分比對，此外，因爲「reg32: ASSIGN LOCAL32 reg32」是完全比對規則，其左半部的 non-terminal 是 reg32，因此又可以根據包覆運算將右半部含有 reg32 這個標誌的規則給涵蓋進來。Figure 2.14 顯示了狀態 6 所包含的相關資訊。

State 6

(1)full match rule

- reg32→[IADD32 reg32 INEG32 reg32]

(2)partial match rule

(3)closure

- statement→ ASSIGN LOCAL32 [reg32]
- reg32→INEG32 [reg32]
- reg32→IADD32 [reg32] reg32
- reg32→IADD32 reg32 [reg32]
- reg32→ISUB32 [reg32] reg32
- reg32→ISUB32 reg32 [reg32]
- reg32→IADD32 [reg32] INEG32 reg32
- reg32→IADD32 reg32 INEG32 [reg32]

Figure 2.14 狀態 6 的內容

在建立好 tree automaton 的每個狀態後，就可以開始針對每個 terminal 建立

其狀態轉移表。以下就來說明建立狀態轉移表的過程。

2.4.1.2 建立 tree automaton

在這個指令文法的 6 個規則中，零維的 terminal 只有 LOCAL32，而其 tree automaton 表格就會是個零維的表格，也就是一個常數而已。因爲狀態 1 的完全比對規則是「reg32: LOCAL32」，因此，LOCAL32 的 Tree Automaton Table 就是狀態 1。這是表示當以 BURS 爲基礎所產生的目的碼產生器，例如：前面章節所提到即時編譯器中的目的碼產生器，去剖析樹狀的 Intermediate Representation 時，若碰到 LOCAL32 這個節點則會直接指定狀態 1 給該節點。

零維的 terminal 處理完，就來處理一維的 terminal，也就是 INEG32，其 tree automaton table 會是個一維的表格。Figure 2.15 顯示 INEG32 的轉移表，左邊欄位代表其子節點的狀態，右邊欄位則代表父節點相對應的狀態。現在舉其中的兩個項目來說明此表格建立的過程。

當子節點的狀態是 3 時，從 Figure 2.11 中狀態 3 的內容可以發現其完全比對規則(full-match rule)是「reg32: IADD32 reg32 reg32」，這就是子節點會執行的規則，而當子節點執行完此規則時，父節點 INEG32 該執行什麼規則可從狀態 3 的包覆運算結果來看。因爲父節點是 INEG32，所以在包覆運算結果中只有「reg32: INEG32 reg32」這個規則符合，亦即，父節點該執行這條規則，而此規則相對應的狀態是 2。因此，當子節點的狀態是 3 時，父節點的狀態應該是 2。

當子節點的狀態是 5，父節點的狀態卻是-1，這是因爲從狀態 5 的內容來看，當子節點在執行其完全比對規「statement:ASSIGN LOCAL32 reg32」時，由於狀態 5 的包覆運算結果是空的，也就是父節點 INEG32 無法執行什麼規則，因此父節點的狀態被設定爲-1。這其實從 Figure 2.8 所示的指令文法中就可以看出 INEG32 的子節點所會執行的規則必須是左半部的 non-terminal(left-hand-side non-terminal)爲 reg32，而非 statement。

descedent state	result state
1	2
2	2
3	2
4	2
5	-1
6	2

Figure 2.15 INEG32 的 tree automaton 表格

一維 terminal 處理完，就來處理二維 terminal，也就是 IADD32、INEG32 和 ASSIGN，在此僅以 IADD32 的轉移表建立過程來做說明。Figure 2.16 顯示 IADD32 的轉移表，左邊欄位代表其左子節點的狀態，右邊欄位代表其右子節點的狀態，中間欄位則代表父節點相對應的狀態。現在也舉其中的兩個項目來說明此表格建立的過程。

left desc. state	1	2	3	4	5	6
1	3	6	3	3	-1	3
2	3	6	3	3	-1	3
3	3	6	3	3	-1	3
4	3	6	3	3	-1	3
5	-1	-1	-1	-1	-1	-1
6	3	6	3	3	-1	3

Figure 2.16 IADD32 的 tree automaton 表格

當左子節點的狀態是 3，右子節點的狀態是 2 時，從 Figure 2.11 中狀態 3 的內容和 Figure 2.10 中狀態 2 的內容可以發現，兩者各自的完全比對規則(full-match rule)是「reg32: IADD32 reg32 reg32」和「reg32: INEG32

reg32」，亦即，這是兩個子節點各自會執行的規則，而當兩個子節點執行完各自的規則時，父節點 IADD32 該執行什麼規則可從狀態 3 和狀態 2 的包覆運算結果來看。因爲父節點是 IADD32，所以在包覆運算結果可發現有兩個選擇符合所需，也就是「reg32: IADD32 reg32 reg32」和「reg32: IADD32 reg32 INEG32 reg32」，如果父節點選擇執行前者的話，則總共執行的規則就是 Figure 2.17 中的 Option 1 的三條規則，所以所花成本是 1+1+1=3；選擇執行前者的話，則總共執行的規則就是 Figure 2.17 中的 Option 2 的兩條規則，所以所花成本是 1+1=2。因此相比之下，這時候父節點就該執行後者的規則，所以父節點的狀態就是 6。

Option 1：Total Cost = 3

reg32：IADD32 reg32 reg32

reg32：INEG32 reg32

reg32：IADD32 reg32 reg32

Option 2：Total Cost = 2

reg32：IADD32 reg32 reg32

reg32：IADD32 reg32 INEG32 reg32

Figure 2.17 兩組互比大小的規則

這兩組規則在經過大小比較後，Option 2 的成本較小所以被保留。

當左子節點的狀態是 3，右子節點的狀態也是 3 時，從 Figure 2.11 中狀態 3 的內容可以發現，兩者的完全比對規則(full-match rule)都是「reg32: IADD32 reg32 reg32」，亦即，這是兩個子節點各自會執行的規則，而當兩個子節點執行完相同的規則時，父節點 IADD32 該執行什麼規則可從狀態 3 的包覆運算結果來看。因爲父節點是 IADD32，所以在包覆運算結果中只有「reg32: IADD32 reg32 reg32」這個規則符合，亦即，父節點該執行這條規則，而此規則相對應的狀態是 3。因此，父節點的狀態就是 3。

至於 ASSIGN 和 ISUB32 的 tree automaton 轉移表的建立過程如同 IADD32 一般，就不再贅述。Figure 2.18 和 Figure 2.19 顯示兩個轉移表的內容。

left desc. state \ right descendent state	1	2	3	4	5	6
1	5	5	5	5	-1	5
2	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1

Figure 2.18 ASSIGN 的 tree automaton 表格

left desc. state \ right descendent state	1	2	3	4	5	6
1	4	4	4	4	-1	4
2	4	4	4	4	-1	4
3	4	4	4	4	-1	4
4	4	4	4	4	-1	4
5	-1	-1	-1	-1	-1	-1
6	4	4	4	4	-1	4

Figure 2.19 ISUB32 的 tree automaton 表格

從以上的說明可以發現，BURS 演算法花了大量的時間在建立 tree automaton 轉移表，而表格在建立時就有使用成本觀念來找出最佳的指令序列，因此當 BURS 所產生的剖析器，或稱作目的碼產生器，在執行時就只需要查詢這些表格以找出所剖析的對象最佳的指令序列。

2.4.2 執行時的指令選取工作

在執行時間時 BURS 技術需要兩個階段來完成指令選取(instruction

selection)的工作。

第一階段就是使用由下往上的追蹤方式(bottom-up traversal)來去走訪整個 IR 的樹狀資料結構(intermediate-representation tree)，同時指定給每個節點一個狀態，而一個狀態代表了會在以該節點爲跟節點的子樹中，達成完全相配(full match)或部分相配(partial match)的所有可能規則。因爲是用由下往上的追蹤方式去走訪 IR，所以樹葉節點是直接根據它自己擁有的一些資訊，例如：運算子的型態，以決定該被指定爲什麼狀態。而每個內部節點除了要參考自己的資訊，還要參考其子節點的資訊，來決定該被指定爲什麼狀態。在依照由動態規劃(dynamic programming)方式建立的狀態轉移表去完成指定狀態給每個節點的工作後 就可以辨認出最佳的指令序列(optimum instruction sequence)爲何。

第二階段則是採用由上往下的追蹤方式(top-down traversal)，來去走訪整個 IR 的樹狀資料結構，然後根據前一階段所得到的資訊來取決定該執行其指令文法中的什麼規則，然後執行每條規則所相對應的函式。

從以上兩階段的說明來看 可以發現所需的計算量遠小於 tree automaton 的計算量，因此目的碼產生器執行的速度就相當的快。

2.5 樹狀樣式比對和字串樣式比對

從學者 Shankar 等人[15]的研究可以發現要找出一個字串文法(string grammar)G 所有能推導出的序列是等價於找出一個樹狀文法(tree grammar)G'所有能推倒出的序列。還有，我們可以藉由將樹狀文法 G'中所有樹狀規則(tree patterns)的右側(right-hand-side)轉換成一個前序表示法來去建立一個字串文法 G，而轉換後所得到的字串文法 G 就可能是個模糊文法。

因爲樹狀樣式比對(Tree Pattern Matching)和字串樣式比對(String Pattern Matching)某種程度是相當類似的，而 BURS 演算法使用的就是樹狀樣式比對，我們 GLR parser 剖析器則是使用字串樣式比對，因此在這個章節，我們舉了一個簡單的範例來說明這兩個技術在概念上的差異。

Figure 2.20 是一個樹狀樣式比對的例子，包含了一個所要剖析的目標樹以及兩條用來進行目標比對的樹狀規則(tree pattern rule)。而整個樹狀比對的過程是：首先使用 Rule 1 去比對目標樹，然後執行該規則相關的函式並產生目的碼「SUB R, B, C」，然後原本的目標樹變成像 Figure 2.21 中的樣子，所以很顯然可以再用 Rule 2 去比對該目標樹，因此就會執行該規則相關的函式並產生目的碼「MOV A, R」，最終該目標樹變成了 **ε**，也就完成了樹狀樣式比對。

