MyLisp 公理與標準

版本(Ver0.3)

目錄

修訂記錄

時間	操作	編輯者	備註
2014.4.2	創建	葉樹深	起草求值器相關內容

2014.4.6	添加	葉樹深	增加棧相關內容
2014.4.12	修正	葉樹深	修正棧相關內容
2014.4.18	修正	葉樹深	增加 RefLisp 輔助求值器
2014.4.20	添加	葉樹深	添加反引用求值器，；
2014.4.22	添加 修正	葉樹深	添加 sets 棧相關內容修正 setg 棧相關內容 增加 1 種原子求值公理
2014.4.25	添加	葉樹深	添加 return 求值器
2014.4.27	添加	葉樹深	增加 DebugLisp 原子資訊日誌輸入協助工具
2014.5.2	添加	葉樹深	增加一種反引用求值器-隱式引用反引用’ 增加簡單數值計算加 乘 模除
2014.5.4	添加	葉樹深	添加中斷點調試求值器
2014.5.6	增加 修正	葉樹深	添加修正棧引用求值器 StackRefLispGetVal
2014.5.8	添加	葉樹深	添加 Lisp 物件使用權限控制 seta
2014.5.10	添加	葉樹深	添加列表向上融合求值器@ 支援反引用環境 cons list 等列表組裝求值環境中。
2014.5.12	添加	葉樹深	添加 REM 注釋求值器 添加 LessThan <數值判斷求值器
2014.5.14	修訂	葉樹深	添加細節內容，修整錯誤

前言

做了十年 C C++碼農，一直痛苦的掙紮著，我手的動作很慢，代碼編輯器都要適應很久才能簡單用起來，linux 更是不敢玩，寫著很多的看似重複又有區別的代碼，10 年前還能每天手寫千行代碼調試起來的我，現在每天可能寫不出百行那樣的代碼了，因為我老了，所以這十年工作中寫了很多範本工具，巨集工具來簡化自己的工作，比如序列化，隱藏細節的形式代碼，然後還搞 XML 來動態描述業務，用工具來設計業務生成 XML，讓自己不斷的去挑戰完成碼農的極限任務，但是這中間發現用範本用巨集是那樣的有難度又不便，致使寫過的複雜範本和巨集幾乎都不能再維護起來，XML 和 C++結合更是帶來了新的痛苦，是否存在一種靜態的 XML C++介面來以不變應萬變適應不同的應用設計，這是減少工作的重點，C++解析 XML 樹還是有很多認為的錯誤在其中影響工作量的，並且 XML 哲學上講並不是萬能，他給 C++代碼帶來運行期動態能力的同時，卻帶來了新的靜態語言問題，因為 XML 本身容易做靜態業務描述，但是本身不能動態運行邏輯做業務邏輯推理之類的工作，後來我為了簡化 XML C++之間的工作，無奈設計了一種積木式的代碼模式識別合成器來根據靜態代碼結構做代碼生成，比如生成 XML C++物件的映射序列化代碼，生成伺服器後臺物件介面代碼，隱藏 SQL C++異構平臺交互代碼設計工作細節等，設計過程看似順其自然，並且也幫我生成了幾十萬行穩定的代碼，但是之後卻發現代碼識別合成的代碼的複用能力極差，用於描述靜態系統可以，但是動態的生成執行並分析變化的描述能力就很差。

後來學習易經後發現，這個矛盾的過程是必然的且生生不息的需要去做的事情，形而上學的靜態描述語言是成功的，因為描述一個事物是簡單的，並且不需要全面的，也不需要關心描述中的事物具體要去怎麼做什麼時間做什麼地點做，什麼代碼中做，什麼執行緒做，試錯後增加改變描述是方便的，但他同時也是失敗的，因為形而上學的東西實作過程中並不告訴你實現描述的系統應該是什麼樣子的，邏輯是如何實現的，並且靜態描述語言有一個大的問題就是描述不能根據動態變化的需求而動態變化，動態變化意味著必須有可運行的邏輯。

與此同時可以有動態邏輯的語言是成功的，但他同時他也是失敗的，成功是因為有可控制的運行邏輯，可以在確定的空間時間執行緒中去精確的完成一件任何你能描述出來的事情，失敗則是因為他太缺乏形而上學能力脫離空間時間執行緒的去簡單快速的構建我們需要的目標。

所以為了統籌並充分合理的應用矛與盾是我們必然要去面對的事情，怎樣用靜態描述目標需求來簡化工作，怎樣用動態可執行的精確邏輯來實現靜態描述，怎樣用靜態描述來代替動態可執行邏輯的細節複雜性，怎樣用動態可執行邏輯來使靜態描述擁有動態變化適應目標的能力，這將是我們一直以來生生不息的去碼農的事情，而我找到了為這個目標而生的 MyLisp。

