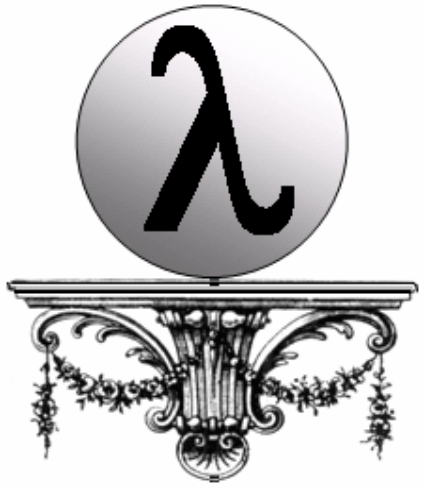
TS. PHAN HUY KHÁNH



Lập trình hàm

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

Mục lục

LỜI NÓI ĐẦU

Cuốn sách này trình bày cơ sở lý thuyết và những kỹ thuật lập trình cơ bản theo phong cách «lập trình hàm» (Functional Programming). Đây là kết quả biên soạn từ các giáo trình sau nhiều năm giảng dạy bậc đại học và sau đại học ngành công nghệ thông tin của tác giả tại Đại học Đà Nẵng.

Cuốn sách gồm sáu chương có nội dung như sau :

- *Chương 1 giới thiệu quá trình phát triển và phân loại các ngôn ngữ lập trình, những đặc điểm cơ bản của phong cách lập trình mệnh lệnh. Phần chính của chương trình bày nhữngnguyên lý lập trình hàm sử dụng ngôn ngữ minh hoạ Miranda.*
- *Chương 2 trình bày những kiến thức mở đầu về ngôn ngữ Scheme : các khái niệm và các kiểu dữ liệu cơ sở, cách định nghĩa hàm, biểu thức, kỹ thuật sử dụng đệ quy và phép lặp.*
- *Chương 3 trình bày các kiểu dữ liệu phức hợp của Scheme như chuỗi, vectơ, danh sách và cách vận dụng các kiểu dữ liệu trừu tượng trong định nghĩa hàm.*
- *Chương 4 trình bày những kiến thức nâng cao về kỹ thuật lập trình hàm, định nghĩa hàm nhờ phép tính lambda, ứng dụng thuật toán quay lui, truyền thông điệp...*
- *Chương 5 trình bày chi tiết hơn kỹ thuật lập trình nâng cao với Scheme sử dụng các cấu trúc dữ liệu : tập hợp, ngăn xếp, hàng đợi, cây và tệp.*
- *Chương 6 trình bày khái niệm môi trường, cách tổ chức và cấp phát bộ nhớ, cách vào/ra dữ liệu của Scheme với thế giới bên ngoài.*
- *Phần phụ lục giới thiệu vắn tắt ngôn ngữ lập trình WinScheme48, hướng dẫn cách cài đặt và sử dụng phần mềm này.*

Cuốn sách này làm tài liệu t khảo cho sinh viên các ngành công nghệ thông tin và những bạn đọc muốn tìm hiểu thêm về kỹ thuật lập trình cho lĩnh vực trí tuệ nhân tạo, giao tiếp hệ thống, xử lý ký hiệu, tính toán hình thức, các hệ thống đồ hoạ...

Trong suốt quá trình biên soạn, tác giả đã nhận được từ các bạn đồng nghiệp nhiều đóng góp bổ ích về mặt chuyên môn, những động viên khích lệ về mặt tinh thần, sự giúp đỡ tận tình về biên tập để cuốn sách được ra đời. Do mới xuất bản lần đầu tiên, tài liệu tham khảo chủ yếu là tiếng nước ngoài, chắc chắn rằng nội dung của cuốn sách vẫn còn bộc lộ nhiều thiếu sót, nhất là các thuật ngữ dịch ra tiếng Việt.

Tác giả xin được bày tỏ ở đây lòng biết ơn sâu sắc về mọi ý kiến phê bình đóng góp của bạn đọc gần xa.

Đà Nẵng, ngày 06/09/2004
Tác giả.

PHAN HUY KHÁNH

LẬP TRÌNH HÀM

Functional Programming

Lập trình hàm là phong cách lập trình dựa trên định nghĩa hàm sử dụng phép tính lambda (λ-calculus). Lập trình hàm không sử dụng các lệnh gán biến và không gây ra hiệu ứng phụ như vẫn gặp trong lập trình mệnh lệnh. Trong các ngôn ngữ lập trình hàm, hàm (thủ tục, chương trình con) đóng vai trò trung tâm, thay vì thực hiện lệnh, máy tính tính biểu thức. Đã có rất nhiều ngôn ngữ hàm được phát triển và ứng dụng như Miranda, Haskell, ML, các ngôn ngữ họ Lisp : Scheme, Common Lisp...

Phần đầu cuốn sách này trình bày cơ sở lý thuyết và những khái niệm cơ bản của lập trình hàm sử dụng ngôn ngữ minh hoạ là Miranda, một ngôn ngữ thuần tuý hàm do D. Turner đề xuất 1986. Phần chính trình bày kỹ thuật lập trình hàm trong Scheme, một ngôn ngữ do Guy Lewis Steele Jr. và G. Jay Sussman đề xuất 1975.

Ngôn ngữ Scheme có tính sư phạm cao, giải quyết thích hợp các bài toán toán học và xử lý ký hiệu. Scheme có cú pháp đơn giản, dễ học, dễ lập trình. Một chương trình Scheme là một dãy các định nghĩa hàm góp lại để định nghĩa một hoặc nhiều hàm phức tạp hơn. Scheme làm việc theo chế độ thông dịch, tương tác với người sử dụng.

Cuốn sách này rất thích hợp cho sinh viên các ngành công nghệ thông tin và những bạn đọc muốn tìm hiểu về kỹ thuật lập trình ứng dụng trong lĩnh vực trí tuệ nhân tạo, giao tiếp người-hệ thống, xử lý ký hiệu, tính toán hình thức, thiết kế các hệ thống đồ hoạ...

VỀ TÁC GIẢ :

Tốt nghiệp ngành Toán Máy tính năm 1979 tại trường Đại học Bách khoa Hà Nội. Từ 1979 đến nay giảng dạy tại khoa Công nghệ Thông tin, trường Đại học Bách khoa, Đại học Đà Nẵng. Bảo vệ tiến sĩ năm 1991 tại Pháp. Giữ chức chủ nhiệm khoa Công nghệ Thông tin 1995-2000.

Hướng nghiên cứu chính : xử lý ngôn ngữ, xử lý đa ngữ, lý thuyết tính toán.

E-mail: phanhuykhanh@dng.vnn.vn

PHỤ LỤC

Scheme48 for Windows

Hiện nay có nhiều phiên bản trình thông dịch Scheme cung cấp miễn phí trên Internet. Trong cuốn sách này, tác giả sử dụng chủ yếu Scheme48 for Windows, phiên bản 0.52, để chạy minh hoạ các ví dụ trong phần trình bày lý thuyết. Đây là phần mềm được phát triển từ năm 1993 bởi R. Kelsey và J. Rees, sau đó được tiếp tục phát triển tại trường Đại học Northwestern, Chicago, Hoa Kỳ. Phiên bản mới nhất hiện nay là WinScheme48 0.57 được tìm thấy tại trang web :

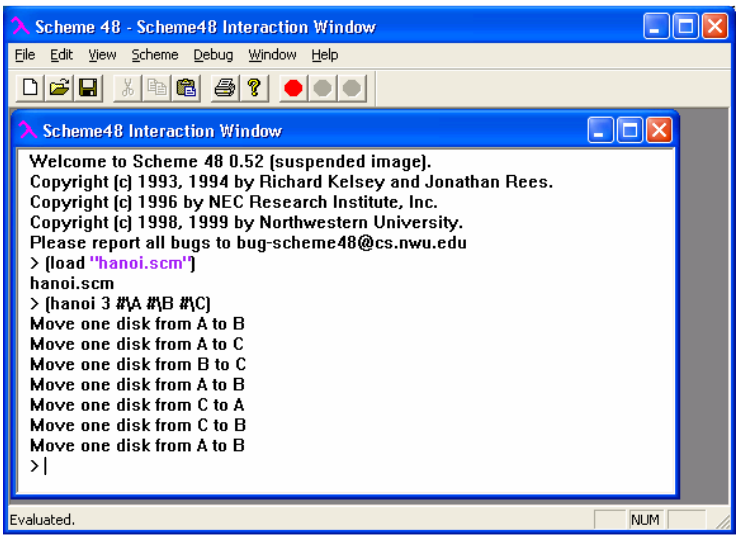
http://s48.org/index.html, hoặc trang web của trường Đại học Northwestern :

http://www.cs.nwu.edu/groups/su/edwin/.

Bạn đọc có thể tìm thấy nhiều phiên bản Scheme khác như MITScheme, DrScheme, … Những phiên bản này không ngừng được cập nhật, đổi mới và có thư viện s-lib rất phong phú. Tuy nhiên đối với những bạn đọc mới bắt đầu làm quen lập trình hàm với ngôn ngữ Scheme, chỉ nên chạy phiên bản 0.52 gọn nhẹ và tương đối ổn định trong mọi nền Win98/NT/2K hay XP được tải về từ địa chỉ sau :

http://www.cs.nwu.edu/groups/su/Scheme48/program/Scheme48.zip

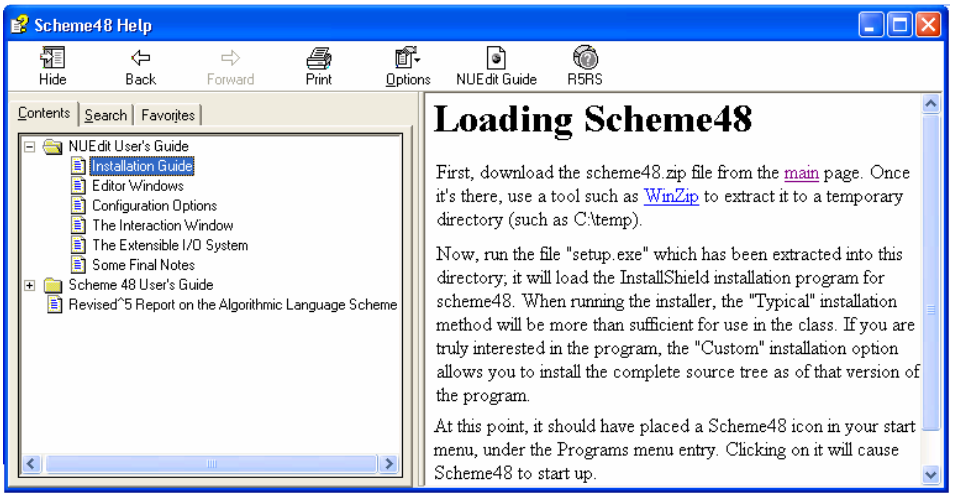
Tiến trình cài đặt và thực hiện các thao tác chỉ định đường dẫn rất dễ dàng từ tệp nén Scheme48.zip chứa đủ bộ thông dịch và tài liệu hướng dẫn. Sau khi khởi động, màn hình làm việc của Scheme48 for Windows như sau :



Hình 1. Phiên bản Scheme48 for Windows 0.52.

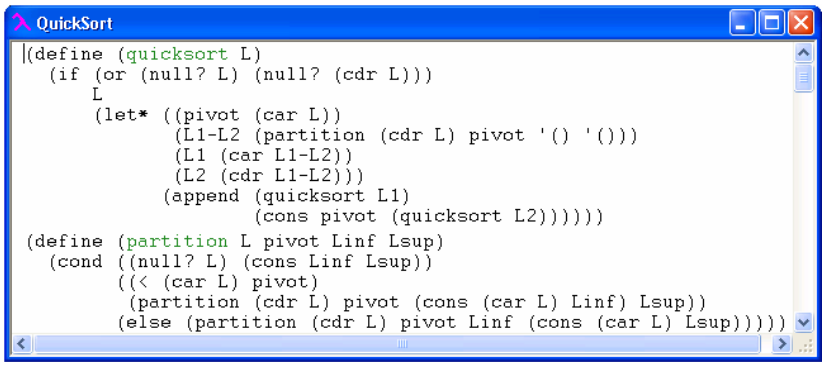
WinScheme48 rất dễ thao tác và dễ sử dụng. Cửa sổ làm việc chứa dấu nhắc lệnh là một dấu lớn hơn >. Sau mỗi dòng lệnh, nếu s-biểu thức đưa vào đã đúng đắn về mặt cú pháp, WinScheme48 sẽ tiến hành thực hiện (evaluation) để trả về kết quả.

Để xem tài liệu hướng dẫn, NSD gọi chạy chương trình `scheme48.chm` có sẵn trong thư mục `..\Scheme48\doc` đã được tự động tạo ra sau khi cài đặt (xem hình 2).