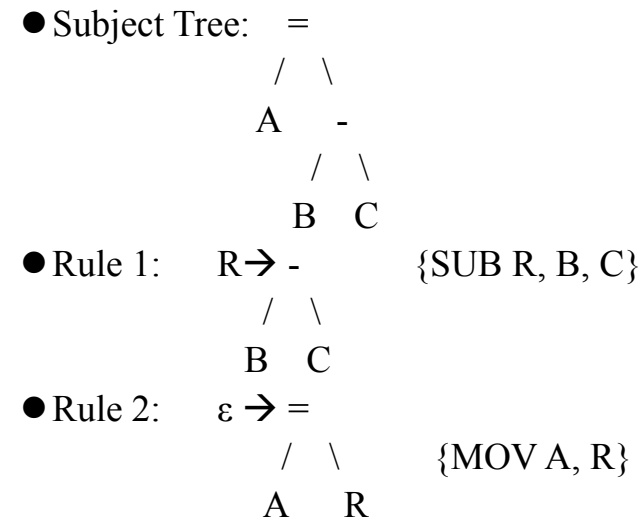


Figure 2.20 一個樹狀樣式比對的例子

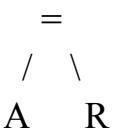


Figure 2.21 暫時的剖析結果

Figure 2.22 是一個字串樣式比對的例子，仍然如同 Figure 2.20 一般，包含了一個所要剖析的目標字串以及兩條用來進行目標比對的規則。而整個字串比對的過程是：首先使用 Rule 1 去比對目標字串，然後執行該規則相關的函式並產生目的碼「SUB R, B, C」，然後原本的目標字串變成「=AR」，所以很顯然可以再用 Rule 2 去比對該目標字串，因此就會執行該規則相關的函式並產生目的碼「MOV A, R」，最終該目標字串變成了 ε，也就完成了字串樣式比對。

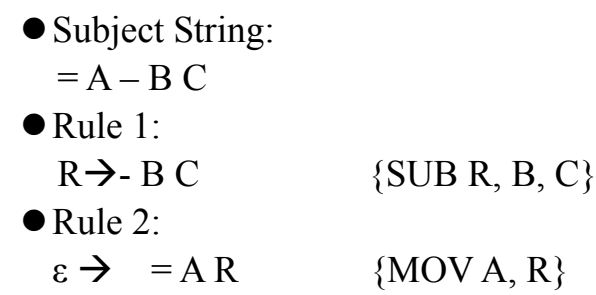


Figure 2.22 一個字串樣式比對的例子

從以上的說明可以發現樹狀樣式比對和字串樣式比對是可以互相轉換的，因此在將我們 GLR parser 應用在以 BURS 的樹狀樣式比對演算法爲基礎的指令選取過程中，就需要一項重要的轉換：將一個 IR 的樹狀架構變成字串形式，在這樣的轉換後，我們 GLR parser 才有辦法進行剖析。

第三章 GLR parser 設計與實作

在這一章將會說明實作我們 GLR parser 的過程，以及在使用 cost 機制的情況下，如何解決文法中所含有的衝突。

3.1 GLR parser 的實作

藉由實作 2.2.2 章所說的兩個主要改進 GLR parser 實作的技術，Graph-Structure Stack 確實可以變小。但因爲本篇論文主要希望將 GLR parser 應用在指令選取的過程，所以爲了更加簡化其 GSS 的架構，因此以陣列來實作該堆疊。Figure 3.1 列出我們 GLR parser 的演算法。

Input: The input string
Output: The input string's least-cost parse stackMethod:

```
1.  GLRParser(Input String)
2.  {
3.      FrontierSet is the set which collects each stack-top state of each parse stack;
4.      TempSet is the set which collects the newly-added stack-top state during one
            parsing stage in the parsing process;
5.      Push a new state 0 into FrontierSet;
6.      while(1){
7.          T := Choose a token from the input string
8.          While(FrontierSet is not empty) {
9.              S1 := Choose a stack-top state from FrontierSet arbitrarily;
10.             for each action in ActionTable[S1, T] {
11.                 case Shift：
12.                     Search the entry ActionTable[S1, T] to find what the
                            new stack-top state S2 is;
13.                     Add the new stack-top state S2 to the TempSet;
14.                     Add the new parent-child relationship "S2 is S1's parent"
                            to the ParentArray;
15.                 case Reduce：
16.                     S2 := DoReduceAction(S1, T);
17.                     while(S2 has a reduce action to do)
18.                         S2 := DoReduceAction(S2, T);
19.                     Add the new stack-top state S2 to the TempSet;
20.                 case Error：
21.                     discard the parse stack represented by this stack-top state S1;
```

```
22.                 }
23.             }
24.             for each state in the TempSet {
25.                 Find out what parse stacks are derived from the same R/R conflict;
26.                 Check whether their parsing paths are the same or not.
27.                 If the same, use their RuleList to calculate their own costs.
28.                 Compare their costs, and leave the least-cost parse stacks.
29.             }
30.             FrontierSet := TempSet;
31.             if(T is the last token in the input string) break;
32.         }
33.     }

34.     int DoReduceAction(state_number S1, token T)
35.     {
36.         Search the entry ActionTable[S1, T] to find what the reduce rule R
                should be executed;
            Add the rule R into this parse stack's RuleList;
37.         NT := Left hand side non-terminal of rule R;
38.         L := the length of Right hand side of rule R;
39.         for i := 1 to L {
40.             Use the ParentArray to find the state S1' s parent state P;
41.             S1 := P;
42.         }
43.         Search the entry ActionTable[P, NT] to find what the
                new stack-top state S2 is.
44.         Add the new parent-child relationship "S2 is P's parent"
                to the ParentArray;
45.         return S2;
46.     }
```

Figure 3.1 GLR parser 演算法

3.1.1 實作技術

因爲使用陣列來實作 GSS，所以有三項實作上的技術需使用到。

- **記錄每個剖析堆疊的頂端**
- **使用一對數字來定址每個節點**
- **記錄每個剖析堆疊的規則**

接下來我們就分別敘述這三項實作上的技術。

【技術 1: 記錄每個剖析堆疊的頂端】

爲了控制 GSS 中所有各自獨立的剖析堆疊，我們使用一個鏈結串列的資料結構，稱作 Frontier List，來去紀錄圖狀結構中每個剖析堆疊頂端的狀態。當我們 GLR parser 從輸入序列中讀取一個新的標誌(token)時，我們稱作一個新的階段開始。在每一個新階段開始時，若 Frontier List 中有多個相同的狀態存在，則表示有多個各自獨立的剖析堆疊有相同的堆疊頂端。

在剖析過程結束時，由此鏈結串列中即可知道最終到底有多少個剖析堆疊存在於 GSS 中，亦即，有多少種不同的剖析方式同時存在。因此，就需要一套機制以便我們 GLR parser 能在多個剖析方式中選出一個。這部分我們使用的是成本機制，在後面章節會作說明。

【技術 2: 使用一對數字來定址每個節點】

因爲使用節點的方式來建立 GSS 去維持整個剖析的過程會比較花時間與空間，又我們希望減少剖析過程中當執行縮減策略時，節點指標所需的移動時間，所以我們並未用節點方式來維持整個 GSS，而是利用一對數字來定址原本 GSS 中的每個節點。

每一個節點所對應的一對數字**(state_num, state_count)**會紀錄該節點的狀態以及該裝態在相同的狀態中是第幾次出現的情況，因此每一對數字可用來唯一定址一個節點。爲了建構原本在 GSS 中每個節點之間的父子關係(parent-child relationship)，因此使用一個二維陣列，稱作 Parent Array 來做這樣的紀錄。

當一個剖析堆疊在執行過程中要使用某個規則去進行縮減策略時，該規則的右半部有多少個標誌，原本在 GSS 中就需要推出多少個節點以找出其縮減路徑(reduction path)中的根節點，但在我們 GLR parser 中則是使用該剖析堆疊頂端節點的一對數字，然後去 Parent Array 中搜尋該次數以找出其根節點，以利後續其他操作的進行。如此一來可以省卻許多節點增刪和指標移動的時間。

Figure 3.2 是一個縮減路徑(reduction path)的範例。其中剖析堆疊是以節點來建立，它的頂端狀態是 9，而當此剖析堆疊要使用一個規則去進行縮減策略，而該規則的右半部有 3 個標誌時，則此剖析堆疊就必須推出三個節點，如此才可以找到所需要的根節點，也就狀態爲 4 的節點。

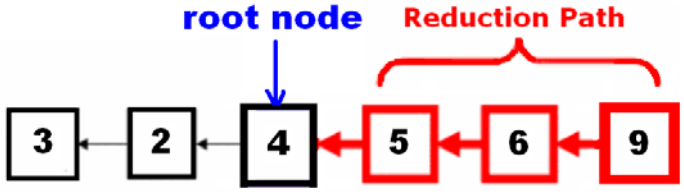


Figure 3.2 一個縮減路徑的範例

當此剖析堆疊要執行一個縮減規則，然後該規則的

右半部標誌個數是 3，則該堆疊就必須要推出圖中
的三個紅色節點，因此該三個節點就稱爲縮減路徑。

【技術 3: 記錄每個剖析堆疊的規則】

因爲我們希望將 GLR parser 應用在指令選取的過程，所以當我們 GLR parser 剖析一個目標句子時，必須要詳實紀錄在剖析過程中，每個各自獨立的剖析堆疊會執行文法中的哪些規則，因此我們使用一個稱作 Rule List 的相鄰串列(adjacent list)的來作紀錄。

在使用 Rule List 完全紀錄 GSS 中每個剖析堆疊所會執行的規則有哪些之後，可以輔助我們判斷最終該留下哪一個剖析堆疊。

3.1.2 成本機制

當我們 GLR parser 根據一個模糊文法去剖析一個目標句子時，若遇到有移動/縮減或 reduce/reduce 衝突的情況，則原本在 GSS 中的一個剖析堆疊就會因此而分裂成兩個剖析堆疊，也就等於有兩個不同的剖析樹存在，亦即，存在著兩組不同的指令序列會被編譯器後端的目的碼產生器給使用，但這當然不合理，因此需要有一套機制讓我們在所有不同的剖析樹中做選擇。

我們使用如同 BURS 演算法中的成本觀念去設計一個成本機制來解決因爲模糊文法所引起的這種問題。在我們的成本機制中，每個文法中的規則都會有一個數值型態的成本(numeric cost)，此成本可以代表該規則對應的機器指令所需要的執行時間、消耗的記憶體使用量或機器周期等，而在將文法中的每個規則搭配一個成本後就可以幫助我們計算在剖析過程中所有剖析樹的總成本。