2014.5.14

------葉樹深

MyLisp 基本哲學

MyLisp 基本哲學是易經或者說是馬哲矛盾論，通常會堅持認為是易經，因為易經是空洞的馬哲矛盾論充實形而上學內容後的相對完整版本，最古老的 Lisp 中我們經常會看到一個類似易經符號的形象的 Lisp 原理符號，但是那個符號太具型了，描述具體的 Lisp 求值矛盾可以很形象，但是缺乏泛型抽象的形而上學能力，致使其不能被覆用在其他領域，另外 Lisp 基本公理中本身並不完全符合易經的泛型演算法，其只能算是易經思想的一個範圍受限的子應用情節，Lisp 本身只描述了矛盾事物的雙方，但是雙方變化的結果與目標卻沒有顯式的提出，原子還是原子，列表還是列表，矛與盾之間的關係如何統籌，如何相互利用，相互變化，則很難體會，如果說列表用於描述一個事物，那麼列表的第一個原子就用於求值這個事物，但是求值方法各色各異，我們是不是能語言原生的提供所有求值器？如果語言不原生的提供這個求值器原子，那麼這個語言是不是就是一片死水，毫無可再生設計能力，解決這個問題的關鍵在於列表的第一個原子要誰去構造設計，如果用易經原理，那麼矛是可以被盾設計構造並形成新的類型的（64 卦就是這麼自指反覆運算來的），那麼我們借鑒這個思想用列表去設計構造原子，那麼問題就解決了，但同時出來的問題是類型變的更多了，一個原子是個列表，這個列表是個可執行的求值器，另一個場景一個原子是一個列表，但是這個列表只是用來描述事物，而不執行，這種複合模式使得泛型的兩個類型可以生生不息構造出各種各樣的其他類型。

經常有人會對比 Lisp 和 C 之類的，說些不符合易經哲學原理的目標，比如 Lisp 一定要編譯出來，一定要靜態化，一定要多類型為不同場合做優化，這樣才能跑的像 C 一樣快，夢想是美麗的，現實是骨感的，矛盾就是在折騰你的耐心，你在獲取了絕對矛的時候，就丟失了盾的能力，MyLisp 不追求絕對的矛盾，有取有舍，有時會犧牲矛換來更多的矛盾容量，但反過來會用更多的矛盾在其他領域去做更好的矛。目標是矛，目標是盾，需要舍去矛，需要舍去盾，一切按照環境目標需求去做，而過程未必是直接去做，更多的時候可能是南轅北轍的去曲線救國，因為直接救國的方法技術是矛盾變化糾纏的不存在。

很多人重結果而輕過程，總會問結果是多少，你把結果給我算出來，但顯然更多的結果只是一個變化的過程，並且是矛盾的存在著的事實，而靜態的結果意味要求的結果可能會使另外的要求結果變的很糟，就像大多數老闆總喜歡說你快點把事情做出來，不管你用什麼辦法，只要最快的就好，結果事情做出來了，需求卻發生了變化，任何人沒準備的情況下再重新來一遍不同的過程吧，結果重新來一遍後發現事情不如第一次的那個好，這咋辦呢，過程被一遍一遍的走過。問題的關鍵在哪？我們要的其實是一個 X，是 1 不是 1，是 2 不是 2，X 代表的是過程，而非靜態結果，這是易經符號所描述的事實，在現實中得到一次又一次的體

現。一個變化的過程攜帶的是無窮的甚至矛盾的結果，所以我們要的其實是能不斷的創造過程，複用過程，修改過程，刪除過程，輸出一個過程式的結果。而且最好人不要去做這種事情，這種事情如果機器能做最好機器做吧，很多語言中描述的lambda 延遲求值即這個思想，但是是不是真正的做到了上面說的這些，就很難講了。

像易經像化學一樣的為過程而設計，易經符號沒有結果只有一個動態矛盾過程的變化描述，可以類比為化學式，一般的化學式等式兩邊都是互相轉化的變化的存在著達到一個相對平衡，正反應負反應都在同時進行著，這描述的是一種矛盾的需求和結果，我們很多時候設計的軟體也類似這種模式，需求中存在矛盾，我們根本不能用一種具體方法去簡單的用一種明確模式去做，所以我們應該把一個能包容矛盾的又清晰合理的過程設計出來交付給用戶，而不是某一個靜態解，所以 DSL 程式設計就是為這種矛盾的需求而生，大部分功能可以清晰簡單明確的讓用戶可以去二次開發出滿足矛和盾應用的場合。