Hình 2. Hướng dẫn sử dụng Scheme48 for Windows 0.52.

Trình soạn thảo văn bản thường trực của WinScheme48 tương tự NotePad/WordPad của Windows nên rất dễ sử dụng, nhờ các lệnh sao chép cut/paste. Khi soạn thảo chương trình, các dòng lệnh được tự động thụt dòng (indentation) và thay đổi màu sắc giúp NSD dễ dàng kiểm tra cú pháp, phát hiện lỗi.



Hình 3. Cửa sổ soạn thảo chương trình.

Trong cửa sổ soạn thảo, NSD có thể định nghĩa các hàm, các biến, các chuỗi hay đưa vào các dòng chú thích. WinScheme48 cung cấp bộ kiểm tra dấu ngoặc (parenthesis matching), chẳng hạn phần chương trình đứng trước dấu chèn có màu xanh dương cho biết s-biểu thức tương ứng đã hợp thức về mặt cú pháp, màu đỏ là xảy ra lỗi do thừa dấu ngoặc. Sau khi soạn thảo các hàm, có thể thực hiện chương trình bằng cách:

- Lựa (highlight) vùng hàm (s-biểu thức) cần thực hiện, hoặc đặt con trỏ (dấu chèn) tại một vị trí bên trong.
- Nhấn tổ hợp phím `Ctrl+Alt+X` hoặc gọi lệnh `Scheme-Eval` ((evaluate) trên thanh lệnh đơn (menu bar).

Tài liệu tham khảo chính

[1] H. Abdulrab. *de Common Lisp à la programmation objet*. Editions HERMES, Paris 1990.

[2] D. Appleby & J.J.VandeKopple. *Programming Language: Paradigm and Pratice*. THE MCGRAW–HILL COMPANIES, INC., 1997.

[3] J. Arsac. *Nhập môn lập trình*. Đinh Văn Phong và Trần Ngọc Trí dịch. Trung tâm Hệ thống Thông tin ISC, Hà Nội 1991.

[4] H. E. Bal & D. Grune. *Programming Language Essentials*. ADDITION-WESLEY PUBLISHING COMPANY INC., 1994.

[5] J. Bentley. *Nhữngviên ngọc trong kỹ thuật lập trình*. Lê Minh Trung và Nguyễn Văn Hiếu dịch, Trung tâm Tin học và Điện tử Phương Đông xuất bản, 1992.

[6] J. Chazarain. *Programmer avec Scheme – de la pratique à la théorie*. INTERNATIONAL THOMSON PUBLISHING, Paris 1996.

[7] W. Clinger, J. Rees & all. *Revised5. Prport on the Algorithmic Language Scheme*. Tài liệu Internet : http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html

[8] H. Farrency & M. Ghallab. *Elément d'intélligence artificielle*. Editions HERMES, Paris 1990.

[9] Ch. Froidevaux và các tác giả khác. *Types de Données et Algorithmes*. EDISCIENCE international, Paris 1994.

[10] Phan Huy Khánh. *Giáo trình lập trình hàm*. Giáo trình xuất bản nội bộ, Đại học Đà Nẵng 2002.

[11] Phan Huy Khánh & Phan Chí Tùng. *Nhập môn Tin học*. Nhà Xuất bản Giáo dục, 1997.

[12] Đỗ Xuân Lôi. Cấu trúc dữ liệu và giải thuật. Nhà Xuất bản Giáo dục, 1993.

[13] Ch. Rapin. *Programmation Déclarative*. Springer Verlag 1993.

[14] Y. Rouzaud. *Algorithmique et Programmation en Scheme*. Tài liệu nội bộ, ENSIMAG, Grenoble 1996.

[15] D. A. Turner. *An Overview of Miranda*. Springer Lecture Notes in Computer Science, vol 201, 1986.

[16] Ngô Trung Việt. *Kiến thức cơ bản về lập trình*. Nhà Xuất bản Giao thông Vận tải, 1995.

[17] N. Wirth. "Algorithms + Data Structures = Programs", Prentice-Hall 1976.

[18] J. Malenfant. *Sémantique des langages de programmation*. Tài liệu lấy từ internet : Université de Bretagne-Sud, Pháp.

CHƯƠNG I. NGUYÊN LÝ LẬP TRÌNH HÀM

« The primary purpose of a programming language
is to help the programmer in the practice of his art »
Charle A. Hoare (*Hints on programming language design*, 1973)

I.1 Mở đầu về ngôn ngữ lập trình

I.1.1. Vài nét về lịch sử

Buổi ban đầu

Những *ngôn ngữ lập trình* (programming language) đầu tiên trên máy tính điện tử là *ngôn ngữ máy* (machine language), tổ hợp của các con số hệ hai, hay hệ nhị phân, hay các *bit* (viết tắt của **bi**nary digi**t**) 0 và 1. Ngôn ngữ máy phụ thuộc hoàn toàn vào kiến trúc phần cứng của máy tính và những quy ước khắt khe của nhà chế tạo. Để giải các bài toán, người lập trình phải sử dụng một tập hợp các lệnh điều khiển rất sơ cấp mà mỗi lệnh là một tổ hợp các số hệ hai nên gặp rất nhiều khó khăn, mệt nhọc, rất dễ mắc phải sai sót, nhưng lại rất khó sửa lỗi.

Từ những năm 1950, để giảm nhẹ việc lập trình, người ta đưa vào kỹ thuật *chương trình con* (sub-program hay sub-routine) và xây dựng các *thư viện chương trình* (library) để khi cần thì gọi đến hoặc dùng lại những đoạn chương trình đã viết.

Ngôn ngữ máy tiến gần đến ngôn ngữ tự nhiên

Cũng từ những năm 1950, ngôn ngữ *hợp dịch*, hay *hợp ngữ* (assembly) hay cũng còn được gọi là ngôn ngữ *biểu tượng* (symbolic) ra đời. Trong hợp ngữ, các mã lệnh và địa chỉ các toán hạng được thay thế bởi các từ tiếng Anh gợi nhớ (mnemonic) như `ADD`, `SUB`, `MUL`, `DIV`, `JUMP`... tương ứng với các phép toán số học + - × /, phép chuyển điều khiển, v.v...

Do máy tính chỉ hiểu ngôn ngữ máy, các chương trình viết bằng hợp ngữ không thể chạy ngay được mà phải qua giai đoạn *hợp dịch* (assembler) thành ngôn ngữ máy. Tuy nhiên, các hợp ngữ vẫn còn phụ thuộc vào phần cứng và xa lạ với ngôn ngữ tự nhiên (natural language), người lập trình vẫn còn gặp nhiều khó khăn khi giải các bài toán trên máy tính.

Năm 1957, hãng IBM đưa ra ngôn ngữ FORTRAN (FORmula TRANslator). Đây là ngôn ngữ lập trình đầu tiên gần gũi ngôn ngữ tự nhiên với cách diễn đạt toán học. FORTRAN cho phép giải quyết nhiều loại bài toán khoa học, kỹ thuật và sau đó được nhanh chóng ứng dụng rất rộng rãi cho đến ngày nay với kho tàng thư viện thuật toán rất đồ sộ và tiện dụng. Tiếp theo là sự ra đời của các ngôn ngữ ALGOL 60 (ALGOrithmic Language) năm 1960, COBOL (Comon Business Oriented Language) năm 1964, Simula năm 1964, v.v...

Phát triển của ngôn ngữ lập trình

Theo sự phát triển của các thế hệ máy tính, các ngôn ngữ lập trình cũng không ngừng được cải tiến và hoàn thiện để càng ngày càng đáp ứng nhu cầu của người sử dụng và giảm nhẹ công việc lập trình. Rất nhiều ngôn ngữ lập trình đã ra đời trên nền tảng *lý thuyết tính toán* (theory of computation) và hình thành hai loại ngôn ngữ : ngôn ngữ bậc thấp và ngôn ngữ bậc cao.

Các ngôn ngữ *bậc thấp* (low-level language), hợp ngữ và ngôn ngữ máy, thường chỉ dùng để viết các chương trình điều khiển và kiểm tra thiết bị, chương trình sửa lỗi (debugger) hay công cụ...

Các ngôn ngữ lập trình *bậc cao* (high-level language) là phương tiện giúp người làm tin học giải quyết các vấn đề thực tế nhưng đồng thời cũng là nơi mà những thành tựu nghiên cứu mới nhất của khoa học máy tính được đưa vào. Lĩnh vực nghiên cứu phát triển các ngôn ngữ lập trình vừa có tính truyền thống, vừa có tính hiện đại. Ngày nay, với những tiến bộ của khoa học công nghệ, người ta đã có thể sử dụng các công cụ hình thức cho phép giảm nhẹ công việc lập trình từ lúc phân tích, thiết kế cho đến sử dụng một ngôn ngữ lập trình.

I.1.2. Định nghĩa một ngôn ngữ lập trình

Các ngôn ngữ lập trình bậc cao được xây dựng mô phỏng ngôn ngữ tự nhiên, thường là tiếng Anh (hoặc tiếng Nga những năm trước đây). Định nghĩa một ngôn ngữ lập trình là định nghĩa một *văn phạm* (grammar) để sinh ra các câu đúng của ngôn ngữ đó. Có thể hình dung một văn phạm gồm bốn thành phần : *bộ ký tự, bộ từ vựng, cú pháp* và *ngữ nghĩa.*

1. Bộ ký tự (character set)

Gồm một số hữu hạn các ký tự (hay ký hiệu) được phép dùng trong ngôn ngữ. Trong các máy tính cá nhân, người ta thường sử dụng các ký tự ASCII. Có thể hiểu bộ ký tự có vai trò như bảng chữ cái (alphabet) của một ngôn ngữ tự nhiên để tạo ra các từ (word).

2. Bộ từ vựng (vocabulary)

Gồm một tập hợp các từ, hay *đơn vị từ vựng* (token), được xây dựng từ bộ ký tự. Các từ dùng để tạo thành câu lệnh trong một chương trình và được phân loại tuỳ theo vai trò chức năng của chúng trong ngôn ngữ. Chẳng hạn chương trình Pascal sau đây :

```
program P;
var x, y : integer;
begin
   read(x);
   y:=x+2;
   write(y)
end.
```

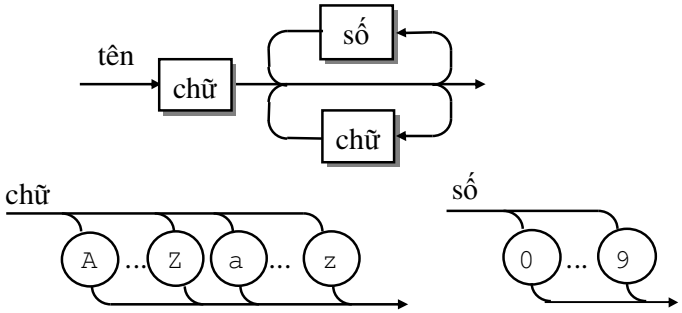
gồm các đơn vị từ vựng :

- Từ khoá (keyword), hay từ dành riêng (reserved word) : **program**, **var**, **integer**, **begin**, **end**.
- Tên, hay định danh (identifier) : `read`, `write`, `P`, `x`, `y`.
- Hằng (constants) : `2`
- Phép toán (operators) : `+` , `:=`
- Dấu phân cách (delimiters) : `:`, `(`, `)`, ...

3. Cú pháp (syntax)

Cú pháp quy định cách thức kết hợp các ký tự thành từ, kết hợp các từ thành câu lệnh đúng (statement hay instruction), kết hợp các câu lệnh đúng thành một chương trình hoàn chỉnh về mặt văn phạm. Có thể hình dung cách kết hợp này giống cách đặt câu trong một ngôn ngữ tự nhiên. Thường người ta dùng sơ đồ cú pháp (syntax diagram) hoặc dạng chuẩn Backus-Naur (Backus-Naur Form, viết tắt BNF), hoặc dạng chuẩn Backus-Naur mở rộng (EBNF – Extended Backus-Naur Form) để mô tả cú pháp của văn phạm.

Ví dụ I.1.1 : Trong ngôn ngữ Pascal (hoặc trong phần lớn các ngôn ngữ lập trình), tên gọi, hay định danh (identifier) có sơ đồ cú pháp như sau :



Hình I.1. Sơ đồ cú pháp tên trong ngôn ngữ Pascal .

Trong một sơ đồ cú pháp, các ô hình chữ nhật lần lượt phải được thay thế bởi các ô hình tròn. Quá trình thay thế thực hiện thứ tự theo chiều mũi tên cho đến khi nhận được câu đúng. Chẳng hạn có thể «đọc» sơ đồ trên như sau : tên phải bắt đầu bằng chữ, tiếp theo có thể là chữ hoặc số tuỳ ý, chữ chỉ có thể là một trong các chũ cái A..Za..z, số chỉ có thể là một trong các chũ số 0..9. Như vậy, `Delta`, `x1`, `x2`, `Read`, v.v... là các tên viết đúng, còn `1A`, β, π, `bán kính`, v.v... đều không phải là tên vì vi phạm quy tắc cú pháp.