因爲在剖析樹中的每個內部節點其實就是一個縮減節點(reduce node)，暗示著在該節點會有一個縮減的動作發生，加上我們的 GLR parser 使用的是由下往上的剖析技術(bottom-up parsing)，因此剖析樹中的每個內部節點其實都只會被它所有的子節點給影響，而從這樣的觀點來看，一個剖析樹的成本其實就是去計算它所有內部節點的成本，也就是我們 GLR parser 在剖析過程中若執行了一個縮減的動作時，勢必會有一個規則被執行，而該規則的數值成本就會被加入到目前剖析樹的總成本中，如此一來，每個剖析樹有了自己的總成本，我們 GLR parser 就可以據此來比較各剖析樹的成本大小，而保留成本最小的剖析樹。

接下來我們使用一個例子來說明成本觀念。在 figure 3.3 是一個範例文法，存在著三條規則，其成本分別是 20、15 和 10。當使用我們的 GLR parser 依此文法來去剖析一個目標句子「a + a - a」時，其剖析過程會形成像 figure 3.4 中的剖析樹，從這個剖析樹就看得出來，剖析過程分別會執行的規則是規則 3、規則 3、規則 3、規則 2 和規則 1，在將每個規則的成本相加起來就可以得到該剖析樹的總成本，因此「a + a - a」的剖析樹其成本就是 10+10+10+15+20=65。

- (1) A→A + A　　20
- (2)A→A – A　　15
- (3)A→a　　10

Figure 3.3 一個含有成本觀念的文法範例

每個規則都附有一個數值成本，如此可輔助計算每個剖析樹的成本

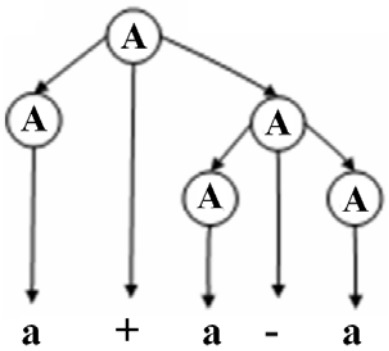


Figure 3.4「a+a-a」的一個剖析樹

在這樣的成本機制下有一個問題需要特別注意，就是若多個剖析樹有相同的總成本，則無法分辨該選擇哪一個。例如使用 figure 25 中的文法去剖析目標句子「a + a - a」就會產生兩個不同的剖析樹(figure 27)，而很明顯可以看出兩個剖析樹都具有相同的成本，也就是 65。因此使用者在設計文法中每個規則的成本時必須特別注意這樣的問題。

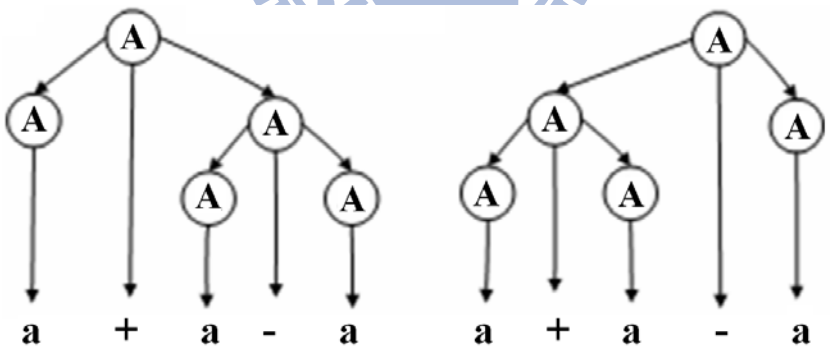


Figure 3.5 剖析「a+a-a」產生的兩個剖析樹

3.1.3 運作範例

我們使用一個假設的模糊文法來說明我們 GLR parser 在實作上的方式。如同前面所述，若將這個模糊文法交由 NewBison 當作輸入文法來處理後，可發現會有 shift/reduce 衝突或 reduce/reduce 衝突的存在，而產生出來的有限狀態機就可以保留這些所有的衝突資訊。

在 NewBison 產生我們 GLR parser 和狀態機之後，我們以該 GLR parser 藉

由狀態機去剖析「minus NUM」這樣假設的字串來顯示每一步的過程中幾個重要資料結構的變化，以說明我們 GLR parser 實作的技術。

Stage 0

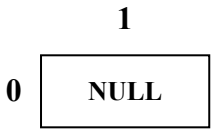
在一開始的時候，一般 GLR parser 使用的 GSS 中只會有一個起始狀態爲 0 的節點，因此我們 GLR parser 所使用的 Frontier Set 中也就只會有一個狀態爲 0 的節點。而此狀態 0 是第一次出現，因此使用(0, 1)來代表此第一次出現的狀態 0。又因爲此狀態 0 並沒有任何的父節點，所以(0, 1)在 Parent Array 的欄位是 NULL。

GSS：

FrontierSet：

{0}

ParentArray：



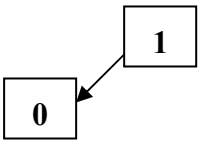
Stage 1: minus

在這階段開始的時候，Frontier Set 僅有一個狀態 0 的元素，我們 GLR parser 再根據這個狀態 0 和新讀入的標誌 minus 去有限狀態機表格(FSM table)中搜尋相關的欄位，判斷該執行移動策略或縮減策略。而假設由有限狀態機表格可知共有兩個策略該執行：

「Shift 1」、「Reduce 2」。

1st action: FSMTable[state 0][minus] = “Shift 1”

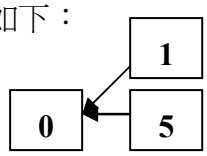
一個狀態爲 1 的節點會被推入到 GSS 中。所以這時候 GSS 的內容如下：



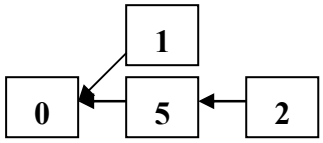
2nd action: FSMTable[state 0][minus] = “Reduce 2”

使用規則 2 來縮減，假設其右半部並沒有任何的標誌，縮減路徑(reduction path)長度就是 0，所以 GSS 中不需要推出任何的節點，因此可以找出根節點就是狀態 0，又假設該規則的左半部 non-terminal 爲 TokenA，因此我們 GLR parser 就去有限狀態機表格中搜尋 FSMTable[state 0][TokenA]的欄位，發現該執行「go to

5」的移動策略，因此一個狀態爲 5 的節點被推入到堆疊中，則 GSS 的內容變成如下：

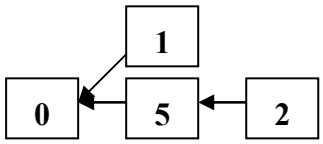


然後我們 GLR parser 再去有限狀態機表格查詢 FSMTable[state 5][minus]的欄位，發現要執行「shift 2」，所以一個狀態爲 2 的節點被推入到堆疊中。則 GSS 的內容又變成如下：



此時會有兩個各自獨立的剖析堆疊存在，而因爲狀態 1、狀態 5 和狀態 2 都是第一次出現，所以分別使用(1, 1)、(5,1)和(2, 1)這樣的數字組合來去定址 GSS 中的這些節點，而這三個節點在 GSS 中的父節點分別是狀態 0、狀態 0 和狀態 5，這樣的父子關係資訊也就會紀錄在 Parent Array 中。還有，堆疊頂端是(2, 1)的剖析堆疊則在 Rule List 紀錄它所執行過的規則。

GSS：

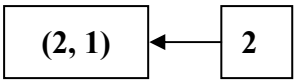


FrontierSet
{(1,1), (2,1)}

Parent Array:

	1
0	NULL
1	(0, 1)
2	(5, 1)
3	
4	
5	(0, 1)

Rule List:

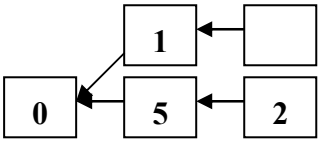


Stage 2: NUM

在這階段開始的時候，GSS 中有兩個剖析堆疊，因此 Frontier Set 包含了兩個元素，我們 GLR parser 再分別根據這些狀態和新讀入的標誌 NUM 去有限狀態機表格中搜尋相關的欄位，判斷該執行移動策略或縮減策略。先處理第一個剖析堆疊，然後假設由有限狀態機表格可知要執行「shift 2」的策略，所以此時 GSS 變成如下：

1st action: FSMTable[state 1][NUM] = "Shift 2"

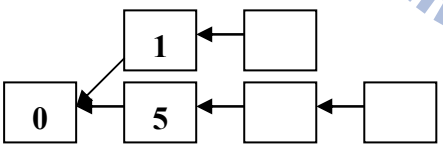
一個狀態爲 2 的節點會被推入到 GSS 中。所以這時候 GSS 的內容如下：



接著再來處理第二個剖析堆疊，而假設由有限狀態機表格可知共有「**Shift 5**」、「**Reduce 3**」兩個策略該執行：

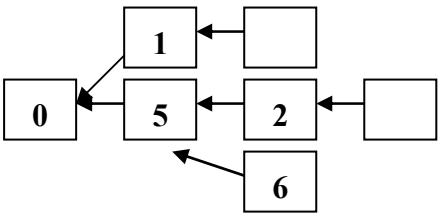
2nd action: FSMTable[state 2][NUM] = "Shift 5"

一個狀態爲 5 的節點會被推入到 GSS 中。所以這時候 GSS 的內容如下：

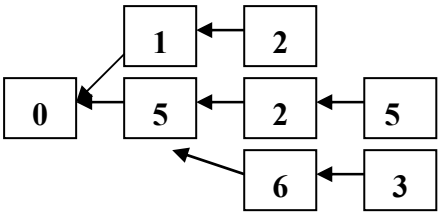


3rd action: FSMTable[state 2][NUM] = "Reduce 3"

使用規則 3 來縮減，假設其右半部有 1 個元素，則其縮減路徑(reduction path)長度就是 1，所以 GSS 中需要推出一個節點，但由於該節點被另外一個剖析堆疊所使用，因此不會真正推出該節點。因爲縮減路徑爲 1，因此可找到所需的根節點就是狀態 5，又假設規則 3 的左半 non-terminal 爲 TokenB，因此我們 GLR parser 就去有限狀態機表格中搜尋 FSMTable[state 5][TokenB]的欄位，發現該執行「go to 6」的移動策略，因此一個狀態爲 6 的節點被推入到堆疊中，則 GSS 的內容變成如下：

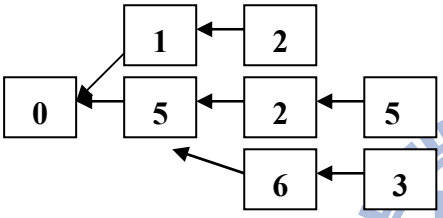


然後我們GLR parser再去有限狀態機表格查詢FSMTable[state 6][NUM]的欄位，發現要執行「shift 3」，所以一個狀態爲 3 的節點被推入到堆疊中。則 GSS 的內容又變成如下：