文本文檔只發行繁體字版本，這是因為繁體字本身攜帶更多形而上學內容，有利於鍛煉讀者的形而上學抽象能力。

求值 MyLisp 目標

第1节.需求分析

MyLisp 創作之初是因為作者本人學習易經過程中得到的啟發，之後根據易經的基本哲學原理要求，在電腦已有流行語言中尋找最容易表達易經思想的語言，經過學習分類各種語言後篩選出一個叫 Lisp 的語言，對比學習 Lisp 的主要的兩個版本 Scheme 和 Common Lisp 後，發現這兩個版本基本都不符合自己的使用習慣，下面是這兩種語言作者認為不好的方面。

1. Scheme 的問題
    1. 看 Scheme 幾乎搞不清楚什麼是函數，什麼是巨集，而這兩個概念對於電腦程式設計語言來講是很重要的兩個概念。
    2. 我到現在也不知道他的符號空間是如何的確定的，規則與方法未知。
    3. Scheme 雖然號稱要簡單，但是原生的非易經(Lisp)基本原理的靜態類型規則內容較多。

2. Common Lisp 問題
    1. Common Lisp 則規模龐大，底層代碼比如 defun 之類的函數代碼因為為了避免名稱空間問題，所以命名很晦澀，比彙編還難看，甚至為瞭解決符號衝突而是用隨機符號生成，這樣調試也難調試。
    2. Common Lisp 是編譯執行，所以這個過程必然將一些動態語言特性靜態化，比如巨集就很怪異，對我來講巨集就是巨集，他的目標是生成代碼，而非執行代碼，但是 Common Lisp 中的巨集看起來像函數一樣求值出了結果，這樣就不利於代碼的再分析變形以及推理，這使得 DSL 設計功能被弱化。

Scheme 和 Common Lisp 共同的問題是原生語言的概念類型模式等內容都太多而讓人不能從根本上自由的開始使用這兩個語言，有很多類型比如調用函數物件要用#`之類，lambda

之類似乎都是靜態內容而不能再次被設計,各種資料結構 vector string 要做類型轉換，感覺這兩個語言都在希望把自己做成 C 語言,所以丟失了最初 lisp 的簡潔優美的泛型資料結構的優勢， MyLisp 認為這些確定的資料類型不應該集成在語言原生支援中，而應該用 Lisp 基本泛型求值原理在上層的 DSL 中去可定制的自由實現,這樣就可以根據需要靈活制定出各種符合目標系統的描述語義。

當然按照易經哲學原理來看上面分析的 Scheme 和 Common Lisp 的失敗則是他們另一個領域的成功，那就是靜態化一些概念和類型後，那麼就可以做編譯器靜態優化，運行時速度就會很快，並且強大而多的概念和功能就可以短期內做到適配各種目標的需要功能效果。對於 Lisp 來講，不計算出結果的宏是非常重要的元程式設計手段，比如比如求斐波那契，一般人喜歡用 C 的方法直接寫代碼來求出結果，但是元程式設計不同 fib(4)函數求出來的數值是 5，而元程式設計的目標結果是生成這樣的求值代碼 (+ (+1 1) (+ 1 (+1 1))),這樣做的原因是 5 是不能被推理的結果,而 (+ (+1 1) (+ 1 (+ 1 1)))是可以推理再變形從而降低複雜度的設計過程，這個過程是有意義的，因為它意味這一些重要技術比如延遲計算技術將可能實現，比如我要計算 π 並且是 100000 個 π 相加，並且這個過程是動態形成的而非事先能靜態化的，那麼你的求值過程可能會是這樣(+ Get(π) Get(π) …….. Get(π)),那麼用元程式設計思路設計來做延遲求值的話，就是不求值計算(+ Get(π) Get(π) …….. Get(π))，而是將這個代碼推理求值為(* 1000000 Get(π)),然後再求值那麼複雜度將降低到百萬分之一，並且不會因為值而動態增長規模。

第2节. MyLisp 需求定義

經過上述分析，設計定義 MyLisp 需求如下：

MyLisp 主要面向設計代碼的過程，而非編寫求出計算目標結果的代碼， MyLisp 主要做元程式設計，代碼生成器，DSL(自然語言) 以及公式推理，所以看了 MyLisp 需求之後，希望不要再有人問函數和巨集有啥區別，我要編寫個代碼，誰更快，誰更簡單的問題，MyLisp 的目標不是像 C 一樣運行,但卻可以是設計 C 代碼的語言，你甚至可以去實現個 UML 代碼生成器去做 C 代碼，MyLisp 為了達到元程式設計的目標，很多時候會南轅北轍的不惜降低 1000 倍速度的去做一些看起來很簡單的事情。