Văn phạm BNF gồm một dãy quy tắcc. Mỗi quy tắc gồm *vế trái*, *dấu định nghĩa* ::= (đọc *được định nghĩa bởi*) và *vế phải*. Vế trái là một ký hiệu phải được định nghĩa, còn vế phải là một dãy các ký hiệu, hoặc được thừa nhận, hoặc đã được định nghĩa từ trước đó, tuân theo một quy ước nào đó. EBNF dùng các ký tự quy ước như sau :

Ký hiệu	*Ý nghĩa*
::=, hoặc →, hoặc =	được định nghĩa là
{ }	chuỗi của 0 hay nhiều mục liệt kê tuỳ chọn (option)
[]	hoặc 0 hoặc 1 mục liệt kê tuỳ chọn
< >	mục liệt kê phải được thay thế
|	hoặc (theo nghĩa loại trừ)

Các quy tắc BNF định nghĩa tên trong ngôn ngữ Pascal :

<*tên*> ::= <*chữ*> { <*chữ*> | <*số*> }
<*chữ*> ::= `'A'|...|'Z'|'a'|...|'z'`
<*số*> ::= `'0'|...|'9'`

Ví dụ I.1.2

Văn phạm của một ngôn ngữ lập trình đơn giản dạng EBNF như sau :

```
<program>    ::= program <statement>* end
<statement>  ::= <assignment> | <loop>
<assignment> ::= <identifier> := <expression> ;
```

```
<loop>       ::=
       while <expression> do <statement>+ done
<expression> ::=
       <value> | <value> + <value> | <value> <= <value>
<value>      ::= <identifier> | <number>
<identifier> ::=
       <letter>|<identifier><letter>|<identifier><digit>
<number>     ::= <digit> | <number><digit>
<letter>     ::= 'A' | … | 'Z' | 'a' | … | 'z'
<digit>      ::= '0' | … | '9'
```

Một câu, tức là một chương trình đơn giản, viết trong văn phạm trên như sau :

```
program
  n := 1 ;
  while n <= 10 do n := n + 1 ; done
end
```

4. Ngữ nghĩa (semantic)

Căn cứ vào cú pháp của ngôn ngữ lập trình, người lập trình viết chương trình gồm các câu lệnh theo trình tự cho phép để giải quyết được bài toán của mình. Để đạt được mục đích đó, mỗi câu lệnh viết ra không những đúng đắn về mặt cú pháp, mà còn phải đúng đắn cả về mặt ngữ nghĩa, hay ý nghĩa logic của câu lệnh. Tính đúng đắn về mặt ngữ nghĩa cho phép giải quyết được bài toán, chương trình chạy luôn luôn dừng, ổn định và cho kết quả phù hợp với yêu cầu đặt ra ban đầu.

I.1.3. Khái niệm về chương trình dịch

Chương trình được viết trong một ngôn ngữ lập trình bậc cao, hoặc bằng hợp ngữ, đều được gọi là *chương trình nguồn* (source program).

Bản thân máy tính không hiểu được các câu lệnh trong một chương trình nguồn. Chương trình nguồn phải được *dịch* (translate) thành *một chương trình đích* (target program) trong ngôn ngữ máy (là các dãy số *0* và *1*), máy mới có thể đọc «hiểu» và thực hiện được. Chương trình đích còn được gọi là *chương trình thực hiện* (executable program).

Chương trình trung gian đảm nhiệm việc dịch đó được gọi là các *chương trình dịch*.

Việc thiết kế chương trình dịch cho một ngôn ngữ lập trình đã cho là cực kỳ khó khăn và phức tạp. Chương trình dịch về nguyên tắc phải viết trên ngôn ngữ máy để giải quyết vấn đề xử lý ngôn ngữ và tính vạn năng của các chương trình nguồn. Tuy nhiên, người ta thường sử dụng hợp ngữ để viết các chương trình dịch. Bởi vì việc dịch một chương trình hợp ngữ ra ngôn ngữ máy đơn giản hơn nhiều. Hiện nay, người ta cũng viết các chương trình dịch bằng chính các ngôn ngữ bậc cao hoặc các công cụ chuyên dụng.

Thông thường có hai loại chương trình dịch, hay hai chế độ dịch, là trình biên dịch và trình thông dịch, hoạt động như sau :

- Trình *biên dịch* (compilater) dịch toàn bộ chương trình nguồn thành chương trình đích rồi sau đó mới bắt đầu tiến hành thực hiện chương trình đích.
- Trình *thông dịch* (interpreter) dịch lần lượt từng câu lệnh một của chương trình nguồn rồi tiến hành thực hiện luôn câu lệnh đã dịch đó, cho tới khi thực hiện xong toàn bộ chương trình.

Có thể hiểu trình biên dịch là dịch giả, trình thông dịch là thông dịch viên.

Những ngôn ngữ lập trình cấp cao ở chế độ biên dịch hay gặp là : Fortran, Cobol, C, C++, Pascal, Ada, Basic... Ở chế độ thông dịch hay chế độ tương tác : Basic,Lisp, Prolog...

I.1.4. Phân loại các ngôn ngữ lập trình

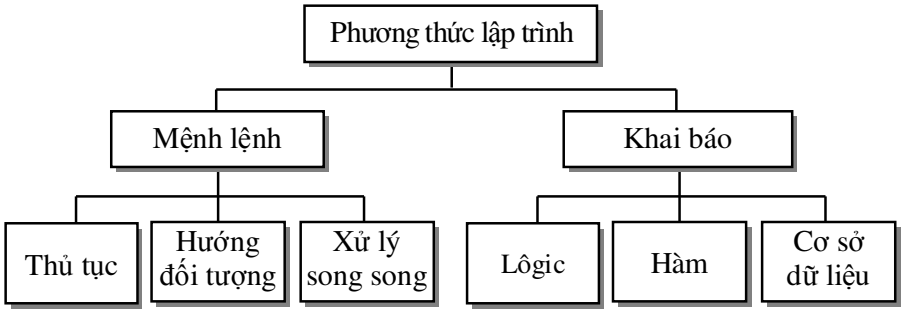
Cho đến nay, đã có hàng trăm ngôn ngữ lập trình được đề xuất nhưng trên thực tế, chỉ có một số ít ngôn ngữ được sử dụng rộng rãi. Ngoài cách phân loại theo *bậc* như đã nói ở trên, người ta còn phân loại ngôn ngữ lập trình theo *phương thức* (paradigm), theo *mức độ quan trọng* (measure of emphasis), theo *thế hệ* (generation), v.v...

Cách phân loại theo *bậc* hay *mức* (level) là dựa trên mức độ trừu tượng so với các yếu tố phần cứng, chẳng hạn như *lệnh* (instructions) và *cấp phát bộ nhớ* (memory allocation).

Mức	Lệnh	Sử dụng bộ nhớ	Ví dụ
Thấp	Lệnh máy đơn giản	Truy cập và cấp phát trực tiếp	Hợp ngữ, Autocode
Cao	Biểu thức và điều khiển tường minh	Truy cập và cấp phát nhờ các phép toán, chẳng hạn new	FORTRAN, ALGOL, Pascal, C, Ada
Rất cao	Máy trừu tượng	Truy cập ẩn và tự động cấp phát	SELT, Prolog, Miranda

Hình I.2. Ba mức của ngôn ngữ lập trình.

Những năm gần đây, ngôn ngữ lập trình được phát triển theo phương thức lập trình (còn được gọi là phong cách hay kiểu lập trình). Một phương thức lập trình có thể được hiểu là một tập hợp các tính năng trừu tượng (abstract features) đặc trưng cho một lớp ngôn ngữ mà có nhiều người lập trình thường xuyên sử dụng chúng. Sơ đồ sau đây minh hoạ sự phân cấp của các phương thức lập trình :



Hình I.3. Phân cấp của các phương thức lập trình.

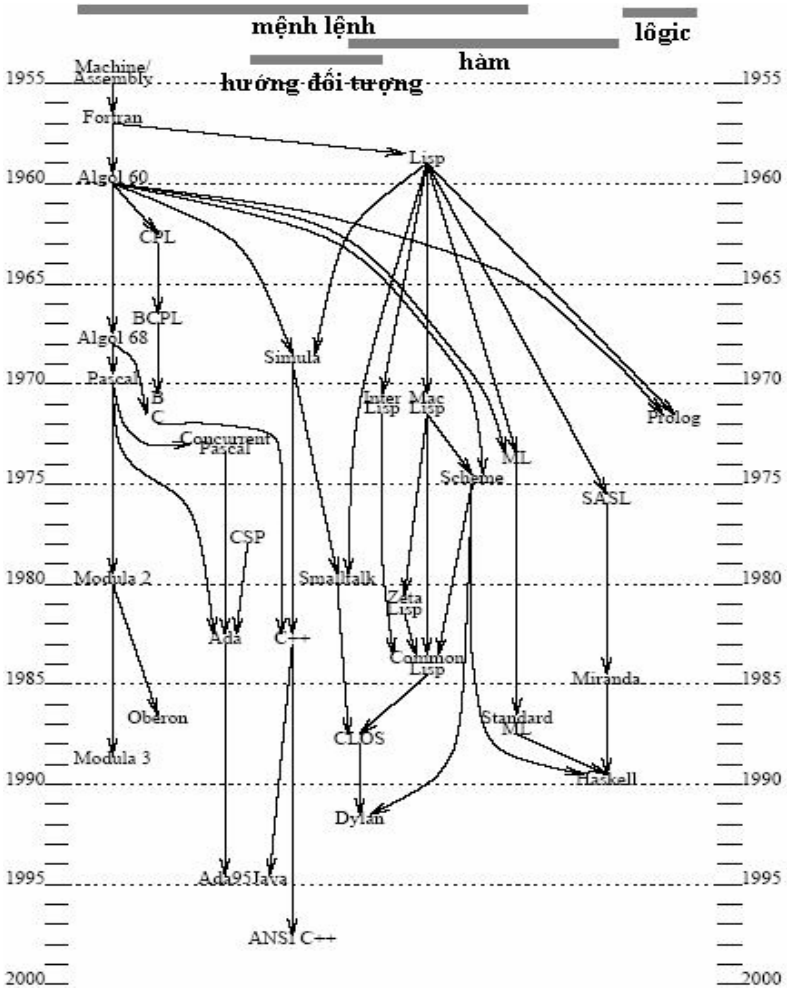
Sau đây là một số ngôn ngữ lập trình quen thuộc liệt kê theo phương thức :

- Các ngôn ngữ *mệnh lệnh* (imperative) có Fortran (1957), Cobol (1959), Basic (1965), Pascal (1970), C (1971), Ada (1979)...
- Các ngôn ngữ *định hướng đối tượng* (object-oriented) có Smalltalk (1969), C++ (1983), Eiffel (1986), Java (1991), C# (2000), ...
- Các ngôn ngữ *hàm* (functional) có Lisp (1958), ML (1973), Scheme (1975), Caml (1987), Miranda (1982), ...
- Các ngôn ngữ *dựa logic* (logic-based) chủ yếu là ngôn ngữ Prolog (1970).
- Ngôn ngữ thao tác cơ sở dữ liệu như SQL (1980)...
- Các ngôn ngữ xử lý *song song* (parallel) như Ada, Occam (1982), C-Linda, ...

Ngoài ra còn có một số phương thức lập trình đang được phát triển ứng dụng như :

- *Lập trình phân bổ* (distributed programming).
- *Lập trình ràng buộc* (constraint programming).
- *Lập trình hướng truy cập* (access-oriented programming).
- *Lập trình theo luồng dữ liệu* (dataflow programming), v.v...

Việc phân loại các ngôn ngữ lập trình theo *mức độ quan trọng* là dựa trên *cái gì* (what) sẽ thao tác được (achieved), hay tính được (computed), so với cách thao tác *như thế nào* (how). Một ngôn ngữ thể hiện cái gì sẽ thao tác được mà không chỉ ra cách thao tác như thế nào được gọi là ngôn ngữ *định nghĩa* (definitional) hay *khai báo* (declarative). Một ngôn ngữ thể hiện cách thao tác như thế nào mà không chỉ ra cái gì sẽ thao tác được gọi là ngôn ngữ *thao tác* (operational) hay *không khai báo* (non-declarative), đó là các ngôn ngữ mệnh lệnh.



Hình I.4. Phát triển của ngôn ngữ lập trình.