此時 GSS 終會有三個各自獨立的剖析堆疊存在，而每個新加入的節點都會有適當的一對數字來定址，也會將節點之間彼此的父子關係紀錄在 Parent Array 中，然後有執行過一些規則也會紀錄在該剖析堆疊的 Rule List 中。

GSS：



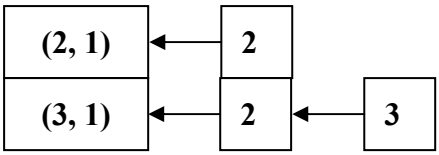
FrontierSet：

{(2,2), (5,2), (3,1)}

ParentArray：

	1	2
0	NULL	
1	(0, 1)	
2	(5, 1)	(1, 1)
3	(6, 1)	
4		
5	(0, 1)	(2, 1)
6	(5, 1)	

Rule List：



3.2 衝突解決技術

因爲當我們 GLR parser 依照一個模糊文法來剖析一個句子，而在剖析的過程中因爲 shift/reduce 衝突或 reduce/reduce 衝突而使得 GSS 中有多個剖析堆疊的存在，如此會浪費空間，加上我們 GLR parser 爲了加快剖析的過程而使用一個二維陣列來紀錄整個 GSS，因此需要有一些技術來處理這樣的衝突情況，以減輕這樣的負擔。

一般剖析器用來解決這樣衝突情況的技術如同 2.1 章所述，對 shift/reduce conflict 的解決方法是固定選擇移動策略，而對 reduce/reduce conflict 的解決方法則是固定選擇第一條的縮減規則(reduce rule)，但這樣的經驗法則並不適用於我們 GLR parser，主要原因在於我們 GLR parser 必須要能夠保證最終挑選出來的是總成本最小的剖析方式，因此我們提出不同的方法來處理這樣的衝突問題。

3.2.1 解決 shift/reduce conflict

爲了讓我們 GLR parser 在剖析時所維持的剖析堆疊數可以減少，以減輕其執行時的負擔，因此設計一個演算法(Figure 3.6)來解決 shift/reduce 衝突，使得在建立有限狀態機表格時就能解決這樣的衝突問題。由於並非在執行剖析的過程處理，因此無法知道我們 GLR parser 真正所要剖析的目標句子爲何，也無法知道一個目標句子的剖析樹的真正長相，也無法去計算出該剖析樹真正的總成本。因此需要一個方法讓我們可以在不需要知道目標句子爲何的情況下，就可以知道對於一個 shift/reduce 衝突來說，採取哪個策略可以讓我們 GLR parser 得到成本最低的剖析樹。接下來我們先以一個例子來說明我們設計該演算法的背後想法。

Input: **A context-free grammar G with S/R conflict(s)**
Output: If G‘s one S/R conflict can be solved, the S/R conflict can be solved by shift action or reduce action.
Method:

```
1. PreprocessSRconflict(G)
2. {
3.     for each S/R conflict in LR(1) parse table {
5.         int Flag := -1;    //Flag=0 represents shift action, Flag=1 represents reduce action.
```

```
6.      int Flag2 := -1;
7.      Find out all of the corresponding shift rules and reduce rules;
8.      for each shift rule RuleS do {
9.          if(Flag2 == 0) break;
10.         for each reduce rule RuleR do {
11.             if(Flag2 == 0) break;
12.             for each parent rule RuleP of RuleR do {
13.                 Compare RuleS's RHS symbols with RuleP's RHS symbols,
                        from left to right to see if each two symbol is the same. If not
                        the same, stop comparing;
14.                 Symbol1 := RuleP's symbol;
15.                 Compare RuleS's RHS symbols with RuleP's RHS symbols,
                        from right to left to see if each two symbol is the same. If not
                        the same, stop comparing;
16.                 Symbol2 := RuleP's symbol;
17.                 if(Symbol1 == Symbol2) {
18.                     if(Symbol1 == RuleR's LHS non-terminal) {
19.                         Use the RuleR's RHS to replace that LHS non-terminal
                            in RuleP's RHS;
20.                     }
21.                 }
22.                 if(RuleS's RHS == RuleP's RHS) {
23.                     if(RuleS's LHS non-terminal ==RuleP's LHS non-terminal
                        or one non-terminal can become the same with another
                        through using chain rules or
                        the two can become the same through using chain rules)
24.                     {
25.                         Flag2 := 1;
26.                         Calculate all the shift action's rules' total costs;
27.                         Calculate all the reduce action's rules' total costs;
28.                         if(shift action's cost < reduce action cost) {
29.                             if(Flag == 1) the S/R conflict can not be solved;
30.                             Flag := 0;
31.                         }
32.                         else if (shift actoin's cost > reduce action's cost) {
33.                             if(Flag == 0) the S/R conflict can not be solved;
34.                             Flag := 1;
35.                         }
```

```
36.                         }
37.                     }
38.                 }
39.                 if(Flag2 == -1) Flag2 : = 0
40.             }
41.         }
42.         if(Flag2 == 1) {
43.             if(Flag == 0)    choose the shift action;
44.             else if(Flag == 1)    choose the reduce actoin;
45.             else    choose the shift actoin;
46.       }
47.         else    the S/R conflict can not be solved;
73.   }
74.}
```

Figure 3.6 處理 shift/reduce conflict 的演算法

在 Figure 3.7 中所顯示的是 CVM 的即時編譯器所擁有的一部份指令文法，共有五條規則，其中 BOUNDS_CHECK 和 ICONST_32 是 terminal，其餘則是 non-terminal。因爲規則(1)和規則(2)的同時存在，因此該文法會在 ICONST_32 的地方有一個 shift/reduce 衝突。現在假設我們 GLR parser 要利用即時編譯器完整的指令文法來處理 Figure 3.8 中的目標句子。當我們 GLR parser 在讀取到第二個 ICONST_32 時，就會面臨到前述的 shift/reduce 衝突，而因爲這樣的衝突，所以使得在剖析過程中，我們 GLR parser 必須去維持兩個獨立的剖析堆疊。

- (1)iconst32Index→BOUNDS_CHECK ICONST_32 reg32
- (2)reg32→ICONST_32
- (3)reg32→BOUNDS_CHECK reg32 reg32
- (4)arraySubscript→reg32
- (5)arraySubscript→iconst32Index

Figure 3.7 含有 shift/reduce conflict 的模糊文法

ASSIGN INDEX IDENTITY NEW_ARRAY_BASIC ICONST_32 BOUNDS_CHECK ICONST_32 IDENTITY ARRAY_LENGTH IDENTITY NEW_ARRAY_BASIC ICONST_32 ICONST_32

Figure 3.8: 一個目標句子

當我們 GLR parser 處理完整個目標句子時，最終就會含有兩個剖析堆疊在 GSS 中，而這兩個剖析堆疊各自相對應的剖析樹則如 Figure 3.9 和 Figure 3.10 所

示，其中，前者在遇到 shift/reduce 衝突時，採取的是移動策略，後者則是採取縮減策略。觀察這兩個不同的剖析樹可以發現它們長得很像，不同點就僅在於衝突發生之處，因此若要比較這兩個剖析樹的成本大小，其實只要比較因爲衝突所造成的成本差異變化即可，亦即，採取移動策略所花的成本跟採取縮減策略所花的成本相比較。