第3节. MyLisp 設計目標與原則

經過上面分析結合自己的需要定義 MyLisp 的設計目標如下：

1. MyLisp 原生堅持易經基本原則，即簡單構造美，使用最簡化的泛型易經資料結構，即原子與清單，將其他需要的特定領域的資料類型比如函數，巨集,lambda,class 謂詞等內容全部用這種基本資料結構在 DSL 中去表示和實現,在最終需要一個靜態結果時再去真正的解類型（動態多態，延遲求值），如果不滿足速度，則用代碼生成器的方式去構造目標系統代碼。

2. MyLisp 借用易經哲學修改了 Lisp 基本公理中的概念，使得原子和列表不再是兩個極端對立而不能簡單統籌協作的矛盾。原子可以是列表的引用，列表的求值結果可以是一個原子，並彼此嵌套應用，列表中的原子還可以是一個列表，列表中的列表求值後也可以是一個原子，而該原子又可能是個列表引用，矛和盾在求值變化中彼此借用對方來達到自己的目的，並且很多時候自己變成了對方。

3. MyLisp 使用棧概念來明確原子符號的空間範圍以及轉移方法，將 Scheme 和 Common Lisp 中隱藏的詭異的符號概念完全揪出來考慮對待，之後再用 DSL 的方法去隱藏符號空間的細節問題。

4. MyLisp 使用解釋執行方法，儘量將所有事情交給運行期執行，包括原子符號空間的確定，代碼的合成，棧的運行期自動按需變化（比如一個函數棧內執行一個參數代碼，那麼參數代碼的內容將動態決定棧上的原子數量內容），這樣的直接結果是 MyLisp 將很緩慢的做事情，目前棧的速度為 C 棧的 1/1000,將來如果擴展運行期設計能力（代碼生成）有可能進一步提升達到 1/10000,以包容更多的運行期可設計的編譯器或者代碼生成器，MyLisp 暫時不會考慮編譯執行，因為那樣會帶來可怕的靜態化，不過倒是可能加入一些併發執行，執行緒池之類的概念，做些併發的 MyLisp 虛擬機器，比如用 OpenCL 技術去設計併發虛擬機器。

5. MyLisp 程式設計命名方法借鑒易經哲學，包容各種命名格式以及母語語義，甚至完全中文程式設計也是可以的，因為按照易經哲學原則，沒有絕對的最好的命名格式，多數時候缺陷是一種美，是一種成功的方法，這是設計 DSL 的基本哲學思想，因為不能每種語言格式都是存在的必要，比如 C 有強制的符號式語法格式，所以他能在全球範圍內成為所有人共同能理解的語言，而 C++的範本程式設計巨集程式設計或者類似 python 那種英文語法式程式設計，這種類型我們都可以稱之為母語程式設計，母語程式設計是優秀的同時也是失敗的，優秀是因為用母語隱藏了很多準確的符號語言細節，你可以很快速的達到目標，但對於不熟悉你的母語以及你的社會環境的人來講這將是困惑失敗的開始，同時母語程式設計是需要社交環境來傳播維持的，你如果沒有那個環境不能進入那個圈子，只是靠讀書那就很難學會母語語言。

第一部分MyLisp 基礎篇

第一章. MyLisp 泛型求值原理

第1节.基本概念

MyLisp 只有兩種泛型的資料結構，即原子和清單，其他 DSL 擴展的類型都用這兩種資料結果表示，原子和列表有相互求值轉化並描述的能力，原子可以是一個列表的引用，對原子求值就是對列表求值，列表可以是被原子引到其他列表中去，列表由原子組成，但原子可

能是列表，所以也可以說列表是由列表組成。

第2节.原子求值

1. 如果一個原子，其值為列表，則對該原子求值就是對該列表的求值。
2. 如果一個原子，其值為 T NIL 字串 或者一個原子符號，則返回一個符號名字為三種值的原子，這樣新生成的原子又能被賦值求值，產生新的資料代碼物件。
3. 如果一個原子，其符號名稱是基本公理中的函數原子，求值該原子，如果這個函數原子沒有提供原子求值器，則會求值失敗（一般的基本公理不支援原子求值器只提供列表求值器，暫時只有 StackRefLisp 會考慮做成原子求值器）
4. 如果一個原子，其沒有被賦值過，則對其求值會按照自動名稱綁定規則將其分配在棧上或者綁定到全功能變數名稱稱空間上，求值結果為自己的一個引用。

第3节.列表求值

1. 列表和 lisp 基本公理不同，第一個參數表示原子，該原子可以是一個公理原子，也可以是一個原子列表，也可以是個形式上的列表(形式上的列表，非原子列表)，
2. 如果是個形式列表則對列表遞迴求值直到返回值是一個原子（可能造成閉環）然後對該原子做列表求值。
3. 如果第一個參數是一個原子，該原子是一個公理原子，則用公理原子對引用列表求值。
4. 如果第一個參數是一個原子，該原子是一個列表的引用，則求值該列表作為引用列表的返回值，一般這個列表是一個棧，該棧會對引用列表做出求值。