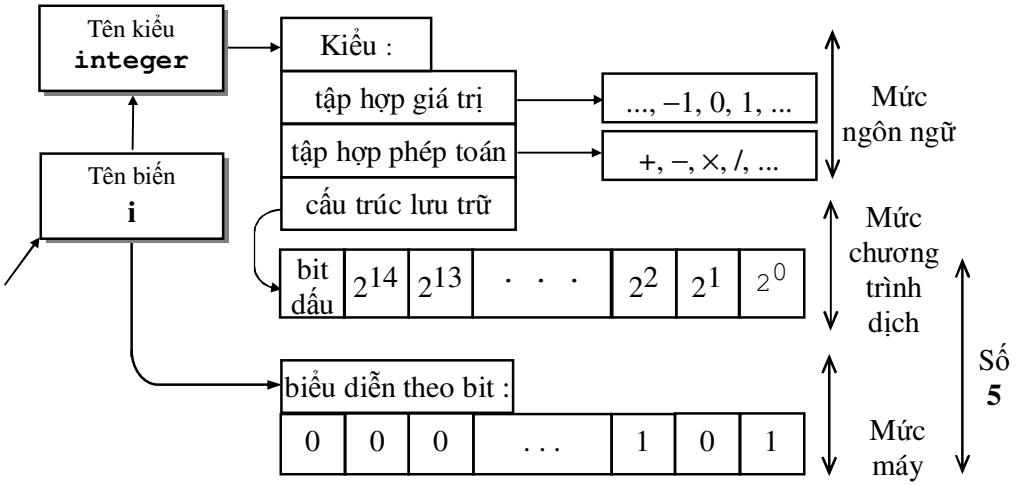
Các ngôn ngữ lập trình cũng được phân loại theo *thế hệ* như sau :

- Thế hệ 1 : ngôn ngữ máy
- Thế hệ 2 : hợp ngữ
- Thế hệ 3 : ngôn ngữ thủ tục
- Thế hệ 4 : ngôn ngữ áp dụng hay hàm
- Thế hệ 5 : ngôn ngữ suy diễn hay dựa logic
- Thế hệ 6 : mạng nơ-ron (neural networks)

Trước khi nghiên cứu lớp các ngôn ngữ lập trình hàm, ta cần nhắc lại một số đặc điểm của lớp các ngôn ngữ lập trình mệnh lệnh.

I.1.5. Ngôn ngữ lập trình mệnh lệnh

Trong các ngôn ngữ mệnh lệnh, người lập trình phải tìm cách diễn đạt được thuật toán, cho biết *làm cách nào* để giải một bài toán đã cho. Mô hình tính toán sử dụng một tập hợp (hữu hạn) các trạng thái và sự thay đổi trạng thái. Mỗi trạng thái phản ánh nội dung các biến dữ liệu đã được khai báo. Trạng thái luôn bị thay đổi do các lệnh điều khiển và các lệnh gán giá trị cho các biến trong chương trình. Chương trình biên dịch cho phép lưu giữ các trạng thái trong bộ nhớ chính và thanh ghi, rồi chuyển các phép toán thay đổi trạng thái thành các lệnh máy để thực hiện.



Hình I.5. Quan hệ giữa tên biến, kiểu và giá trị trong ngôn ngữ mệnh lệnh

Hình I.5. minh họa cách khai báo dữ liệu trong các ngôn ngữ mệnh lệnh và các mối quan hệ theo mức. Người ta phân biệt ba mức như sau : mức ngôn ngữ liên quan đến tên biến, tên kiểu dữ liệu và cấu trúc lưu trữ ; mức chương trình dịch liên quan đến phương pháp tổ chức bộ nhớ và mức máy cho biết cách biểu diễn theo bit và giá trị dữ liệu tương ứng. Mỗi khai báo biến, ví dụ **`int i`**, *nối kết* (bind) tên biến (**`i`**) với một cấu trúc đặc trưng bởi tên kiểu (**`int`**) và với một giá trị dữ liệu được biểu diễn theo bit nhờ lệnh gán **`i := 5`** (hoặc nhờ một lệnh vừa khai báo vừa khởi gán **`int`** `i=5`). Tổ hợp tên, kiểu và giá trị đã tạo nên đặc trưng của biến.

Các ngôn ngữ mệnh lệnh được sử dụng hiệu quả trong lập trình do người lập trình có thể tác động trực tiếp vào phần cứng. Tuy nhiên, tính thực dụng mệnh lệnh làm hạn chế trí tuệ của người lập trình do phải phụ thuộc vào cấu trúc vật lý của máy tính. Người lập trình luôn có khuynh hướng suy nghĩ về những vị trí lưu trữ dữ liệu đã được đặt tên (nguyên tắc địa chỉ hoá) mà nội dung của chúng thường xuyên bị thay đổi. Thực tế có rất nhiều bài toán cần sự trừu tượng hoá khi giải quyết (nghĩa là không phụ thuộc vào cấu trúc vật lý của máy tính), không những đòi hỏi tính thành thạo của người lập trình, mà còn đòi hỏi kiến thức Toán học tốt và khả năng trừu tượng hoá của họ.

Từ những lý do trên mà người ta tìm cách phát triển những mô hình tương tác không phản ánh mối quan hệ với phần cứng của máy tính, mà làm dễ dàng lập trình. Ý tưởng của mô hình là người lập trình cần đặc tả cái gì sẽ được tính toán mà không phải mô tả cách tính như thế nào. Sự khác nhau giữa «*như thế nào*» và «*cái gì*», cũng như sự khác nhau giữa các ngôn ngữ

mệnh lệnh và các ngôn ngữ khai báo, không phải luôn luôn rõ ràng. Các ngôn ngữ khai báo thường khó cài đặt và khó vận hành hơn các ngôn ngữ mệnh lệnh. Các ngôn ngữ mệnh lệnh thường gần gũi người lập trình hơn.

Sau đây là một số đặc trưng của ngôn ngữ lập trình mệnh lệnh :

- Sử dụng nguyên lý tinh chế từng bước hay làm mịn dần, xử lý lần lượt các đối tượng dữ liệu đã được đặt tên.
- Khai báo dữ liệu để nối kết một tên biến đã được khai báo với một kiểu dữ liệu và một giá trị. Phạm vi hoạt động (scope) của các biến trong chương trình được xác định bởi các khai báo, hoặc toàn cục (global), hoặc cục bộ (local).
- Các kiểu dữ liệu cơ bản thông dụng là số nguyên, số thực, ký tự và lôgic. Các kiểu mới được xây dựng nhờ các kiểu cấu trúc. Ví dụ kiểu mảng, kiểu bản ghi, kiểu tập hợp, kiểu liệt kê,...
- Hai kiểu dữ liệu có cùng tên thì tương đương với nhau, hai cấu trúc dữ liệu là tương đương nếu có cùng giá trị và có cùng phép toán xử lý.
- Trạng thái trong (bộ nhớ và thanh ghi) bị thay đổi bởi các lệnh gán. Trạng thái ngoài (thiết bị ngoại vi) bị thay đổi bởi các lệnh vào-ra. Giá trị được tính từ các biểu thức.
- Các cấu trúc điều khiển là tuần tự, chọn lựa (rẽ nhánh), lặp và gọi chương trình con.
- Chương trình con thường có hai dạng : dạng *thủ tục* (procedure) và dạng *hàm* (function). Sự khác nhau chủ yếu là hàm luôn trả về một giá trị, còn thủ tục thì không không nhất thiết trả về giá trị. Việc trao đổi tham biến (parameter passing) với chương trình con hoặc *theo trị* (by value) và *theo tham chiếu* (by reference).
- Sử dụng chương trình con thường gây ra *hiệu ứng phụ* (side effect) do có thể làm thay đổi biến toàn cục.
- Một chương trình được xây dựng theo bốn mức : *khối* (block), *chương trinh con*, *đơn thể* (module/packages) và *chương trình*.

I.2 Cơ sở của các ngôn ngữ hàm

I.2.1. Tính khai báo của các ngôn ngữ hàm

Trong các ngôn ngữ mệnh lệnh, một chương trình thường chứa ba lời gọi chương trình con (thủ tục, hàm) liên quan đến quá trình đưa vào dữ liệu, xử lý dữ liệu và đưa ra kết quả tính toán như sau :

```
begin
   GetData(...) ;     { đưa vào }
   ProcessData(...);        { xử lý }
   OutPutResults(...);      {  xem kết quả }
end
```

Trong các ngôn ngữ lập trình hàm, các lời gọi chương trình con được viết thành biểu thức rất đơn giản :

```
(print
    (process-data
        (get-data (...))))
```

Các ngôn ngữ hàm là cũng các ngôn ngữ bậc cao, mang tính trừu tượng hơn so với các ngôn ngữ mệnh lệnh.

Những người lập trình hàm thường tránh sử dụng các biến toàn cục, trong khi đó, hầu hết những người lập trình mệnh lệnh đều phải sử dụng đến biến toàn cục.

Khi lập trình với các ngôn ngữ hàm, người lập trình phải định nghĩa các hàm toán học dễ suy luận, dễ hiểu mà không cần quan tâm chúng được cài đặt như thế nào trong máy.

Những người theo khuynh hướng lập trình hàm cho rằng các lệnh trong một chương trình viết bằng ngôn ngữ mệnh lệnh làm thay đổi trạng thái toàn cục là hoàn toàn bất lợi. Bởi vì rất nhiều phần khác nhau của chương trình (chẳng hạn các hàm, các thủ tục) tác động không trực tiếp lên các biến và do vậy làm chương trình khó hiểu. Các thủ tục thường được gọi sử dụng ở các phần khác nhau của chương trình gọi nên rất khó xác định các biến bị thay đổi như thế nào sau lời gọi. Như vậy, sự xuất hiện hiệu ứng phụ làm cản trở việc *chứng minh tính đúng đắn* (correctness proof), cản trở tối ưu hóa (optimization), và cản trở quá trình song song tự động (automatic parrallelization) của chương trình.

Một ngôn ngữ hàm, hay *ngôn ngữ áp dụng* (applicative language) dựa trên việc tính giá trị của biểu thức được xây dựng từ bên ngoài lời gọi hàm. Ở đây, hàm là một hàm toán học thuần túy : là một ánh xạ nhận các giá trị lấy từ một *miền xác định* (domain) để trả về các giá trị thuộc một miền khác (range hay co-domain).

Một hàm có thể có, hoặc không có, các *tham đối* (arguments hay parameters) để sau khi tính toán, hàm trả về một giá trị nào đó. Chẳng hạn có thể xem biểu thức 2 + 3 là hàm tính tổng (phép +) của hai tham đối là 2 và 3.

Ta thấy rằng các hàm không gây ra hiệu ứng phụ trong trạng thái của chương trình, nếu trạng thái này được duy trì cho các tham đối của hàm. Tính chất này đóng vai trò rất quan trọng trong lập trình hàm. Đó là *kết quả của một hàm không phụ vào* ***thời điểm*** (*when*) *hàm được gọi, mà chỉ phụ thuộc vào* ***cách gọi*** *nó như thế nào đối với các tham đối.*

Trong ngôn ngữ lập trình mệnh lệnh, kết quả của biểu thức :

```
f(x) + f(x)
```

có thể khác với kết quả :

```
2 * f(x)
```

vì lời gọi *f*(*x*) đầu tiên có thể làm thay đổi *x* hoặc một biến nào đó được tiếp cận bởi *f*. Trong ngôn ngữ lập trình hàm, cả hai biểu thức trên luôn có cùng giá trị.

Do các hàm không phụ thuộc nhiều vào các biến toàn cục, nên việc lập trình hàm sẽ dễ hiểu hơn lập trình mệnh lệnh. Ví dụ giả sử một trình biên dịch cần tối ưu phép tính :

```
f(x) + f(x)
```

thành :

```
2 * f(x)
```

Khi đó, trình biên dịch một ngôn ngữ hàm luôn luôn xem hai kết quả là một, do có tính nhất quán trong kết quả trả về của hàm. Tuy nhiên, một trình biên dịch ngôn ngữ mệnh lệnh, ngôn ngữ Ada[1] chẳng hạn, thì đầu tiên phải chứng minh rằng kết quả của lời gọi thứ hai không phụ thuộc vào các biến đã bị thay đổi trong quá trình thực hiện bởi lời gọi thứ nhất.

---

[1] Ada là ngôn ngữ lập trình bậc cao được phát triển năm 1983 bởi Bộ Quốc phòng Mỹ (US Department of Defense), còn gọi là Ada 83, sau đó được phát triển bởi Barnes năm 1994, gọi là Ada 9X. Ngôn ngữ Ada lấy tên của nhà nữ Toán học người Anh, Ada Augusta Lovelace, con gái của nhà thơ Lord Byron (1788−1824). Người ta tôn vinh bà là người lập trình đầu tiên.

Một trình biên dịch song song sẽ gặp phải vấn đề tương tự nếu trình này muốn gọi hàm theo kiểu gọi song song.

Bên cạnh tính ưu việt, ta cũng cần xem xét những bất lợi vốn có của lập trình hàm : nhược điểm của ngôn ngữ hàm là thiếu các lệnh gán và các biến toàn cục, sự khó khăn trong việc mô tả các cấu trúc dữ liệu và khó thực hiện quá trình vào/ra dữ liệu.