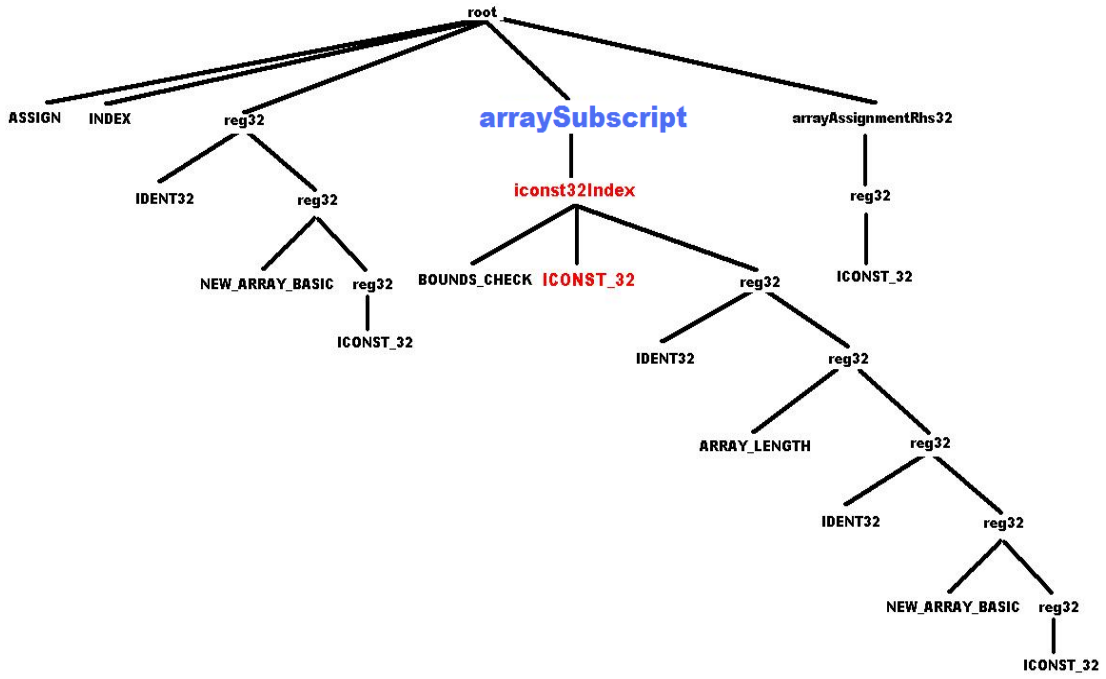


Figure 3.9: 含有移動策略的剖析樹

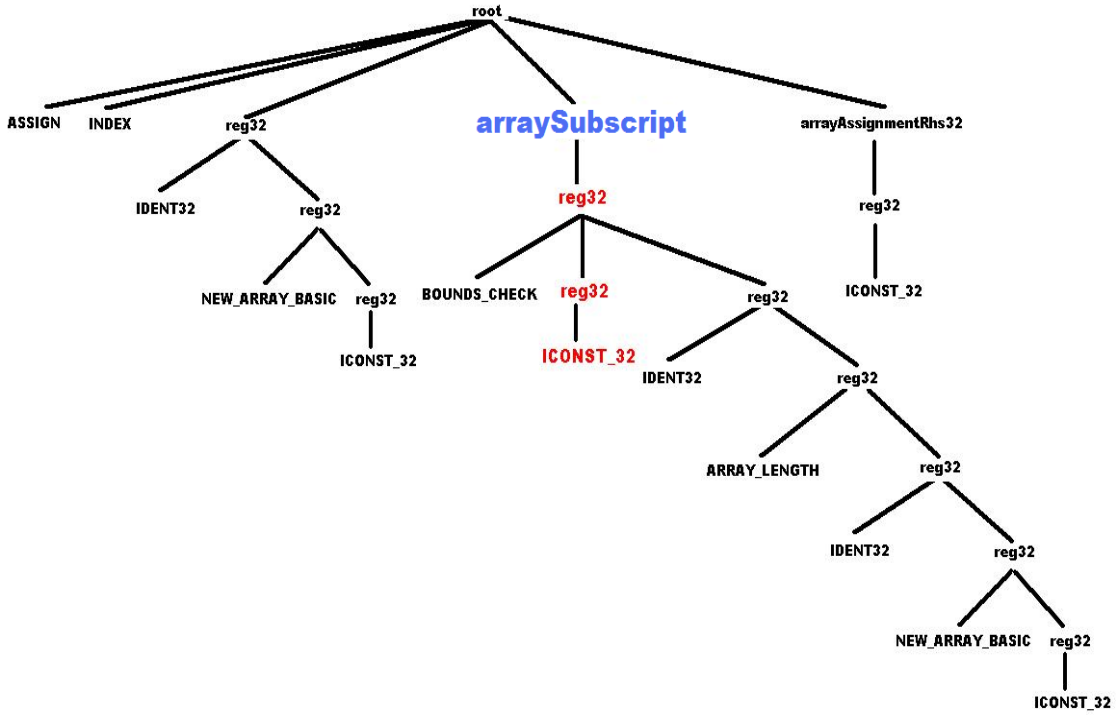


Figure 3.10 含有縮減策略的剖悉樹

從 Figure 3.7 中的規則來看可以發現，採取移動策略的剖析樹會執行的是規則(1)和規則(5)，而採取縮減策略的剖析樹則會執行規則(2)、規則(3)和規則(4)。Figure 3.11 就顯示這樣的資訊。因此我們就可以計算採取移動策略所會執行的規則的總成本，以及採取縮減策略所會執行的規則的總成本，然後再做個比較，如此就可以得知兩個剖析樹中誰的成本比較低。從此說明中可以發現，即使我們不清楚那兩個剖析樹的總成本到底爲何，但由於他們的差異僅存在於該 shift/reduce 衝突所造成的部分，因此我們依然可以只計算採取移動策略和縮減策略所造成的成本差異來做比較。

●(1)iconst32Index→BOUNDS_CHECK ICONST_32 reg32
●(5)arraySubscript→iconst32Index

(a)含有移動策略的剖析樹會執行這些規則

●(2)reg32→ICONST_32
●(3)reg32→BOUNDS_CHECK reg32 reg32
●(4)arraySubscript→reg32

(b)含有縮減策略的剖析樹會執行這些規則

Figure 3.11: 兩個不同的剖析樹分別會執行的規則

在說明完我們的背後想法時，接著要說明的就是 Figure 3.6 中，我們處理 shift/reduce 衝突的演算法。此演算法的目的就是要針對一個模糊文法中所有的 shift/reduce 衝突來做判斷，判斷是否可以在我們建立有限狀態機表格時就確定某個衝突該採取的是移動策略或縮減策略，而所決定採取的策略仍然能夠保證讓我們 GLR parser 最終依然能找到總成本最小的剖析方式。在演算法一開始，會針對每個 shift/reduce 衝突去處理，首先需要先找出所有相關的移動規則和縮減規則，然後針對每個移動規則和每個縮減規則的所有父規則進行後續的比較。在這裡「父規則」的意義我們定義爲：若一個規則的右半部含有另外一個規則的左半部 non-terminal，則前者是後者的父規則。

在比較每個移動規則和每個縮減規則的每個父規則之前，我們必須先針對所要比較的父規則進行替換的工作，也就是要用縮減規則的右半部所有終端節點和 non-terminal 去替換其左半部 non-terminal，而該左半部 non-terminal 是位在其父規則的右半部中。倘若該父規則的右半部只含有一個該左半部 non-terminal，則可以直接進行替換，但若含有兩個以上該左半部 non-terminal，則必須要有一個方法來找出真正該替換的左半部 non-terminal 是哪一個，而使用的方法就是透過兩次掃描來比較移動規則的右半部元素和父規則的右半部元素。第一次掃描是從最右側開始一一比較每個元素是否相等，若發現父規則的一個元素和移動規則的一個元素不相同就停止，第二次掃描則是從最左側開始進行比較，直到也找到一

個不相等的元素爲止。若這兩次掃描所找到的父規則元素是同一個，且該元素就是縮減規則的左半部 non-terminal，則表示該元素就是我們所要進行替換的標的。

在使用縮減規則去替換父規則中相關的左半部 non-terminal 後，接著要比較移動規則和父規則的右半部是否完全相同，若相同的話，則要再比較兩者的左半部 non-terminal 是否相同，或者其中一個 non-terminal 可以透過文法中的某些鏈鎖規則(chain rule)縮減成另一個 non-terminal，或者兩者都可以過鏈鎖規則縮減成相同的 non-terminal，而這樣限制的原因是要避免不同的 non-terminal 所造成無法預測的成本變化。

在移動規則和父規則通過一連串的比較後，接著就要計算採行移動策略和採行縮減策略所花費的成本高低。移動策略包含的規則有移動規則本身，以及若移動規則有經過鏈鎖規則才和父規則達成相同的左半部 non-terminal，則這些鏈鎖規則會包含在移動策略中。縮減策略包含的規則有縮減規則和父規則本身，以及若父規則有經過鏈鎖規則才和移動規則達成相同的左半部 non-terminal，則這些鏈鎖規則也會包含在縮減策略中。再找出移動策略和縮減策略各自包含的所有規則後，就將每個規則的成本加總，如此就可以知道這時後移動策略和縮減策略的成本高低了，而我們需要的當然是成本較低的策略。

以 Figure 3.11 爲例，必須使用規則(2)這個縮減規則的右半部「ICONST_32」去替換其左半部 non-terminal「reg32」在其父規則的位置。因爲規則(2)的三條父規則「iconst32Index: BOUNDS_CHECK ICONST_32 reg32」、「reg32: BOUNDS_CHECK reg32 reg32」和「arraySubscript: reg32」之中，只有第二個父規則的右半部有多個 reg32，因此必須經過掃描以找出要替換的是哪一個 reg32，其餘兩個規則則可以直接進行替換，但替換後只有第二個父規則「reg32: BOUNDS_CHECK ICONST_32 reg32」的右半部和移動規則的右半部，也就是規則(1)的右半部相同。接著比較該父規則和移動規則的左半部 non-terminal 是否相同，而前者是 iconst32Index 後者則是 reg32，因此並不相同，但兩者都可以因爲使用鏈鎖規則而縮減成相同的 non-terminal。iconst32Index 因爲規則(5)而縮減成 arraySubscript，reg32 則因爲規則(4)而縮減成 arraySubscript。所以此時我們可以知道採行移動策略會執行規則(1)和規則(5)，採行縮減策略則會執行規則(2)、規則(3)和規則(4)，因此就可以進行成本的計算和比較了。

由於我們是針對一個 shift/reduce 衝突去對它所有的移動規則和縮減規則的父規則進行比較，以找出對於這一個衝突該採取的策略到底爲何，因此若每一次的比較結果都是採行移動策略或縮減策略，則最終就可確知知道該衝突的解決方法，但若有些狀況是採行移動策略，有些卻是採行縮減策略，則難以判定到底該執行哪一個策略，因此就無法處理此 shift/reduce 衝突。但若每次的比較結果發現兩種策略的成本都相同，則我們會採取移動策略。最終，針對每一個 shift/reduce 衝突來說，可以確定該執行什麼策略的就可以將這樣的資訊回饋到我們 GLR parser 所使用的有限狀態機表格中，而無法確定的話就保留原本的衝突資訊於有限狀態機表格中。

3.2.2 解決 reduce/reduce conflict

解決 reduce/reduce 衝突的方式則和解決 shift/reduce 衝突的方式不同，前者是在 GLR parser 真正執行剖析時才會運作，而非如同後者在剖析之前建立有限狀態機表格時就運作，所以前者是在執行剖析過程時，針對從相同的 reduce/reduce 衝突所衍伸出來的剖析堆疊的路徑來做判斷。

假設當我們 GLR parser 使用含有 reduce/reduce 衝突的模糊文法去剖析一個句子時，一個 GSS 中的剖析堆疊若遭遇到此 reduce/reduce 衝突，然後就會分裂成兩個以上的剖析堆疊。而這些剖析堆疊在各自執行完它們的縮減規則時，就去偵測它們的路徑是否完全相同，若是完全相同，則可表示這些剖析堆疊在未來的剖析過程中必然會有相同的發展歷程，也就是會執行相同的一群規則，因此之後增加的成本數也都會相同。於是就可以在它們之間做個比較，判斷哪個剖析堆疊目前的成本最小，然後保留成本最小的剖析堆疊，而刪除其餘的剖析堆疊，但若成本想同，則保留第一個剖析堆疊。

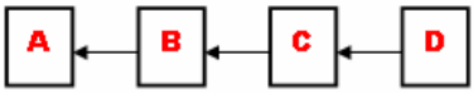


Figure 3.12 一個剖析堆疊的範例

- (1)E→B C D
- (2)E→B G
- (3)G→C D

Figure 3.13 含有 reduce/reduce conflict 的模糊文法

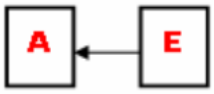


Figure 3.14 剖析結果

在執行縮減策略後，兩個分裂的剖析堆疊擁有相同內容

在 Figure 3.12 中是一個剖析堆疊，然後假設其頂端堆疊 D 會因爲 Figure 3.13 的文法中所存在的 reduce/reduce 衝突而分裂成兩個各自獨立的剖析堆疊。則在兩個剖析堆疊都處理完其縮減規則後，我們 GLR parser 會去掃描其堆疊的路徑，發現兩者的路徑都是如同 Figure 3.14 所示，因此就可以確定這兩個剖析堆疊在之後的剖析過程會相同，故可以比較兩者的大小，而保留成本較小的。

第四章 實驗設計與結果分析

前一章討論了我們 GLR Parser 設計上的理念，本章則實際執行一些實驗，包括程式碼正確性驗證、指令選取的實驗和減輕 GSS 負擔的實驗，並分析這些實驗所得的結果。