第二章. MyLisp 棧

第1节.為什麼要用棧

棧是為了在遞迴以及多塊代碼複用過程中可控制的隔離原子符號而生，以便於簡化符號隔離以及提供複用符號的能力。比如如果你可以在棧內重新定義一個已有的原子求值器，而在棧上其他地方可以使用，或者如你意願為一個原子求值器起一個習慣的別名。

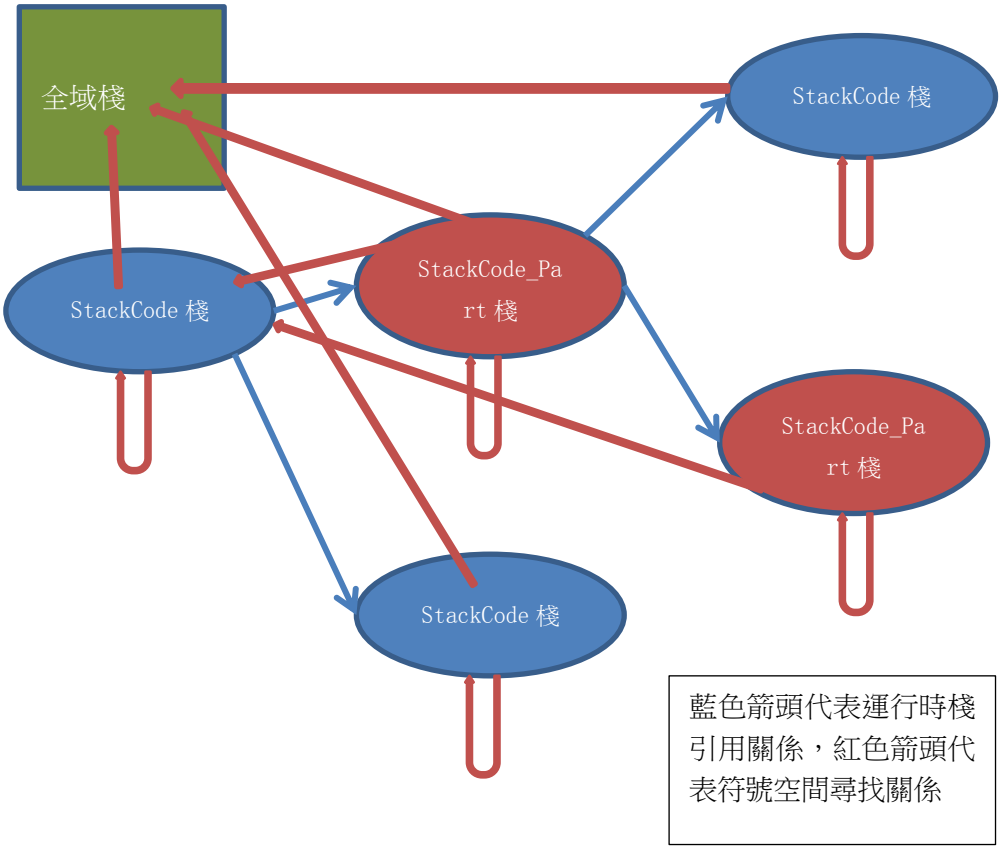
反復沖定義求值器從而達到形而上學效果，比如複數 向量 矩陣的計算。
(list (setg +) (setref + `Add) (+ 3 4 5) (setref + `Mul) (+ 3 4 5))
給自己覺著彆扭的求值器起一個自己喜歡的別名
(list (setg 設置引用) (setref 設置引用 `setref) (設置引用 a `(list 1 2 3)) a)

第2节.MyLisp 棧基本原理與類型

棧是用來運行時隔離原子符號定義的機制，目前將定義實現兩種棧來分別應對巨集以及函數的需要，函數需要完整的棧隔離機制，可以使用 StackCode 公理函數實現，巨集使用局部私有棧並共用其所在的 StackCode 棧，用 StackCode_Part 公理函數實現。

棧上的同名原子物件都只有一份棧拷貝，其他都以引用形式存在，棧上原子在沒有被求值之前，他是不存在於任何命名空間的，只有被真正第一次求值時，才會被確定其名稱空間。

第3节.棧上原子符號確認原則



1. StackCode 棧上原子的命名空間確認順序為 本地棧➔全功能變數名稱稱空間，如果最終都沒找到，那麼就將原子名稱空間確定為本地棧，如果找到則成為一個本地棧或全功能變數名稱稱空間對象的引用(基本公理函數物件都掛在全功能變數名稱稱空間上，所以如果你的原子名稱上是公理名稱，那麼用 setref 對其賦值，將會在棧上重載公理函數)。