Tuy nhiên, ta thấy rằng sự thiếu các lệnh gán và các biến toàn cục không ảnh hưởng hay không làm khó khăn nhiều cho việc lập trình. Khi cần, lệnh gán giá trị cho các biến được mô phỏng bằng cách sử dụng cơ cấu tham biến của các hàm, ngay cả trong các chương trình viết bằng ngôn ngữ mệnh lệnh.

Chẳng hạn ta xét một hàm *P* sử dụng một biến cục bộ *x* và trả về một giá trị có kiểu bất kỳ nào đó (`SomeType`). Trong ngôn ngữ mệnh lệnh, hàm `P` có thể làm thay đổi *x* bởi gán cho *x* môt giá trị mới. Trong một ngôn ngữ hàm, *P* có thể mô phỏng sự thay đổi này bởi truyền giá trị mới của *x* như là một tham đối cho một hàm phụ trợ thực hiện phần mã còn lại của *P*. Chẳng hạn, sự thay đổi giá trị của biến trong chương trình *P* :

```
function P(n: integer) -> SomeType ;
   x: integer := n + 7
   begin
   x := x * 3 + 1
   return 5 * g(x)
   end ;
```

ta có thể viết lại như sau :

```
function P(n : integer) -> SomeType ;
   x: integer := n + 7
   begin
   return Q(3*x + 1) % mô phỏng x := x * 3 + 1
   end ;
```

trong đó, hàm mới *Q* được định nghĩa như sau :

```
function Q(x: integer) -> Some Type
   begin
   return 5 * g(x)
   end ;
```

Ta cũng có thể sử dụng kỹ thuật này cho các biến toàn cục. Như vậy, việc mô phỏng lập trình mệnh lệnh trong một ngôn ngữ hàm không phải là cách mong muốn, nhưng có thể làm được.

Một vấn để nổi bật trong ngôn ngữ hàm là sự thay đổi một cấu trúc dữ liệu. Trong ngôn ngữ mệnh lệnh, sự thay đổi một phần tử của một mảng rất đơn giản. Trong ngôn ngữ hàm, một mảng không thể bị thay đổi. Người ta phải sao chép mảng, trừ ra phần tử sẽ bị thay đổi, và thay thế giá trị mới cho phần tử này. Cách tiếp cận này kém hiệu quả hơn so với phép gán cho phần tử.

Một vấn đề khác của lập trình hàm là khả năng hạn chế trong giao tiếp giữa hệ thống tương tác với hệ điều hành hoặc với người sử dụng. Tuy nhiên hiện nay, người ta có xu hướng tăng cường thư viện các hàm mẫu xử lý hướng đối tượng trên các giao diện đồ hoạ (GUI-Graphic User Interface). Chẳng hạn các phiên bản thông dịch họ Lisp như DrScheme, MITScheme, WinScheme...

Tóm lại, ngôn ngữ hàm dựa trên việc tính giá trị của biểu thức. Các biến toàn cục và phép gán bị loại bỏ, giá trị được tính bởi một hàm chỉ phụ thuộc vào các tham đối. Thông tin trạng thái được đưa ra tường minh, nhờ các tham đối của hàm và kết quả.

Sau đây ta sẽ xét những khái niệm được coi là cơ bản nhất trong các ngôn ngữ hàm : *hàm* (function), *danh sách* (lists), *kiểu* (type), *tính đa kiểu* (polymorphism), *các hàm bậc cao* (higher–order functions), *tham đối hóa từng phần* (Currying), *tính hàm theo kiểu khôn ngoan*[2] (lazy evaluation), *phương trình* (equations), *so khớp* (patterm matching).

Các hàm đệ quy (recursive functions) là một trong những khái niệm chính trong ngôn ngữ hàm. Các hàm bậc cao và phương pháp tính hàm theo kiểu khôn ngoan tạo thế mạnh cho lập trình hàm và đóng vai trò quan trọng trong việc xây dựng các chương trình hàm dạng đơn thể (modular functional programs). Tính đa kiểu bổ sung tính mềm dẻo cho hệ thống định kiểu.

Trước khi tìm hiểu ngôn ngữ Scheme bắt đầu từ chương 2, cuốn sách sử dụng ngôn ngữ Miranda để trình bày những khái niệm cơ bản của lập trình hàm. Miranda là một ngôn ngữ hàm có cú pháp dễ đọc, dễ hiểu do David Turner phát triển năm 1986. Đặc điểm của Miranda là *thuần tuý hàm* (purely functional), không xảy ra hiệu ứng phụ. Một chương trình Miranda, được gọi là một «script», là một tập hợp các *phương trình* (equation) được định nghĩa theo một thứ tự tuỳ ý nào đó (nói chung thứ tự không quan trọng).

Trình thông dịch Miranda chạy chế độ tương tác trong hệ điều hành Unix. Sau đây, chúng tôi không trình bày đầy đủ cú pháp của Miranda mà chỉ qua các ví dụ của Miranda để minh hoạ các yếu tố *thuần tuý hàm* của lập trình hàm.

Ví dụ sau đây la một chương trình Miranda đơn giản. Chú ý cặp ký hiệu `||` để bắt đầu một dòng chú thích của Miranda.

```
z = sq x / sq y    || z = sq(x)/sq(y) = x^2/y^2
sq n = n * n       || sq(n) = n^2
x = a + b
y = a - b
a = 10
b = 5
```

I.2.2. Định nghĩa hàm

Hàm là khái niệm cơ bản trong các ngôn ngữ hàm. Một hàm có thể nhận từ không đến nhiều tham đối vào để tính toán và trả về một giá trị, giá trị này chỉ phụ thuộc vào các tham đối đã nhận mà thôi.

Trong Miranda, một hàm được định nghĩa bởi hai phần : phần khai báo và phần định nghĩa hàm. Phần khai báo có thể vắng mặt có dạng một nguyên mẫu hàm :

<tên hàm> `::` *<miền xác định>* `->` *<miền giá trị>*

Phần định nghĩa hàm có dạng một phương trình, gồm vế trái và một số vế phải, mỗi vế phải có thể có một điều kiện đóng vai trò «lính gác» (guard) phân biệt đứng cuối :

<tên hàm> `[` *<danh sách tham đối>* `]` = *<biểu thức>* [*<điều kiện>*]

Ví dụ hàm đổi nhiệt độ từ độ Fahrenheit (F) sang độ Celsius (C) được định nghĩa trong Miranda như sau :

```
celsius :: num -> num    || khai báo kiểu hàm
celsius f = (f - 32)*5/9     || đổi từ độ Fahrenheit sang độ Celsius
```

[2] Trong cuốn sách có nhiều thuật ngữ tiếng Việt do tác giả tự dịch từ tiếng Anh.

Dòng đầu tiên khai báo `celsius` là một hàm nhận một đối số và trả về một giá trị số. Dòng thứ hai tính đổi nhiệt độ. Áp dụng hàm `celsius` đã định nghĩa cho các đối số cụ thể, ta có :

```
celsius 68
--> 20
celsius (-40)
--> -40
```

Ở đây, ký hiệu `-->` dùng để chỉ rằng giá trị trả về của hàm là những gì theo sau ký hiệu này. Chẳng hạn, lời gọi `celsius 68` trả về giá trị `20`.

Ngôn ngữ Miranda sử dụng cú pháp ngắn gọn và không sử dụng các cặp dấu ngoặc để bao bọc các tham đối hoặc bao bọc các lời gọi hàm.

Các hàm đệ quy đóng vai trò quan trọng trong các ngôn ngữ hàm. Chẳng hạn, xét một ví dụ cổ điển là tính ước số chung lớn nhất *gcd* (greatest common divisor) theo thuật toán Euclide :

```
gcd a b  = gcd (a-b) b,   if a > b
         = gcd a (b-a),   if a < b
         = a,             if a=b     || hoặc otherwise
```

Hàm `gcd` được định nghĩa nhờ phương trình có 3 vế phải phân biệt, mỗi vế phải chứa một điều kiện ($a > b$, $a < b$ và $a=b$ hay `otherwise`). Miranda sẽ thực hiện gọi đệ quy cho đến khi gặp điều kiện dừng là $a=b$ và cũng là kết quả tính `gcd`. Ta thấy các lính gác trong phương trình thực hiện tương tự lệnh `if` trong ngôn ngữ mệnh lệnh.

Trong các giáo trình nhập môn Tin học, người ta thường lấy ví dụ tính giai thừa của một số nguyên không âm bằng phương pháp đệ quy. Ta sẽ thấy ngôn ngữ Miranda tính giai thừa kiểu đệ quy như sau :

```
fac :: num -> num
fac n = 1, if n = 0                    || fac 0 = 1
fac x = x * fac (x - 1), otherwise     || lời gọi đệ quy
```

Để so sánh, chương trình dưới đây chỉ ra cách tính `n!` trong một ngôn ngữ mệnh lệnh nhưng không dùng đệ quy (chỉ dùng đệ quy khi thật cần thiết !).

```
function fac (n: integer) : integer ;
  var i, r: integer ;       % r chứa kết quả
begin
  r := 1
  { tính n * (n - 1) * (n - 2) * ... * 1 : }
  for i :=n downto 1 do r := r * i;
  return r
end;
```

Hàm `fac` trên đây sử dụng một biến trạng thái `r` bị thay đổi giá trị trong vòng lặp `for`. Còn chương trình Miranda không sử dụng vòng lặp `for`, mà sử dụng một lời gọi đệ quy. Mỗi lời gọi đệ quy cho `fac` tương ứng với một lần gặp của vòng lặp `for`. Do tính ngắn gọn của lời gọi đệ quy trong ngôn ngữ hàm, nhiều người ủng hộ lập trình hàm quan niệm rằng phép lặp trong các ngôn ngữ mệnh lệnh là rắc rối, làm người đọc khó hiểu, khó theo dõi.

Sau đây là một ví dụ khác tính nghiệm một phương trình bậc hai $ax^2+bx+c=0$:

```
quadsolve a b c
     = error "complex roots",    if delta<0
     = [-b/(2*a)],               if delta=0
     = [-b/(2*a)] + radix/(2*a),
     = [ -b/(2*a) - radix/(2*a)],                                    if
     where delta = b*b - 4*a*c and radix = sqrt delta
```

Mệnh đề của Miranda có thể lồng nhau (nested) nhiều mức.

I.2.3. Danh sách

Trong hầu hết các ngôn ngữ hàm, *danh sách* (list) là cấu trúc dữ liệu quan trọng nhất. Trong ngôn ngữ Miranda, một danh sách gồm từ *0* đến nhiều phần tử có cùng kiểu, ví dụ :

```
[1, 4, 9, 16]          || danh sách gồm 4 số nguyên
[1..10]                || danh sách 10 số nguyên (1..10)
['u', 'f', '0']        || danh sách gồm 3 ký tự
[]                     || danh sách rỗng (empty list) không có phần tử nào
[[2, 4, 6], [0, 3, 6, 9]]     || danh sách chứa hai danh sách  các số
                         nguyên
week_days = ["Mon","Tue","Wed","Thur","Fri"]
```

Các danh sách có thể lồng nhau (bao hàm nhau). Một điểm mạnh của Miranda và của một số ngôn ngữ hàm khác là cho phép định nghĩa một danh sách nhờ tính chất dễ nhận biết của các phần tử của nó, tương tự cách xác định một tập hợp trong Toán học. Một danh sách được định nghĩa như vậy được gọi là *nhận biết được* hay *ngầm hiểu* (list comprehension). Ví dụ, cách viết :

```
[ n | n <- [1..10] : n mod 2 = 1]
--> [1, 3, 5, 7, 9]
```

chỉ định một danh sách có thứ tự của các số lẻ giữa 1 và 10. Danh sách được hiểu là «gồm tất cả các số `n` nằm giữa `1` và `10` thoả mãn `n mod 2` bằng `1`». Một ví dụ khác về danh sách nhận biết được :

```
[ n*n | n <- [1..100]] ||  danh sách bình phương các số giữa 1 và 100
```

Dạng tổng quát của danh sách nhận biết được :

`[ body | qualifiers ]`

trong đó mỗi `qualifier` hoặc là một bộ tạo sinh phần tử (generator) có dạng như sau :

```
var <- exp
```

ký hiệu <– biểu diễn phần tử **`var`** thuộc (∈) một tập hợp chỉ định bởi `exp`, hoặc là một bộ lọc có dạng là một biểu thức lôgic thu hẹp miền xác định của các biến được tạo ra bởi bộ tạo sinh. Khi có nhiều thành phần `qualifier` cùng xuất hiện thì chúng được liệt kê cách nhau một dấu **`:`** (dấu hai chấm).