4.1 實驗環境

我們 GLR Parser 僅是一個剖析器，若要真正執行剖析的功能，則還需要其他重要資訊，包括輸入文法的有限狀態機資料，因此需要將此 GLR Parser 實作在一個剖析器產生器(parser generator)之中，以便藉由這樣的產生器來輔助我們處理輸入文法的相關資訊。

另外，爲了比較以 BURS 演算法爲基礎和以 GLR Parser 爲基礎的兩種指令選取機制，我們需要選擇一個已經實作 BURS 的編譯器環境，以便再將其機制替換成 GLR Parser 的機制。

4.1.1 剖析器產生器的選擇

我們選擇 Bison 這個剖析器產生器當作實作此 GLR Parser 的環境，也就是將此 GLR Parser 的程式碼撰寫在 Bison 當中，讓 Bison 處理一個輸入文法時，除了產生相關的有限狀態機(finite-state machine)資料外，也會產生我們 GLR Parser，則此 GLR Parser 就可以利用該文法的有限狀態機資料來執行剖析的工作。

因爲我們希望此 GLR Parser 可以處理在模糊文法中所有的衝突情況，因此需要修改 Bison 的原始碼，讓 Bison 在產生有限狀態機資料時，可以在狀態機中保留所有的衝突資訊。而這樣修改後的 Bison，我們稱作「NewBison」。

4.1.2 編譯器的選擇

我們選擇一個應用在嵌入式環境的 Java 虛擬機器，稱作 CVM(CDC Virtual Machine)。選擇 CVM 是因爲它含有一個即時編譯器(just-in-time compiler)，而該即時編譯器在指令選取的機制就是使用 BURS 演算法。如第 2.3 章所介紹的，即時編譯器的目的碼產生器(code generator)是由 Java Code Select 這個剖析器產生器(parser generator)根據一個稱爲 JIT grammar 的指令文法來產生的。然後當分析該文法時，可以發現它是一個高度模糊的文法，具有相當多的 shift/reduce conflict 和 reduce/reduce conflict 的存在，因此我們將這樣的指令文法改成 NewBison 可以接受的格式，交由 NewBison 來處理，以產生相關的有限狀態機和我們的 GLR Parser。

因爲我們希望將 GLR Parser 當作一個指令選取器(instruction selector)，所以

需要將 GLR Parser 整合到 CVM 的即時編譯器的後端部分，以取代原本的 BURS 指令選取機制。但在此真正要取代的是 BURS 機制在執行時剖析 Intermediate Representation 的兩個階段：第一階段是以由下往上的追蹤方式，依照 BURS tree automaton 去爲 Intermediate Representation 的每個節點指定一個狀態，而第二階段則是以由上往下的追蹤方式去萃取每個節點該執行其指令文法中的什麼規則。亦即，BURS 機制在執行時的指令選取過程改由我們 GLR Parser 去剖析 Intermediate Representation 來找出最佳的指令序列。

Java 原始程式的 bytecodes 經由即時編譯器的前端所轉成的 Intermediate Representation，是一種與硬體平台無關的表示法，有時候是個樹狀架構，但有時候爲了消除重複的計算，減少產生重複的程式碼，也可以是一個有向非環狀圖 (directed acyclic graph, DAG)。因爲 GLR Parser 是字串樣式比對的演算法，因此要處理的 Intermediate Representation 必須先轉成字串。如果 Intermediate Representation 是一個樹，便可直接使用前序追蹤方式將其轉成字串，但如果是一個有向非環狀圖(DAG)，則要先轉成相對應的樹，再經由前序追蹤轉成字串，如此我們 GLR Parser 才能順利的剖析該 Intermediate Representation。

Figure 4.1(a)是一個 Java 原始程式，然後 Figure 4.1(b)則是其相對應的 Java bytecodes。此 bytecodes 可以再被轉成 IR，而此 IR 可以如同 Figure 4.2(a)所示是一個樹，或者如 Figure 4.2(b)所示是一個有向非環狀圖。

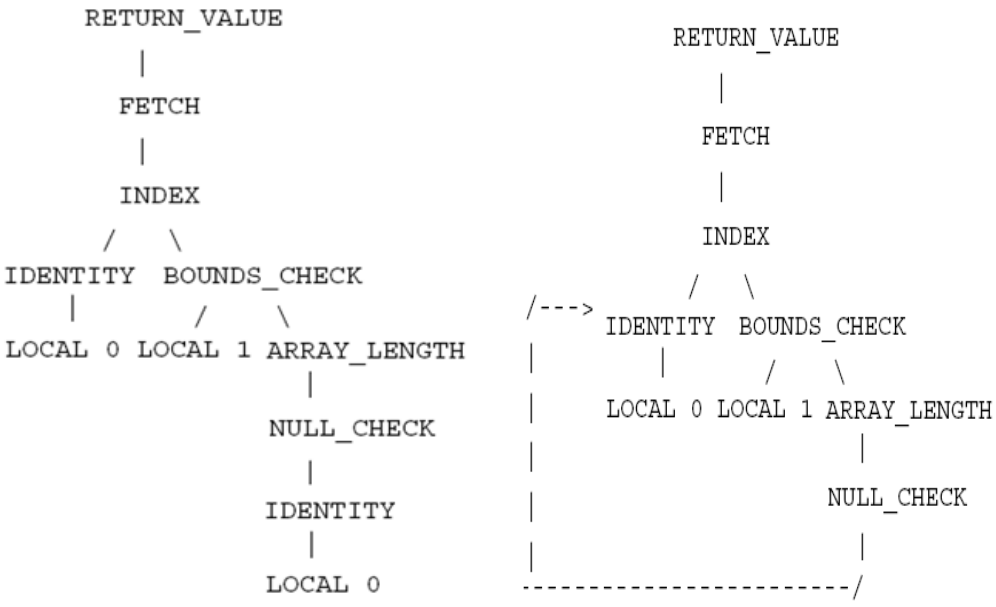
```
public static Object accessArray(Object[] arr, int idx)
{
    return arr[idx];
}
```

(a)A Java source code

```
<0> aload_0
<1> iload_1
<2> aaload
<3> areturn
```

(b)A Java bytecodes

Figure 4.1 Java 程式轉成相對應的 bytecodes



(a)IR tree **(b)IR dag**

Figure 4.2 Java bytecodes 轉成相對應的 IR

在將IR tree或IR dag轉成字串形式交由我們GLR Parser去處理後，可找出其中總成本最小的剖析方式，然後再根據我們GLR Parser所做的紀錄可以找到該剖析方式所會執行的規則是哪些，如此就找出最佳的指令序列。而在前面章節中提出減輕Graph-Structured Stack負擔的方法，亦即，解決shift/reduce conflict和reduce/reduce conflict的技術會應用在我們GLR Parser這樣的指令選取過程，以減輕剖析過程中的一些負擔。

因爲真正產生原生機器碼的責任仍然是在目的碼產生器上，我們GLR Parser僅是負責指令選取的工作，因此在將Intermediate Representation的最佳指令序列找出時，必須再傳回給原來的IR tree，好讓目的碼產生器可以萃取這樣的資訊以便執行相關的規則來產生原生碼。Figure 4.3 顯示了我們將GLR Parser整合到CVM的即時編譯器之中的情況。

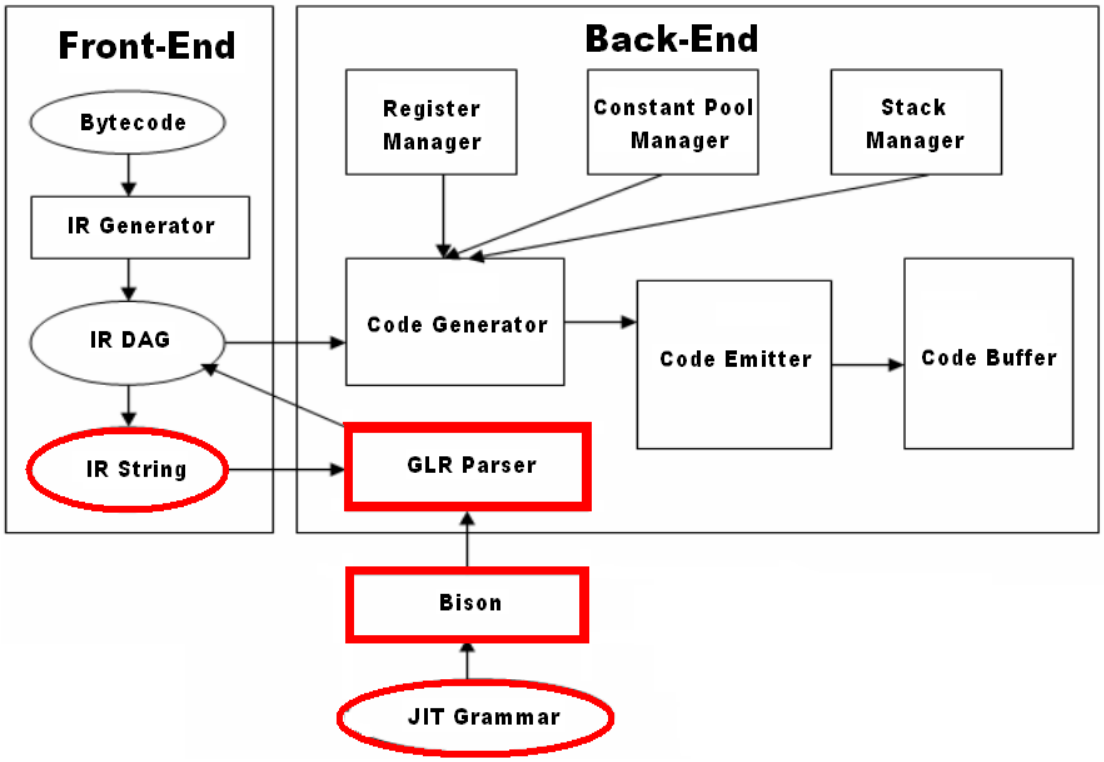


Figure 4.3 將 GLR Parser 整合到 CVM 的即時編譯器中

4.2 實驗一：GLR parser 的驗證

爲了驗證我們所撰寫的 GLR parser 能夠確實執行並忠實紀錄在剖析過程中所有各自獨立的剖析堆疊，我們撰寫了 10 個不同的文法來做驗證，每個文法儘可能擁有更多的模糊性，也就是更多的 shift/reduce 衝突或 reduce/reduce 衝突。