2. StackCode_Part 的棧原子，其符號確認順序為 本地棧➔上層 StackCod 棧➔全域符

號空間，如果這個過程沒找到，那麼就將該原子名稱空間確定為本級棧。

第三章. 棧相關求值器

第1节.絕對隔離棧

(StackCode 代碼列表)
在棧上對代碼列表求值

第4节.部分隔離棧

(StackCode_Part 代碼列表)
在棧上對代碼清單求值,原子符號空間確定會回朔上層棧查詢直到一個 StackCode 類型棧。

注意 StackCode StackCode_Part 返回值去意留形，去除棧上 lisp 的內容形態，比如返回一個 lisp 物件 (原子 1 原子 2)，僅僅在語義上返回這個 lisp,如果原子 1 在本級棧上是個 lisp 物件（清單），那麼 lisp 物件將被去除，返回的原子 1 將僅僅是一個未初始化的原子，因為如果帶 lisp 值返回會將問題複雜化，很可能造成詭異的符號污染，後面考慮非 lisp 物件比如整數字串值的話可返回。

第5节.棧引用

(StackRefLisp n)或者原子 StackRefLisp
功能：用於得到引用棧運算式的引用或者其列表中指定原子
描述：如果是 StackRefLisp 原子，則返回一個棧的引用運算式引用，如果是清單方式求值則返回引用列表中指定位的原子，比如(StackRefLisp 0)返回引用棧第 0 個列表參數

(StackRefLispGetVal n)
和(StackRefLisp n)類似，不同的是 StackRefLispGetVal 返回的是在上層棧環境對傳址參數求值的結果引用，這樣就避免了在本棧需要對參數求值時造成的跨棧符號污染問題

第6节.棧返回

(return x)或者一個被求值的原子 return
功能：棧返回
描述：如果是列表(return x)，則對 x 求值，並返回 x 的求值結果，如果是原子 return 則直接返回棧，並不返回任何值，如果棧上沒有以 return 的形式返回，則棧默認會將棧代碼列表的求值結果返回。

第7节.聲明引用

(setref 列表或者原子 列表或者原子)
功能：聲明引用
描述：對參數 1 參數 2 求值（如果是物件引用則會解引用，也就是說不會有引用的引用存在），並將參數 1 的求值結果當做一個引用原子物件綁定到參數 2 的求值結果物件上，注意參數 1 的命名空間由其他求值器確定，如果沒確定則按照自動物件名稱綁定規則確定。

第8节.聲明全域原子

(setg 列表或者原子)
功能：將目標原子或求值的到的原子綁定創建到全功能變數名稱稱空間（已存在則綁定，不存在則創建）。
描述：參數 1 如果是列表（原子引用的列表，原子值是列表）則遞迴求值直到其為一個原子，如果是未初始化原子則不求值，將原子或者列表求值結果關聯或者創建到全功能變數名稱稱空間，生命期目前在程式結束時釋放

第9节.聲明棧內原子（本地，局部）

(sets 列表或者原子)
功能：將目標原子或求值的到的原子綁定創建到當前棧名稱空間（已存在則綁定，不存在則創建）。
描述：參數 1 如果是列表（原子引用的列表，原子值是列表）則遞迴求值直到其為一個原子，如果是未初始化原子則不求值，將原子或者列表求值結果關聯或者創建到當前棧名稱空間，該函數為運行期遞迴巨集做顯式棧隔離很有用處。

注意使用 setg sets 時，目標原子必須在之前沒有做過求值動作，或者被賦值為可求值為原子的列表或者列表引用，因為做過求值動作後，原子會在棧或者全域命名空間創建物件，如果目標原子已經確認過了名稱空間，則綁定或者創建原子將會失敗
關於引用於物件拷貝。

如果你的 Lisp 代碼物件需要棧式隔離多執行序訪問那麼應該用 setq 做物件拷貝，如果只是執行緒函數內做資料賦值，那麼建議用物件引用拷貝 setref,因為這樣不做物件拷貝，所以效率較高。

第10节. Lisp 泛型對象賦值

(setq `q `(1 2 3)))
對參數 1 2 求值，對參數 1 遞迴求值直到其為一個原子，則為原子賦值為參數 2 的求值結果，如果原子是一個引用原子，則對引用的目標物件進行賦值。返回 T 或者 Nil（自動確認）