Ví dụ sau đây trả về danh sách các hoán vị của một danh sách đã cho :

```
perms [] = [[]]
perms L = [ a:T | a <- L : T <- perms (L--[a]) ]
```

Ta có thể định nghĩa lại hàm tính giai thừa nhờ hàm nhân `product` trong thư viện của Miranda :

```
fac n = product [1..n]
```

Tương tự, ta định nghĩa hàm tính tổng các số lẻ giữa 1 và 100 nhờ hàm `sum` trong thư viện của Miranda :

```
result = sum [1, 3..100]
```

Người ta xây dựng nhiều phép toán khác nhau trên các danh sách. Tùy theo ngôn ngữ hàm, một số phép toán có thể có sẵn (build-in), một số phép toán khác do người sử dụng tự định nghĩa. Các phép toán sơ cấp nhất là xây dựng (construction) và phân tách (decomposing) các danh sách. Chẳng hạn phép toán `hd` (head) lấy ra phần tử đầu tiên của một danh sách để trả về một phần tử. Phép toán `tl` (tail) trả về danh sách còn lại (sau khi đã lấy đi phần tử đầu tiên).

```
hd [10, 20, 30]
--> 10
tl [10, 20, 30]
--> [20, 30]
```

Phép toán `:` (cons) để chèn một phần tử vào trước các phần tử trong một danh sách :

```
5 : []
--> [5]
3 : [4, 5, 6]
--> [3, 4, 5, 6]
0 : [1, 2, 3]
--> [0,1, 2, 3]
```

Các phép toán khác là : phép `#` (dấu `#` đặt trước danh sách) đếm số phần tử là độ dài (length) một danh sách, phép ghép (concatenate) `++` (hai dấu cộng liên tiếp) các danh sách, phép `!` lấy từ danh sách ra một số phần tử (infix), hàm nghịch đảo `reverse` một danh sách, v.v...

```
[1, 2, 3] ++ [4, 5]
--> [1, 2, 3, 4, 5]   || ghép hai danh sách
days = week_days ++ ["Sat","Sun"]
days
--> ["Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun"]
# [3..7]
--> 5                  || độ dài một danh sách
#days
--> 7
days!0
--> "Mon"
```

Miranda còn có phép -- (hai dấu trừ liên tiếp) tính hiệu của hai tập hợp là hai danh sách :

```
[1,2,3,4,5] -- [2,4]
--> [1,3,5]
```

Để hiểu rõ hơn các lập trình trong Miranda, sau đây ta sẽ viết lại một số hàm thư viện của Miranda. Giả sử ta viết hàm `length` tính độ dài như sau :

```
length L = 0, if L = []      || danh sách rỗng có độ dài 0
```

|| danh sách khác rỗng có ít nhất một phần tử

```
length L = 1 + length (tl L), otherwise
```

Tương tự, hàm `concatenate` ghép hai danh sách được xây dựng như sau :

```
concatenate L1 L2 = L2, if L1 = []
concatenate L1 L2
       = (hd L1):concatenate (tl L1) L2, otherwise
```

Hàm `concatenate` đặc trưng cho kiểu lập trình đệ quy của các ngôn ngữ hàm. Dòng đầu tiên chỉ cách thoát ra khỏi hàm, theo kiểu *chia để trị* (divide and conquer). Trong trường hợp ghép một danh sách rỗng (`if L1 = []`) với một danh sách bất kỳ khác (`L2`) thì trả về kết quả chính là danh sách này và quá trình kết thúc.

Dòng thứ hai là trường hợp tổng quát về ghép hai danh sách thành một danh sách mới sử dụng phép toán cons (`otherwise`) theo kiểu đệ quy. Do danh sách thứ nhất `L1` ≠ `[]` nên `L1` có ít nhất một phần tử. Khi đó danh sách mới (là danh sách kết quả) được tạo ra bởi phần tử đầu tiên của `L1`, là (`hd L1`) ghép với (phép `:`) một danh sách còn lại. Danh sách còn lại này lại là kết quả của bài toán ban đầu nhưng nhỏ hơn với tham đối thứ nhất là danh sách còn lại của `L1` và tham đối thứ hai là toàn bộ `L2`. Quá trình cứ thế tiếp tục cho đến khi danh sách còn lại rỗng.

Ta có thể so sánh cách xây dựng các danh sách trong các ngôn ngữ mệnh lệnh với ngôn ngữ Miranda.

Trong các ngôn ngữ mệnh lệnh, danh sách do người lập trình tự xây dựng thường theo kiểu móc nối sử dụng biến con trỏ (pointer) và sử dụng bộ nhớ cấp phát động (dynamic allocation). Các phép toán trên danh sách là sự thao tác ở mức thấp trên các con trỏ. Việc cấp phát và giải tỏa bộ nhớ cũng do người sử dụng tự giải quyết. Thông thường, người lập trình có xu hướng sử dụng việc cập nhật phá hủy (destructive updates) trên các cấu trúc dữ liệu.

Ví dụ, người lập trình Ada ghép danh sách bằng cách viết một thủ tục cho phép móc nối (hook) hai danh sách với nhau nhờ một phép gán con trỏ, mà không cần xây dựng một danh sách mới (thứ ba). Tuy nhiên tiếp cận kiểu Ada có thể gây ra hiệu ứng phụ nếu làm thay đổi danh sách thứ nhất.

Trong các ngôn ngữ hàm, người lập trình không hề quan tâm đến các con trỏ và việc cấp phát bộ nhớ. Mặc dầu những yếu tố này là tồn tại trong ngôn ngữ hiện hành, nhưng chúng được che dấu một cách hiệu quả đối với người lập trình tạo nên một mức trừu tượng có lợi cho người lập trình, giúp họ tránh được những sai sót khi lập trình.

Trong những tình huống như vậy, một *hệ thống thời gian thực hiện* RST (Run Time System) quản lý bộ nhớ lưu trữ các phần tử của danh sách. RTS cho phép cấp phát bộ nhớ khi cần thiết và tự động phát hiện những khối nhớ không còn cần dùng đến nữa để giải phóng chúng.

Ngôn ngữ Miranda còn có kiểu dữ liệu `tuple` tương tự kiểu bản ghi của Pascal. Đó là một dãy phần tử không có cùng kiểu (danh sách là một dãy phần tử có cùng kiểu tương tự kiểu mảng array) được viết giữa một cặp dấu ngoặc. Ví dụ :

```
employee = ("Kim",True, False , 29)
```

Các phần tử kiểu `tuple` không có thứ tự. Việc truy cập đến một phần tử chỉ có thể được thực hiện nhờ phép so khớp (pattern matching).

I.2.4. Phép so khớp

Như đã thấy, một hàm trong Miranda có thể được định nghĩa bằng cách sử dụng nhiều biểu thức vế phải khác nhau, theo sau là «lính gác», đó là các điều kiện như `if x>0`, `otherwise` hay `while`. Tuy nhiên, Miranda cho phép sử dụng một cú pháp tổng quát để lựa chọn giữa các khả năng dựa trên các mẫu (patterns) dạng :

<patterm> = <expression>, <condition>

trong đó, phần *<condition>* là tùy chọn. Mẫu có thể sử dụng các tham biến hình thức, các hằng và một số cấu trúc khác. Khi một hàm được gọi, lời gọi hàm gồm tên hàm và các tham đối thực sự của nó (nếu có) sẽ được so khớp với các mẫu trong chương trình. Nếu so khớp thành công, dòng lệnh tương ứng với mẫu được thực hiện. Nếu có nhiều mẫu được tìm thấy, mẫu đầu tiên (theo thứ tự viết các lệnh) sẽ được chọn.

Sau đây là một số ví dụ đơn giản minh họa cách định nghĩa các hàm sử dụng phép so khớp. Hàm `cond` đã xét có thể viết dưới dạng :

```
cond true x y = x
cond false x y = y
```

Hàm `fac` có thể viết với hai dòng chương trình :

```
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

Phép so khớp thứ nhất xảy ra nếu tham đối bằng 0. Mẫu thứ hai được so chỉ khi nếu tham đối là lớn hơn hoặc bằng 1. Khi đó giá trị của *n* được lấy giá trị của tham đối trừ đi 1 một cách «khôn ngoan».

Trường hợp tổng quát, một biểu thức số xuất hiện trong một mẫu có thể có dạng *V* + *C*, với *V* là một biến và *C* là một trực hằng (literal constant). Mẫu sẽ chỉ được so nếu *V* có thể nhận một giá trị không âm. Hàm `ackerman` được định nghĩa như sau :

```
ack 0 n = n+1
ack (m+1) 0 = ack m 1
ack (m+1) (n+1) = ack m(ack (m+1) n)
```

Hàm tính số fibonacci thứ *n* :

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Chú ý : Các biểu thức số tổng quát không thể được lấy làm các mẫu để tránh nhập nhằng (ambiguities). Ví dụ, mẫu :

```
f (n + m) = n * m
```

là nhập nhằng và không đúng. Một lời gọi hàm, `f 9` chẳng hạn, sẽ gây ra kết quả hoặc `1*8` hoặc `2*7`, hoặc một tổ hợp khác tuỳ ý.

Mẫu cũng có thể chứa danh sách. Ví dụ, hàm tính độ dài của một danh sách đã cho trước đây có thể được viết lại dưới dạng như sau :

```
length [] = 0
length (a : L) = 1 + (length L)
```

Mẫu xuất hiện trong dòng thứ nhất tương ứng với danh sách rỗng, chỉ có danh sách rỗng mới thể được so khớp để có kết quả 0. Dòng thứ hai xử lý danh sách khác rỗng. Trong trường hợp này, một lời gọi đệ quy được tạo ra. Biến *a* được khớp với phần tử đầu tiên của danh

sách tham đối và do đó, *L* là danh sách còn lại trở thành tham đối mới của hàm. Kết quả lời gọi đệ quy đã giảm đi 1.

Một số hàm xử lý danh sách khác sử dụng mẫu so khớp : tính tổng, tích và nghịch đảo một danh sách :

```
sum [] = 0
sum (a:L) = a + sum L

product [] = 1
product (a:x) = a * product x

reverse [] = []
reverse (a:L) = reverse L ++ [a]
```

Để truy cập đến một phần tử của kiểu bản ghi (tuple), sử dụng so khớp, chẳng hạn bản ghi có hai phần tử :

```
fst (a, b) = a ||  truy cập đến một phần tử thứ nhất
snd (a, b) = b ||  truy cập đến một phần tử thứ hai
```

Sau đây định nghĩa lại hai hàm thư viện của Miranda : hàm `take` trả về danh sách `n` phần tử đầu tiên của danh sách đã cho và hàm `drop` trả về phần còn lại của danh sách sau khi đã loại bỏ `n` phần tử đầu tiên.

```
take 0 L = []
take (n+1) [] = []
take (n+1) (a:L) = a : take n L

drop 0 L = L
drop (n+1) [] = []
drop (n+1) (a:L) = drop n L
```

Chú ý rằng hai hàm được định nghĩa sao cho đồng nhất thức sau đây thoả mãn (bao gồm cả trường hợp oái oăm là độ dài của `L` nhỏ thua *n*) :

```
take n L ++ drop n L = L
```

I.2.5. Phương pháp currying (tham đối hoá từng phần)

Một yếu tố quan trọng khác của hầu hết các ngôn ngữ hàm là phương pháp currying (lấy tên nhà logic học Haskell B. CURRY.), hay còn được gọi là phương pháp *tham đối hoá từng phần* (partial parametrization).

Thông thường một hàm được thực hiện theo kết hợp trái, nếu viết $f\ x\ y$ (hàm f tác động lên hai đối $x\ y$), viết quy ước $(x, y) \rightarrow f(x, y)$, thì cũng được xem như viết $(f\ x)\ y$, nghĩa là kết quả của việc áp dụng f cho x là một hàm để áp dụng cho y. Ta viết $x \rightarrow (y \rightarrow f\,(x, y))$.

Một cách tổng quát cho hàm n biến $f\,(x_1, x_2, ..., x_n)$:

$x_1 \rightarrow (x_2 \rightarrow (x_3 \rightarrow ... (x_n \rightarrow f\,(x_1, ..., x_n)) ...))$

Chẳng hạn xét hàm `mult` được định nghĩa như sau :

```
mult x y = x * y
```

Nếu hàm `mult` được gọi với hai đối số, thì `mult` sẽ tính tích số của hai đối số này, theo nghĩa thông thường. Tuy nhiên trong ngôn ngữ Miranda, `mult` có thể được xem như hàm một tham đối (đối thứ nhất `x`), kết quả sẽ là một hàm khác, áp dụng cho một tham đối (đối thứ hai `y`).

Một cách tổng quát, một hàm nào đó có nhiều hơn một tham đối có thể được tham đối hóa từng phần. Chẳng hạn ta định nghĩa hàm `triple` để nhân 3 một số :

```
triple x = 3 * x
```

Nhưng cũng có thể định nghĩa lại hàm `triple` theo cách currying :

```
triple = mult 3
```

Lời gọi :