而這裡的驗證並未包括對解決 shift/reduce 衝突和 reduce/reduce 衝突技術的驗證。

將每一個文法檔當作 NewBison 的輸入，NewBison 就會產生能夠保留文法中所有衝突資訊的有限狀態機，然後藉由分析每一個文法的有限狀態機來針對該文法設計 10 個測試檔，使得每個測試檔被我們 GLR parser 剖析時能儘量產生多個獨立的剖析堆疊。Table 4.1 顯示了每個文法的相關資訊。

	Number of rules	Numb of FSM states	Number of shift/reduce conflicts	Number of reduce/reduce conflicts
Test Grammar 1	32	52	35	41
Test Grammar 2	7	14	0	0
Test Grammar 3	27	50	17	20
Test Grammar 4	16	17	13	14
Test Grammar 5	22	36	4	4
Test Grammar 6	13	31	4	2
Test Grammar 7	12	19	10	5
Test Grammar 8	97	208	89	5
Test Grammar 9	104	209	151	84
Test Grammar 10	269	499	435	99

Table 4.1 驗證程式正確的文法

每一個文法檔當作 NewBison 的輸入，然後 NewBison 會產生能夠保留文法中所有衝突資訊的有限狀態機以及我們的 GLR parser，然後使用產生出來的 GLR parser 根據相對應的有限狀態機去剖析該文法的每個測試檔的內容。

爲了確認我們 GLR parser 處理每一個測試檔都能夠正確而完整的紀錄所有的剖析堆疊，我們針對每個測試檔，利用人工的方式依照相關的有限狀態機去畫出每一個測試檔最終的 GSS 的全貌，然後將我們人工所得到的結果跟 GLR parser 所得到的結果做個比對。每一個測試檔的人工結果在比對前都至少確認過兩次，因爲要確保人工結果是正確的，才能和 GLR parser 的結果作比較。

在比對過全部 100 個測試檔的人工執行結果和 GLR parser 執行結果都是相同的，所以我們 GLR parser 具有一定的堅固性(robust)，但若能做更多的測試則更能確保此程式的堅固性，所以在未來可以再做更多的測試，確保此 GLR Parser 可以準確的紀錄所有的剖析過程中的可能性。

4.3 實驗二：指令選取的比較

在此實驗，我們要比較的是以 BURS 爲基礎的指令選取機制和以 GLR Parser 爲基礎的指令選取機制能否找出相同的解。

4.3.1 實驗設計

在將我們 GLR Parser 整合到 CVM 的即時編譯器之後，我們就挑選幾個測試檔來做實驗。一個 Java 原始程式轉成 bytecodes 再轉成 Intermediate Representation 時，其實 Intermediate Representation 是由一連串的樹所組成的，每個樹的大小不一，而且多數的樹在交由我們 GLR Parser 依照即時編譯器的指令文法去剖析時並不會讓剖析過程中有兩個以上的剖析堆疊的存在，如此的 IR tree 就不能被我們 GLR Parser 所使用。因此，爲了讓我們 GLR Parser 在處理某個樹所轉成的字串之後可以讓 Graph-Structure Stack 中有許多的剖析堆疊存在，我們先研究了即時編譯器的指令文法，找出是哪些規則的什麼特色造成了 shift/reduce conflict 或 reduce/reduce conflict 的情況，以方便我們設計 Java 程式。

在我們所使用的即時編譯器的指令文法中，共有 195 條規則，而在交由 NewBison 當作輸入後，可發現共含有 155 個 shift/reduce conflict 和 1587 個 reduce/reduce conflict，而很多衝突都是發生在一個稱作 ICONST_32 的 terminal 身上，因此當我們 GLR Parser 依照此模糊的指令文法去剖析一個由 IR tree 轉成的字串時，要能在剖析過程中有多個剖析堆疊存在，則該 IR tree 必須含有多個 ICONST_32 節點。而爲了要得到這樣的 Intermediate Representation，我們在設計 Java 原始程式，就必須含有許多的常數運算。

Table 4.2 列出了挑選出來會交由我們 GLR Parser 來處理的測試檔，每個測試檔是從其相關的 Java 原始程式碼所轉成的樹狀 Intermediate Representation 的一連串樹中選出的一個樹，而這個樹能夠讓我們 GLR Parser 在處理時會有多個剖析堆疊存在。第一個測試檔和第二個測試檔都是來自由 Java 程式碼所撰寫的矩陣乘法，所處理的兩個矩陣是 16*16 大小。第一個測試檔僅有偏少的 12 個節點，但仍然拿來做測試，主要是因爲 GLR Parser 在剖析它的過程中會遭遇到 shift/reduce conflict，而第二個測試檔則是我們找到具有最多節點的一個 Intermediate Representation，GLR Parser 若剖析它會遭遇到最多個數的 reduce/reduce conflict。第三個測試檔則是來自最小擴張樹演算法中的 Kruskal 演算法，所使用的圖有 13 個節點，21 個邊。第四個測試檔則是來自最大訪客數演算法，計算有 21 位訪客的情形。第五個測試檔則是大數乘法，計算兩個 7 位數字的乘積。

	Number of IR tree nodes	Number of S/R conflict	Number of R/R conflict	Number of parse stacks in the GSS
Test Case 1	12	1	0	2
Test Case 2	63	0	6	64
Test Case 3	30	1	3	16
Test Case 4	26	0	2	4
Test Case 5	31	1	1	4

Table 4.2 由不同 Java 程式的找到的 IR 測試檔

在設計好含有許多常數運算的 Java 原始程式，然後取得其中我們所需的測試檔後，我們就可以比較以 GLR Parser 和原本的 BURS 技術來作指令選取工作的情況。

4.3.2 實驗結果

我們將上述提到的四個 Java 程式：矩陣乘法、最小擴張樹演算法中的 Kruskal 演算法、最大訪客數演算法和大數乘法分別當作 CVM 的輸入，然後觀察在兩個不同的指令選取機制下，每個測試檔的 IR tree 的情形。我們的 CVM 後端平台是 RISC 架構的 Andes 平台，因此我們就是針對這兩個機制來去觀察五個測試檔的 Andes 指令輸出結果。

而從結果來看可發現這兩個不同的指令選取機制去運作每個測試檔，而每個測試檔最終的最佳指令序列都是相同的，所產生的原生機器指令也都是相同的。而在深入探究兩個演算法之後，發現他們都會針對所傳進來的 IR tree 使用由下而上的剖析順序去求出其中所有可能的結果，也都使用相同的順序去比對該 IR tree，然後再利用成本計算方式找出成本最小的剖析方式，因此兩者在針對 IR tree 的剖析結果其實都會相同。亦即，使用 GLR Parser 當作指令選取器是可以找到最佳的指令序列。

在分析比較兩者的演算法後，可以知道兩者都可以找到相同的最佳指令序列，但若真要將 GLR Parser 應用在指令選取上，則有一些困難待克服。第一點就是 GLR Parser 在執行剖析 Intermediate Representation 時，會需要大量的運算，最終才能找出最佳的指令序列，而 BURS 卻是將許多的計算放在建立 tree automaton 表格時，因此在執行剖析 Intermediate Representation 只需要查表格就可以找出最佳的解。因此若不能減少 GLR Parser 在剖析時所需的計算，則指令選取過程所需的時間會比 BURS 來得多。能夠減少 GLR Parser 執行剖析時間的一個方式就是減輕其 Graph-Structure Stack 的負擔。第二點要克服的困難在於 GLR Parser 目前僅能剖析由 IR tree 轉成的 IR 字串，卻無法剖析由 IR DAG 轉成的 IR 字串，因此剖析後的結果會有重複的指令出現。

4.4 實驗三：測試 GLR parser 剖析時間

在此實驗，我們要測量的是在有使用或沒使用解決 shift/reduce conflict 和 reduce/reduce conflict 技術的情況下，GLR parser 真正執行剖析時所花的時間，因此這部份的時間計算並未包含解決 shift/reduce conflict 演算法所花費的時間，如 Figure 4.3 所示。

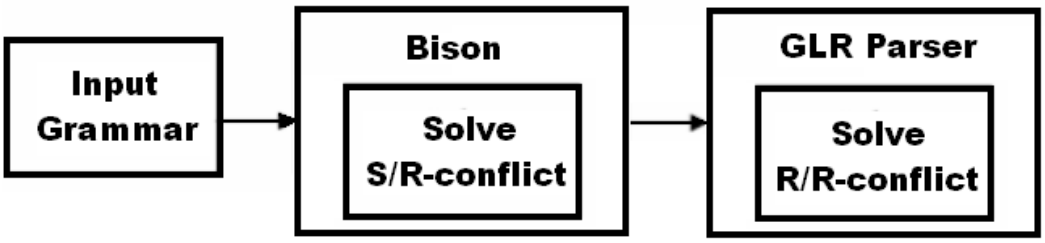


Figure 4.4 衝突解決技術執行的位置

4.4.1 實驗設計

一個 Java 原始程式轉成 bytecodes 再轉成 IR 時，其實 IR 是由一連串的樹所組成的，每個樹的大小不一，而且多數的樹在交由我們 GLR parser 依照即時編譯器的指令文法去剖析時並不會讓剖析過程中有兩個以上的剖析堆疊的存在，如此的 IR tree 就不能被我們 GLR parser 所使用。因此，爲了讓我們 GLR parser 在處理某個樹所轉成的字串之後可以讓 GSS 中有許多的剖析堆疊存在，我們先研究了即時編譯器的指令文法，找出是哪些規則的什麼特色造成了 shift/reduce conflict 或 reduce/reduce conflict 的情況，以方便我們設計 Java 程式。

在我們所使用的即時編譯器的指令文法中，共有 195 條規則，而在交由 Bison 當作輸入後，可發現共含有 163 個 shift/reduce 衝突和 1587 個 reduce/reduce 衝突，而很多衝突都是發生在一個稱作 ICONST_32 的 terminal 身上，因此當我們 GLR parser 依照此模糊的指令文法去剖析一個由 IR tree 轉成的字串時，要能在剖析過程中有多個剖析堆疊存在，則該 IR tree 必須含有多個 ICONST_32 節點。而爲了要得到這樣的 IR，我們在設計 Java 原始程式時，就必須含有許多的常數運算。Table 4.3 列出了 JIT grammar 的資訊，其中包含了在 JIT grammar 中，多少比例的 shift/reduce conflict 和多少比例的 reduce/reduce conflict 能夠被處理，分別是 1.23%和 59.17%

Rules	195
Terminals	143
Non-terminals	19
FSM states	427
S/R conflicts	163
R/R conflicts	1587