第11节. 配置 Lisp 物件屬性與許可權

設置 lisp 物件使用權限
(seta 原子或清單引用原子 ((靜態求值 t) (僅拷貝引用 nil) (禁止寫入 t)))
對原子或引用原子被求值後的目標物件設置物件操作許可權 t 為開啟，nil 關閉，可以僅僅為 1 個或者多個許可權的組合
靜態求值　　目標物件將不再被求值，僅僅引用其之前的求值結果
僅拷貝引用　物件不被拷貝，只是拷貝其引用，比如 setq list car cdr 之類的物件賦值拷貝動作
禁止寫入　　物件不能被 setq 賦值，對一個靜止寫入的物件賦值，將會造成一個求值失敗的錯誤。

第四章. 原子列表形而上學求值(引用並擴充 Lisp 基本公理）

第1节.列表(list)

(list `1 `1 `2 `4)
對多有參數求值並新生成一個由參數求值結果組成的列表。
通常還用來做程式邏輯的執行器，對每個子元素做求值。
需要注意的是 list 對引用原子的對待，配合@列表融合可以將引用的列表融合到本層。
(list (setref a `(1 2 3)) (list 1 a)) 求值結果為(T (1 a))
(list (setref a `(1 2 3)) (list 1 @a)) 求值結果為(T (1 1 2 3))

第2节.物件比較(eq)

對比兩個物件是否相同，現階段暫時並不完善。

第3节.列表取首(car)

第4节.列表取餘(cdr)

第5节.追加列表(cons)

第6节.條件求值(cond)

第7节.判斷原子(atom)

(atom x)
求值 x 並判斷其結果是原子還是列表

(atom `a) 求值結果 T
(atom `{}) 求值結果 T
(atom `{1}) 求值結果 NIL
注意 MyLisp 與 Lisp 不同的是原子是有值的，所以對於一個已經有值的原子，或者是一個清單引用的原子則根據原子的值或者引用來對待判斷。
比如
(list (setref a `(list 2 3 4)) (atom `a)) 求值結果 (T NIL)
(list (setq a `(list 2 3 4)) (atom `a)) 求值結果 (T NIL)
(list (setq a `b) (atom `a)) 求值結果 (T NIL)

通常 atom 與 eq 配合做泛型對象類型判斷

第8节.列表向上融合(@)

(list 1 @(list 1 2 3)) == (list 1 (ListFuse (list 1 2 3))) ==>(1 1 2 3)
清單融合物件求值後如果是列表將被融合到上層列表
@並不單獨存在使用，通常配合 list cons car cdr 來做清單物件的形而上學組裝演算法。

1. 列表中使用融合

(list (setq a `(list 1 2 3)) (setq b `(list 4 5 6)) (setq c `(list @a @b)) c)
求值結果為(T T T (1 2 3 4 5 6))

2. 反引用中使用融合

(list (setq a `(list 1 2 3)) (setq b `(list 4 5 6)) (setq c ``(;@a ;@b)) c)
求值結果為(T T T (1 2 3 4 5 6))

3. 融合列表定義

(list (setq a `@(list 1 2 3)) (setq b `@(list 4 5 6)) (setq c `(list a b)) c)
求值結果為(T T T (1 2 3 4 5 6))

4. 追加列表中使用列表融合，完全顛覆了 cons 本意

(list (setq a `(list 1 2 3)) (setq b `(list 4 5 6)) (setq c `(cons a b)) c)
求值結果(T T T ((1 2 3) 4 5 6))

(list (setq a `@(list 1 2 3)) (setq b `(list 4 5 6)) (setq c `(cons a b)) c)
求值結果為(T T T (1 2 3 4 5 6))

(list (setq a `(list 1 2 3)) (setq b `(list 4 5 6)) (setq c `(cons @a b)) c)
求值結果為(T T T (1 2 3 4 5 6))

第五章. 引用與反引用

第1节.為什麼要引用與反引用

引用與反引用是 Lisp 用來實現宏元程式設計能力的基本技術保障，其主要用於在運行時動態根據需要生成目標 Lisp 代碼,用巨集遞迴演算法可以生成高階嵌套的引用反引用代碼，多層高階的引用反引用通常意味著你的這段集中描述式的代碼內容將分佈在不同的計算地點與時機上，這樣就有利於用形而上學的靜態描述語言方法去描述時間空間分佈不連續且複雜的動態系統。

MyLisp 與 Common Lisp 不同的是 MyLisp 更加完善和強化了反引用能力，尤其是在多層高階反引用中使的每層引用反引用都具有相同完備的能力，而 Common Lisp 在只在第一層反引用過程中具備完整的代碼組裝能力，而在更多層反引用展開過程中失去動態代碼合成的能力。

第2节.引用(`)