```
triple 7
```

cho kết quả là `21` vì dẫn đến lời gọi `mult 3 7`.

I.2.6. Khái niệm về bậc của hàm

Trong phần lớn các ngôn ngữ mệnh lệnh, các đối tượng cơ sở (biến và hằng) được xử lý khác với hàm. Chúng có thể được đọc vào (từ bàn phím, từ tệp...), đưa ra (màn hình, máy in, tệp...), trao đổi giá trị với nhau, nhưng hàm chỉ có thể được gọi hoặc được kích hoạt (invoke).

Các đối tượng cơ sở được gọi là có bậc *0*, các hàm có các tham đối là những đối tượng cơ sở được gọi là các hàm bậc *1*. Một hàm bậc *n* nhận các tham đối bậc nhỏ hơn *n*, để có các biểu thức bậc cao hơn.

Khái niệm về bậc được Russel (Russell's Paradox) và Whitehead (Principia Mathematica ~1900) đề xuất nhằm loại bỏ những *nghịch lý* (paradox) thường hay gặp trong lý thuyết tập hợp của Cantor, chẳng hạn nghịch lý về người thợ cạo (barber paradox), bằng cách làm giảm tất cả các khả năng xảy ra vòng luẩn quẩn. Trong lôgic vị từ bậc một, chỉ có thể sử dụng các lượng tử áp dụng cho các cá thể (các biến). Còn trong lôgic vị từ bậc hai, người ta có thể lượng tử hoá các vị từ...

Trong các ngôn ngữ hàm, hàm luôn luôn được xử lý như những đối tượng được truyền tham đối để trả về kết quả và được lưu giữ trong các cấu trúc dữ liệu. Một hàm nhận một hàm khác như là một tham đối được gọi là hàm bậc cao. Các hàm bậc cao mang lại tính hiệu quả và là nền tảng (corner-stone) của lập trình hàm.

Để hiểu bản chất của một hàm bậc cao, người đọc có thể tự tìm giá trị kết quả của `answer` từ các lệnh Miranda sau đây :

```
answer = twice twice twice suc 0
twice f x = f(f x)
suc x = x + 1
```

Ngôn ngữ Miranda luôn xem những hàm có nhiều hơn một tham đối đều là các hàm bậc cao. Chẳng hạn từ hàm thư viện `member x L` kiểm tra danh sách `L` có chứa phần tử `x` không (kết quả là `true` hoặc `false`), ta có thể áp dụng để định nghĩa các hàm khác nhau như sau :

```
vowel = member ['a', 'e', 'i', 'o', 'u']
digit =
   member ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
month =
   member [ "Jan", "Feb", "Mar", "Apr", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec" ]
```

Các ngôn ngữ hàm đều có phép toán «bao qua phải» (foldr right) dùng để tính toán trên các phần tử của một danh sách. Miranda có hàm `foldr` được định nghĩa như sau :

```
foldr op k [] = k
foldr op k (a:L) = op a (foldr op k L)
```

Lời gọi hàm `foldr` :

```
foldr (+) 0 [2, 4, 7]
```

tương đương với :

```
(2 + (4 + (7 + 0)))
--> 13
```

Chú ý các phép toán trung tố trong ngôn ngữ Miranda, như phép cộng + trên đây, được xem là tham đối và phải được đặt trong một cặp dấu ngoặc.

Ta có thể định nghĩa các hàm xử lý danh sách sử dụng hàm `foldr`. Chẳng hạn hàm tính tổng tất cả các phần tử của một danh sách theo kiểu currying như sau :

```
sumlist = foldr (+) 0
```

Hàm `sumlist` có một tham đối là danh sách sẽ cung cấp ba tham đối cho hàm `foldr` nhưng kết quả là do hàm `op` có một tham đối trả về.

```
sumlist [2, 9, 52]
--> foldr (+) 0 [2, 9, 52]
--> 63
```

Một cách tương tự ta định nghĩa hàm tính tích của tất cả các phần tử trong một danh sách :

```
product = foldr (*) 1
```

Hàm :

```
and = foldr (&) true
```

gọi phép « và» lôgích cho tất cả các phần tử của một danh sách. Kết quả phép gọi hàm `and L` là `true` nếu mọi phần tử của danh sách `L` đều là `true`, kết quả là `false` nếu ngược lại.

Hàm nghịch đảo một danh sách :

```
reverse = foldr postfix []
            where postfix a L = L ++ [a]
```

Các ngôn ngữ hàm có nhiều hàm bậc cao tiền định, và người lập trình cũng có thể tự định nghĩa những hàm bậc cao mạnh và tổng quát như hàm `foldr` vừa xét.

Một hàm bậc cao quan trọng khác là `map f L` với `f` là một hàm và `L` là một danh sách. Hàm `map` trả về kết quả là một danh sách mới gồm các phần tử của `L` đã được áp dụng cho hàm `f`. Ví dụ :

```
map triple [1, 2, 3, 4]
--> [triple 1, triple 2, triple 3, triple 4]
--> [3, 6, 9, 12]
```

Hàm `map` gọi `triple` cho tất cả các phần tử của danh sách `[1, 2, 3, 4]`, để trả về danh sách các số nguyên đã được nhân ba là `[3, 6, 9, 12]`.

I.2.7. Kiểu và tính đa kiểu

Các ngôn ngữ hàm có một hệ thống kiểu dữ liệu hoàn toàn khác với các ngôn ngữ mệnh lệnh. Trong các ngôn ngữ mệnh lệnh (như Pascal, C, Ada..), tính *định kiểu tĩnh chặt chẽ* (static strong typing) bắt buộc người lập trình phải mô tả kiểu cho mỗi biến dùng đến. Sau khi khai báo, người sử dụng không được thay đổi kiểu dữ liệu của biến trong khi chạy chương trình.

Các ngôn ngữ hàm thường không sử dụng định kiểu. Tuy nhiên một số ngôn ngữ hàm lại sử dụng *định kiểu động* (dymamic typing) hoặc *định kiểu ẩn chặt chẽ* (implicit strong typing) (thường gặp trong ngôn ngữ Fortran).

Như vậy, các ngôn ngữ hàm tạo ra được tính mềm dẻo về *các hàm đa kiểu* (polymorphic functions) với lời gọi có các tham đối có các kiểu dữ liệu khác nhau.

Miranda sử dụng chế độ định kiểu tĩnh chặt chẽ tường minh (implicit static strong typing). Mọi biến mọi biểu thức đều có một kiểu được xác định tĩnh. Có ba kiểu tiền định trong Miranda là `num`, `bool` và `char`. Kiểu `num` không phân biệt số nguyên hay số thực. Kiểu `bool` có hai giá trị hằng là `true` và `false`. Kiểu `char` gồm các ký tự ASCII. Một hằng ký tự được đặt giữa cặp dấu nháy đơn và sử dụng quy ước tương tự ngôn ngữ C, như `'x'`, `'&'`, `'\n'`, v.v...

Nếu `T` là một kiểu, thì `[T]` là một kiểu danh sách gồm các phần tử kiểu `T`. Ví dụ danh sách sau đây :

```
[[1, 2], [2, 3], [4, 5]]
```

có kiểu `[[num]]` vì đây là một danh sách gồm các phần tử là danh sách các số. Một hằng chuỗi có dạng `[ char ]`, chẳng hạn chuỗi «bonjour» được viết trong Miranda `['b', 'o', 'n', 'j', 'o', 'u', 'r']`.

Nếu `T1`, ..., `Tn` là các kiểu nào đó, `n>0`, thì `(T1, ..., Tn)` là một kiểu bản ghi, hay bộ *n* phần tử bất kỳ. Ví dụ bộ 3 `(true, "Honda", 7331)` có kiểu `(bool, [char], num)`.

Nếu `T1` và `T2` là hai kiểu nào đó, thì `T1->T2` là một kiểu hàm với tham đối có kiểu `T1` và giá trị trả về có kiểu `T2`. Ví dụ hàm `triple` vừa định nghĩa trên đây có kiểu :

```
num -> num
```

Chú ý phép `->` có kết hợp phải (right associative).

Người lập trình có thể sử dụng các khai báo định nghĩa hàm. Ví dụ, hàm `mult` được khai báo kiểu currying như sau :

```
mult :: num -> (num -> num)
```

Có nghĩa rằng `mult` được xem là hàm có một tham đối trả về một hàm khác có kiểu :

```
num -> num.
```

Dấu `::` trong khai báo hàm được đọc «có kiểu là » (is of type). Ví dụ :

```
sq :: num -> num
sq n = n * n
```

Tuy nhiên, việc khai báo kiểu hàm là không cần thiết. Các ngôn ngữ hàm thường có khả năng uy diễn kiểu tự động nhờ một bộ kiểm tra kiểu (type checker). Khi không khai báo kiểu hàm, chẳng hạn nếu chỉ định nghĩa hàm `triple` bởi :

```
triple x = 3 * x
```

thì bộ kiểm tra kiểu sẽ suy ra rằng `x` phải là một số, vì `x` là một thừa số trong một phép nhân. Bộ kiểm tra kiểu cũng suy ra được rằng kết quả của `triple x` phải là một số, do đó kiểu của hàm này phải là :

```
triple :: num -> num
```

Tính đa kiểu là yếu tố rất quan trọng trong nhiều ngôn ngữ hàm. Với mỗi *tham biến hình thức* (formal parameters), một hàm đa kiểu có thể chấp nhận lời gọi tương ứng với nhiều *tham đối thực sự* (actual parameters) có các kiểu khác nhau. Khác với các thủ tục trong Ada, một hàm đa kiểu là một *hàm đơn* (single function), không phải là các *hàm bội* (multiple functions) có cùng tên. Ví dụ xét hàm đa kiểu `pair` sau đây trả về một danh sách từ hai tham đối là hai phần tử :

```
pair x y = [x, y]
```

Hàm này có thể được sử dụng như sau :

```
pair 1 2
--> [1, 2]
pair true false
--> [true, false]
pair [] [2]
--> [[], [2]]
```

Hàm `pair` được định nghĩa đơn, nhưng có thể sử dụng nhiều kiểu tham đối khác nhau, như `bool`, `num`, `[num]`, v.v..., sao cho cả hai tham đối đều phải có cùng một kiểu. Như vậy, kiểu của hàm `pair` không thể được diễn tả như kiểu của hàm `triple` vừa định nghĩa ở trên, vì rằng các kiểu của các tham đối trong `pair` là không cố định. Người ta đưa ra giải pháp sử dụng *các biến kiểu đặc thù* (generic type variables). Bên trong một khai báo kiểu, một biến kiểu có thể chỉ định một kiểu nào đó, cùng một biến kiểu cho phép chỉ định cùng một kiểu xuyên suốt cả khai báo.

Miranda sử dụng các ký hiệu `*`, `**`, `***` để chỉ định kiểu tuỳ ý. Ví dụ hàm đồng nhất `id` (identity) được định nghĩa như sau :

```
id x = x
```

Hàm `id` có khai báo kiểu như sau :

```
id :: * -> *
```

có nghĩa là `id` có nhiều kiểu. Sau đây là một số khai báo kiểu của các hàm đã được định nghĩa trên đây :

```
fac :: num -> num
ack :: num -> num-> num
sum :: [num] -> num
month :: [char] -> bool
reverse :: [*] -> [*]
fst :: (*, **) -> *
snd :: (*, **)-> **
foldr :: (*-> ** -> **) -> ** -> [*] -> **
perms :: [*] -> [[*]]
```

Các quy tắc định nghĩa kiểu như trên làm tránh được sai sót khi sử dụng hàm không đúng, chẳng hạn :

```
pair 1 true        || các tham đối phải cùng kiểu
5 + (pair 1 2)     || kiểu kết quả là một danh sách, không là một số.
```

Trong ví dụ thứ hai, Miranda xem rằng kiểu của `(pair 1 2)` là một danh sách các số, nên không thể dùng để làm số hạng cho một phép cộng.

Dù các hàm có thừa nhận các tham đối khác kiểu và không cần khai báo kiểu một cách tường minh, hệ thống kiểu vừa mô tả trên đây là chặt chẽ và ở dạng tĩnh. Sự vi phạm các quy tắc định kiểu sẽ dẫn đến các thông báo lỗi khi dịch (compile time error messages).

Tuy nhiên việc thêm các khai báo kiểu hàm thường làm cho bộ kiểm tra kiểu dễ vi phạm sai sót khi sử dụng hàm.

I.2.8. Tính hàm theo kiểu khôn ngoan

Thông thường, khi gọi tính giá trị một hàm, các tham đối được tính giá trị trước tiên, sau đó mới tiến hành thực hiện tính toán trong hàm. Chẳng hạn, lời gọi :

```
mult (fac 3) (fac 4)
```

yêu cầu tính giá trị các hàm giai thừa `(fac 3)` và `(fac 4)` trước, theo một thứ tự tùy ý, sau đó mới thực hiện hàm nhân :