Table 4.3 JIT grammar 的資訊

在設計好含有許多常數運算的 Java 原始程式，然後取得其中我們所需的測試檔後，接著就將指令文法 JIT grammar 交由 NewBison 來處理，然後以產生出來的 GLR parser 來去剖析這些測試檔，以觀察 GLR parser 剖析完整個測試檔的目標句子會花多少時間。

實驗的電腦是安裝 Ubuntu 的作業系統，然後 CPU 是 Intel 3.4GHz，共 2GB 的記憶體。第一個測試的是沒有任何衝突解決技術的情況，然後因爲在所有測試檔的衝突中，reduce/reduce conflict 佔很大的部分，所以第二個要測試的是在只有執行解決 reduce/reduce conflict 技術的情況，第三個要測試的則是兩種衝突技術都有使用的情況，第四個要測試的則是兩種解決 conflict 技術都有使用的情況。

4.4.2 實驗結果

Table 4.4 顯示了所測試的結果，單位是毫秒(millisecond)，而每一次的測試都是執行 30 次再求平均數，括弧中的百分比則是顯示剖析時間減少的百分比。

	Total Time	Total Time with R/R technique	Total Time with S/R technique	Total Time with R/R and S/R technique
Test Case 1	**29.57**	**No R/R conflict**	**18.43 (37.67%)**	**No R/R conflict**
Test Case 2	**917.3**	**604.97 (34.04%)**	**No S/R conflict**	**No S/R conflict**
Test Case 3	**152.63**	**115.6 (24.26%)**	**132.37 (13.27%)**	**98.17 (35.68%)**
Test Case 4	**99.17**	**63.7 (35.76%)**	**No S/R conflict**	**No S/R conflict**
Test Case 5	**108.46**	**91.1 (16%)**	**86.93 (21.53%)**	**67.27 (37.97%)**

Table 4.4 GLR parser 剖析時間(ms)

第一個測試檔由我們 GLR parser 來處理，僅會遭遇到 1 次 shift/reduce conflict，因此在剖析過程中共會有兩個剖析堆疊的存在。而因爲僅有 shift/reduce conflict 的存在，因此僅使用了解決 shift/reduce conflict 的技術，而未使用解決 reduce/reduce conflict 的技術。最終剖析所需的時間平均來說會減少 37.67%。第二個測試檔雖然不會遭遇到任何的 shift/reduce conflict，但會遭遇到 6 次 reduce/reduce conflict，所以在剖析過程中共會有 64 個剖析堆疊的存在。因爲僅有 reduce/reduce conflict 的存在，因此僅使用了解決 reduce/reduce conflict 的技術，而剖析所需的時間平均來說可以減少 34.04%。第三個測試檔會遭遇到 1 次 shift/reduce conflict 和 3 次 reduce/reduce conflict，所以在剖析過程中共會有 16 個剖析堆疊的存在。在使用了解決 reduce/reduce conflict 的技術後，剖析時間會減少 24.26%，而使用解決 shift/reduce conflict 的技術則執行時間會減少 13..27%。若兩個衝突解決的技術同時使用，則剖析時間會減少 35.68%。第四個測試檔會遭遇到 2 次 reduce/reduce conflict，所以在剖析過程中共會有 4 個剖析堆疊的存在，而在使用了解決 reduce/reduce conflict 的技術後，執行時間會減少 35.76%。第五個測試檔會遭遇到 1 次 shift/reduce conflict 和 1 次 reduce/reduce conflict，所以在剖析過程中共會有 4 個剖析堆疊的存在。在使用了解決 reduce/reduce conflict 的技術後，執行時間會減少 16%，而使用解決 shift/reduce conflict 的技術則執行時間會減少 21..53%。若兩個衝突解決的技術同時使用，則剖析時間會減少 37.97%。

從我們測試的結果可以看出，使用解決 shift/reduce conflict 和 reduce/reduce conflict 的技術確實可以減少我們 GLR parser 在剖析一個目標句子時所需要的時間，但若能有更大的測試檔且含有更多的衝突，則可以做更精確的測試。雖然我們所使用的指令文法儘管有 163 個 shift/reduce conflict 和 1587 個 reduce/reduce conflict，但真正造成前者衝突的規則僅有幾條而已，造成後者衝突的規則則較多，因此在找尋測試檔的過程中，比較不容易找到含有 shift/reduce conflict 的目標句子，至於所找到的測試檔最多也僅有 63 個，主要是因爲一個 Java 原始檔會轉成一連串的中間表示樹，但大部份的樹都不會遭遇到任何的衝突，只有少數的樹會遭遇到衝途，因此在未來希望能測試更多的 Java 原始檔來找到更大的測試檔。

這兩個衝突解決的技術雖然能確實減少剖析的時間，且儘管這五個測試檔中的每個衝突都順利被解決，但就整體來看某些衝突仍然無法解決，因此這兩個技術還有許多改進的空間。

第五章 結論

在本篇論文中，我們使用文法剖析的方式來進行 CVM 即時編譯器後端的指令選取工作，而在實驗的過程中跟原本的 BURS 指令選取機制相比之後，發現兩者其實都會針對所要處理的目標去求出其中所有可能的結果，然後再找出成本最小的，因此兩者都可以找出相同且最佳的指令序列。因此，GLR Parser 是可以應用在指令選取上，但需要一個好的方法來解決執行時間較多的問題。

減少執行時間的一個方式就是減輕 Graph-Structure Stack 負擔，亦即，要能解決 shift/reduce conflict 和 reduce/reduce conflict。而爲了配合 GLR Parser 在剖析過程中能使用成本機制來找出最佳解，解決衝突的方式也要確保不會影響到最終的結果，因此使用成本機制來解決衝突。解決前者衝突的技術是在建立自動狀態機表格時就處理好，因此可以減輕剖析時的負擔，但由於該演算法是針對文法中的相關規則去比較他們右半部元素的長相，能解決的衝突類型就受到限制，而後者衝突的解決技術更是在剖析時才能處理，因此在應用成本資訊去解決這些衝突的技術還有值得的改進空間。

儘管本篇論文有許多尚待改進之處，但我們將使用文法剖析方式來做指令選取工作所會碰到的困難呈現出來，以期能當作將來後續研究的一個基礎，進而能更加擴充 GLR Parser 在指令選取上的應用。

參考文獻

[1].Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tijiang. “Code Generation Using Tree Pattern Matching and Dynamic Programming”, ACM Transactions on Programming Languages and Systems, 1989, 491-516.

[2].Alfred V. Aho, and Margaret J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, Communications of the ACM archive. Volume 18 , Issue 6, 1975, 333-340.

[3].David R. Chase, “An Improvement to Bottom-Up Tree Pattern Matching”, In Proceedings of the l4th Annual Symposium on Principles of Programming Languages, 1987, 168–177.

[4] M. Anton Ertl, “Optimal Code Selection in Dags”, In Proceedings of the 26th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, 1999, 242–249.

[5].Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting, “Engineering a Simple, Efficient Code-Generator Generator”, ACM Letters on Program Language and Systems, 1992, 213-226.

[6].Christopher W. Fraser and Robert R. Henry, “Hard-Coding Bottom-Up Code Generation Tables to Save Time and Space”, Software—Practice and Experience, 1991, 1–12.

[7].Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting, “BURG — Fast Optimal Instruction Selection and Tree Parsing”, Technical Report 1066, University of Wisconsin, 1991.

[8].Mahadevan Ganapathi, and Charles N. Fischer, “Affix Grammar Driven Code Generation”, ACM Transactions on Programming Languages and Systems, 1985, 560-599.

[9].Mahadevan Ganapathi, and Charles N. Fischer, “Description-Driven Code Generation Using Attribute Grammars”, ACM, 1982, 108-119

[10].Philip J. Hatcher and Thomas W. Christopher, “High Quality Code Generation Via Bottom-Up Tree Pattern Matching”, In Conference Record of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, 1986, 119-130.

[11].Robert Henry, “Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree-Pattern Matcher”, Technical Report 89-02-04, Computer Science Department, University of Washington, 1989.

[12].M. Jazayeri, K.G. Walter, “Alternating Semantic Evaluators”, In Proceedings of the ACM Annual Conference, 1975, 230-234.

[13].Ken Kennedy, Scott K. Warren, “Automatic Generation of Efficient Evaluators

for Attribute Grammars”, 3rd Annual ACM Symposium Principles of Programming Languages, 1976, 32-49.

[14].Maya Madhavan, Priti Shankar, Siddhartha Rai, and U.Ramakrishna, “Extending Graham-Glanville Techniques for Optimal Code Generation”, ACM Transactions on Programming Languages and Systems, 2000, 973-1001.

[15].Maya Madhavan, “Optimal Linear Regular Tree Pattern Matching Using Pushdown Automata”, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 1998.

[16].S.McPeak and G. C. Necula. “Elkhound: A Fast,Practical GLR parser Generator”, Proceedings of Conference on Compiler Construction, pages 73-88,2004.

[17].P. Marwedel and G. Goossens, “Code Generation for Embedded Processors”, Kluwer, 1995.

[18].Steven Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers, 1997.

[19].Todd A. Proebsting, Benjamin R.Whaley, “One-Pass, Optimal Tree Parsing-With Or Without Trees”, Proceedings of the 6th International Conference on Compiler Construction, 1996, 294-308.

[20].Masaru Tomita, “Graph-Structured Stack and Natural Language Parsing”, In 26th Annual Meeting of the Association for Computational Linguistics, 1988, 249–257.

[21].Eduardo Pelegri-Llopart, "Rewrite Systems, Paitern Matching, and Code Generation", Phd Thesis, Technical Report UCB/CSD 88/423, Computer Science Division, University of California, Berkeley, 1988.

[22].Eduardo Pelegri-Llopart and Susan L. Graham. "Optimal Code Generation for Expression Trees: An Application of BURS Theory", In Proceedings of the 15th Annual Symposium on Principles of Programming Languages, 1988, 294-308.

[23].Todd A. Proebsting, “Simple and Efficient BURS Table Generation”, In Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation ACM, 1992, 331–340.

[24].Sun Microsystems. “CDC HotSpot Implementation Dynamic Compiler Architecture Guide”, 2005.

[25].Beatrix Weisgerber, and Reinhard Wilhelm, “Two Tree Pattern Matchers for Code Selection”, In Compiler compilers and high speed compilation, LNCS 371, 1988, 215-229.

[26].Wuu Yang, “A Fast General Parser for Automatic Code”, 2nd Russia-Taiwan Symposium on Methods and Tools of Parallel Programming Multicomputers (MTPP 2010), Vladivostok, Russia, Lecture Notes in Computer Science, Vol. 6803, May 16-19, 2010. (Indexed by EI, DBLP, INSPEC, Thomson)