引用可以認為是拒絕其子運算式的一次求值，並不對子運算式求值，需要注意的是，引用在求值之前，其運算式中的原子的符號空間是未知不確定的，引用在被展開並且其子原子

被求值時的棧環境才是確定其原子符號空間的依據，這樣在你的引用運算式設計過程中就要注意你不能認為運算式中的原子就是你棧上的原子。

簡記為`a 實際為(quote a) 求值結果 a
`(list 1 2 3)=> (quote (list 1 2 3)) 求值結果 (list 1 2 3)
`(1 2 3)=> (quote (1 2 3)) 求值結果 (1 2 3)

第3节.絕對反引用(,)

只要其父運算式引用棧上有引用解開動作，則發生求值動作，將結果替換為求值結果
簡記 ,x 實際為(BackQuote x)

第4节.棧式反引用(;)

與絕對反引用不同的是，他只在父運算式棧上最近的引用被展開時才發生求值動作。
簡記;x 實際為(BackQuote_Stack x)
該反引用是為了彌補 Common Lisp 中多層高階反引用不能在二次展開過程中具備反引用能力而設計的。

第5节.隱式引用(')

配合棧式反引用技術使用，因為棧式反引用單獨使用時其存在一個缺陷，就是代碼生成過程中在之前不能夠在代碼中有引用存在，因為引用會導致棧式反引用的求值棧時機發生變化。
隱式引用起作用主要是為了使得棧式反引用;和最近的引用`求值時中間能加一個引用`，如果沒有這個隱式引用存在，那麼當需要生成引用的代碼時，反引用將不能自由控制自己的求值時機

隱式引用其在反引用展開過程中被求值為一個正常的引用求值器。
簡記'x 實際為 (BackQuote_Quote x) 求值結果為`x 即(quote x)

第6节.反引用簡單實例

(list (car (list 1 2 3))) 求值結果為 (1)
`(list ,(car (list 1 2 3))) 求值結果為 (list 1)
`(list ;(car (list 1 2 3))) 求值結果為 (list 1)
``(list ,(car (list 1 2 3))) 求值結果為 `(list 1)
``(list ;(car (list 1 2 3))) 求值結果為 `(list ;(car (list 1 2 3)) 如果再次對結果求值則得到 (list 1)

隱式反引用應用場合

`(a `;a)求值結果 (a `;a)
`(a ';(cdr (list x y z)))  求值結果為  (a `(y z))
``(a ';(cdr (list x y z))) 求值結果為 `(a ';(cdr (list x y z)))
`(a `;(cdr (list x y z))) 求值結果為 (a `;(cdr (list x y z)))

第六章. 簡單數值求值器

第1节. Add Mul Mod LessThan

第七章. 輔助求值器

第1节. Lisp 代碼檔載入(RefLisp)

(RefLisp "..\Test\TestStack.txt")
RefLisp 是個很特別的函數，對其求值就是對"..\Test\TestStack.txt"檔列表中的 Lisp 求值，並且求值後 RefLisp 所在的列表也被替換成"..\Test\TestStack.txt"中的 lisp 物件

第2节. Lisp 原子資訊日誌輸出與配置(DebugLisp)

(DebugLisp (需要調試的目標 lisp 清單) (日誌打開項清單) (日誌關閉項清單))
功能：列印目標 lisp 值以及開啟關閉日誌輸出項
注意：其中 日誌打開項清單和日誌關閉項清單可以沒有或者為空表，需要調試的目標 lisp 清單中 lisp 物件不被求值，只是引用列印。
可以開啟或關閉的日誌過濾項有：
求值錯誤
求值成功
所有求值過程
日誌 1 - 日誌 8

比如 (DebugLisp () (求值錯誤) (求值成功)) 表示顯示求值錯誤資訊，關閉求值成功資訊

第3节. Lisp 原子中斷點調試(dp)

(bp (lisp 物件清單) (中斷點時機列表) (中斷點條件運算式求值為 T 則設置中斷點 Nil 則取消該中斷點))
中斷點時機列表
求值前
求值後

求職失敗

比如 (bp (y) (求值後) (< y 3))

第4节. Lisp 注釋(REM)

(REM Lisp 運算式或者一個字串)
REM 本身做一次空求值返回 T

第二部分MyLisp DSL 篇

第一章. 基本抽象庫 CommonDSL

第1节. 定義函數(defun)

第2节. 定義宏(defmacro)

第3节. 定義函數巨集 (defun_m)

第4节. 處理列表(dolist)

第5节. 調用處理 N 次(dotimes）

第6节. Lisp 物件模式路由(LispCodeModeRoute)

第7节. Lisp 代碼模式合成(LispCodeModeMake）

第二章. LikeC (Common Lisp)

第1节. C 基本語法

第2节. Class 資料結構

第三章. LikeProlog

第1节. 論域描述

第2节. 謂詞邏輯