```
mult 6 24
```

để cho kết quả cuối cùng là 144. Cách viết rút gọn biểu thức lúc đầu thành dạng đơn giản hơn được gọi là *phép rút gọn theo thứ tự áp dụng* (applicative order reduction). Chú ý rằng phép rút gọn được bắt đầu từ các biểu thức trong cùng nhất (innermost expressions).

Tuy nhiên có một cách khác để rút gọn một biểu thức là bắt đầu từ biểu thức ngoài nhất (outermost expression) và không tính giá trị các biểu thức con (subexpressions) cho đến khi cần dùng đến kết quả của chúng.

Người ta gọi cách này là *rút gọn theo thứ tự thường* (normal order reduction). Chẳng hạn biểu thức :

```
mult (fac 3) (fac 4)
```

sẽ được rút gọn thành :

```
(fac 3) * (fac 4)
```

sau đó thành :

```
6 * 24
```

để nhận được kết quả 144.

Những người lập trình có kinh nghiệm trên các ngôn ngữ mệnh lệnh có thể nghĩ rằng phép rút gọn theo thứ tự áp dụng là thuận tiện hơn, nhưng cách này cũng có những điều bất tiện.

Giả sử rằng ta muốn xây dựng hàm `cond` nhận một giá trị `bool b` như là tham đối thứ nhất và trả về tham đối thứ hai của nó nếu `b` có giá trị `true` hoặc trả về tham đối thứ ba nếu `b` là `false` :

```
cond b x y = x, if b
cond b x y = y, otherwise
```

hay gọn hơn :

```
cond true x y = x
cond false x y = y
```

Nếu cả ba tham đối của `cond` được tính giá trị trước khi hàm `cond` được thực hiện, thì sẽ xảy ra hai trường hợp :

- Một trong các tham đối của `cond` tham gia tính toán một cách vô ích, dẫn đến kết quả sai, chẳng hạn `cond (x=0) 0 (1/x)` với `x=0`.
- Nếu việc tính toán biểu thức tham đối là vô ích và không kết thúc thì sẽ gây ra toàn bộ biểu thức bị tính lặp vô hạn lần.

Trường hợp thứ hai có thể được minh họa qua ví dụ sau : giả sử ta cũng sử dụng hàm `cond` cho một định nghĩa khác của hàm `fac` :

```
fac n = cond (n = 0) 1 (n * fac (n - 1))
```

Nếu tham đối thứ ba của `cond` luôn luôn được tính, thì hàm `fac` sẽ không bao giờ dừng. Bởi vì lời gọi `fac 1` kéo theo lời gọi `fac 0`, lời gọi `fac 0` sẽ kéo theo lời gọi `fac – 1` và cứ thế tiếp tục.

Những vấn đề trên đã dẫn đến khái niệm tính giá trị theo kiểu khôn ngoan trong lập trình hàm sử dụng phép rút gọn theo thứ tự thường. Ý tưởng của phương pháp tính giá trị hàm theo kiểu khôn ngoan là chỉ tính giá trị các tham đối của một hàm khi các giá trị của chúng là cần thiết, những biểu thức nào không cần thiết thì bỏ qua.

Sử dụng tính giá trị hàm theo kiểu khôn ngoan, phép tính hàm `fac 1` sẽ được rút gọn như được trình bày dưới đây. Việc tính giá trị kết thúc và cho kết quả đúng. Ơ đây ta sử dụng biểu thức `if` dạng giả ngữ (pseuclo-notation). Tham đối thứ ba của `cond` sẽ không được tính nếu `n` bằng `0`, vì rằng trong trường hợp này, không cần tính tham đối nữa.

```
fac 1
--> cond (1=0) 1 (1*fac (1-1))          || gọi fac
--> if (1=0) then 1 else (1*fac (1-1))  || gọi cond
--> if false then 1 else (1*fac (1-1))  || tính 1 = 0
--> 1*fac (1-1)
--> 1*(cond (1-1 = 0) 1 ((1-1)-1)))     || gọi fac
--> 1*((if 1-1=0) then 1 else ((1-1)*fac ((1-1)-1)))
                                        || gọi cond
--> 1*((if true then 1 else ((1-1)*fac ((1-1)-1)))
                                        || tính 1-1 = 0
--> 1*1                                 || tính 1*1
--> 1
```

Phương pháp tính hàm kiểu khôn ngoan còn thể hiện một lợi ích quan trọng khác : cho phép định nghĩa các cấu trúc dữ liệu vô hạn theo kiểu ý niệm (conceptually infinite). Chẳng hạn trong Miranda, danh sách các số nguyên dương có thể được định nghĩa :

```
[1.. ]
```

Danh sách theo kiểu định nghĩa này có thể tham gia tính toán như mọi danh sách khác. Chẳng hạn :

```
hd [1.. ]
--> 1              || phần tử thứ nhất
hd (tl [1.. ])
```

```
--> 2              || phần tử thứ hai
hd (tl (map triple [1.. ]))
--> 6              || phần tử thứ hai của danh sách đã được nhân 3
```

Thực tế, ngôn ngữ Miranda chỉ cho phép xây dựng các danh sách hữu hạn. Chỉ khi toàn bộ các phần tử của danh sách được yêu cầu, như là :

```
sumlist [1..]      || cộng dồn tất cả các số nguyên dương
```

hoặc :

```
#[1.. ]            || tính độ dài của danh sách vô hạn
```

sẽ làm cho hệ thống rơi vào một vòng lặp vô hạn (hoặc gây tràn bộ nhớ). Sau đây là một số ví dụ khác :

```
ones = 1 : ones
repeat a = L
     where L = a : L
nats = [0.. ]
odds = [1, 3.. ]
squares = [ n*n | n <- [0.. ] ]
perfects = [ n | n <- [1.. ] : sum(factors n) = n ]
|| Tìm các số nguyên tố n ≥ 2
primes = sieve [ 2.. ]
     where sieve (p:x)=p : sieve [n|n<-x: n mod p>0 ]
```

Mặc dù phương pháp tính giá trị hàm theo kiểu khôn ngoan tỏ ra có nhiều ưu thế nhưng thực tế, việc cài đặt rất tốn kém. Một vấn đề xảy ra là chiến lược rút gọn theo thứ tự thường có thể tính một biểu thức mất nhiều lần, trong khi phép rút gọn theo thứ tự áp dụng chỉ tính giá trị các biểu thức đúng một lần. Chẳng hạn, từ định nghĩa hàm `double` sau đây :

```
double x = x + x
```

Phép rút gọn theo thứ tự thường tính biểu thức :

```
double 23 * 45
```

như sau :

```
double 23 * 45
--> 23 * 45 + 23 * 45
--> 1035 + 23 * 45
--> 1035 + 1035
--> 2070
```

Trong khi đó, phép rút gọn theo thứ tự áp dụng lại tính đơn giản hơn và hiệu quả hơn :

```
double 23 * 45
--> double 1035
--> 1035 + 1035
--> 2070
```

Vấn đề này có thể được giải quyết bằng cách sử dụng kỹ thuật đồ thị rút gọn (graph reduction). Đây là kỹ thuật hoặc không tính, hoặc tính giá trị các biểu thức đúng một lần, nhưng không tính nhiều lần.

Một vấn đề khác là tính giá trị của các tham đối trước khi gọi hàm. Thực tế cho thấy việc tính trước tham đối dễ dàng cài đặt hơn là trì hoãn thực hiện chúng. Tuy nhiên, điều này làm

tổng chi phí tính giá trị hàm theo kiểu khôn ngoan cao hơn bình thường. Hiện nay, người ta đang phát triển các phương pháp tối ưu nhằm giảm tổng chi phí này.

Không phải mọi ngôn ngữ hàm đều có kiểu tính khôn ngoan như vừa trình bày. Những ngôn ngữ không có khả năng này cho các hàm do người sử dụng tự xây dựng (user defined functions) thường có sẵn một vài hàm với ngữ nghĩa khôn ngoan (ví dụ hàm `cond`). Các ngôn ngữ mệnh lệnh cũng có những yếu tố mang ngữ nghĩa khôn ngoan được xây dựng sẵn, chẳng hạn lệnh `if`. Ngôn ngữ C và Ada có các phép logic (dạng short cut) `and` và `or` đặc biệt cho phép tính giá trị tham đối thứ hai chỉ khi cần thiết.

I.2.9. Một số ví dụ

Sau đây ta xét một số ví dụ khác viết trong Miranda để minh họa những đặc trưng cơ bản của lập trình hàm vừa được trình bày trên đây : loại bỏ những phần tử trùng nhau trong một danh sách đã được sắp xếp thứ tự, sắp xếp nhanh một danh sách (quicksort), giải bài toán 8 quân hậu và tìm dãy số Hamming.

1. Loại bỏ những phần tử trùng nhau

Ta cần xây dựng hàm `uniq` nhận một danh sách các phần tử đã được sắp xếp để trả về một danh sách đã loại bỏ những phần tử trùng nhau đứng trước, chỉ giữ lại một phần tử đứng sau cùng. Ta có chương trình như sau :

```
uniq [] = []          || cửa thoát (escape hatch) : danh sách rỗng
uniq (a:(a:L)) = uniq (a:L) || bỏ một trong hai phần tử đầu tiên bằng nhau
uniq (a:L) = a : uniq L     || chỉ xét danh sách đã bỏ đi phần tử đầu
```

Dòng chương trình thứ nhất tầm thường : danh sách rỗng xem như đã được xử lý. Dòng thứ hai sử dụng khả năng thể hiện một biến (`a`) hai lần trong cùng một vế trái. Một khi dòng này được tiếp cận, thì một lời gọi đệ quy sẽ xuất hiện với tham đối là danh sách ban đầu nhưng đã loại bỏ phần tử đầu (head) do trùng nhau. Chẳng hạn danh sách `[3, 3, 4]` chỉ còn lại `[3, 4]` phải xử lý. Nếu như cả hai dòng chương trình đầu bị bỏ qua thì dòng thứ ba được chọn. Dòng này này cũng tạo ra một lời gọi đệ quy với phần tử đầu tiên của danh sách là cái ra của nó. Chẳng hạn danh sách `[3, 4, 5]` dẫn đến chỉ còn phải xử lý `[4, 5]`.

Cách định nghĩa hàm `uniq` trên đây sử dụng các mẫu so khớp. Một cách khác là sử dụng các điều kiện «lính canh» ở vế phải nhưng như vậy sẽ làm chương trình trở nên dài dòng hơn.

Cuối cùng, ta thấy hàm `uniq` là đa kiểu. Hàm có thể được gọi với nhiều kiểu khác nhau của tham đối thực sự. Chẳng hạn :

```
uniq [3, 3, 4, 6, 6, 6, 6, 7]
--> [3, 4, 6, 7]
uniq ['a', 'b', 'b', 'c']
--> ['a', 'b', 'c']
```

2. Sắp xếp nhanh quicksort

Thuật toán quicksort sắp xếp nhanh các phần tử của một danh sách (thường là các số) theo thứ tự không giảm. Để sắp xếp, đầu tiên quicksort xác định giá trị của phần tử đầu tiên của danh sách, gọi là `x`. Tiếp theo, quicksort tạo ra hai danh sách con, một danh sách chứa các giá trị nhỏ hơn hoặc bằng `x`, và một danh sách chứa giá trị lớn hơn `x`. Mỗi một danh sách được sắp xếp một cách đệ quy (bằng cách gọi lại quicksort). Cuối cùng, các danh sách kết

quả và giá trị x được ráp lại với nhau và trả về kết quả là danh sách đã được sắp xếp. Phần tử được gọi là phần tử trục (pivot).

Thuật toán quicksort được viết như sau :

```
quicksort [] = []          || danh sách rỗng coi như đã được sắp xếp
quicksort (x:Tail)         || trường hợp tổng quát :
   = quicksort [a | a<-Tail: a<=x ]    || danh sách con thứ nhất
   ++ [x]                              || phần tử đầu (head)
   ++ quicksort [a | a<-Tail: a>x ]    || danh sách con thứ hai
```

Dòng thứ nhất của chương trình xử lý danh sách rỗng dùng để kết thúc phép đệ quy. Dòng thứ hai là trường hợp tổng quát, cho phép xây dựng hai danh sách con sử dụng danh sách nhận biết. Ví dụ :

```
[a | a <- Tail : a < = x]
```

là danh sách tất cả các phần tử a của danh sách Tail có giá trị nhỏ hơn hoặc bằng x. Tiếp theo quicksort gọi đệ quy cho cả hai danh sách con. Các kết quả và danh sách [x] chỉ chứa mỗi phần tử x được ráp lại với nhau bằng cách sử dụng phép ghép danh sách ++.

Thuật toán quicksort minh họa kiểu lập trình đệ quy tổng quát trong Miranda, cũng như phản ánh tính ưu việt của các danh sách tự nhận biết. Mặt khác, hàm quicksort là đa kiểu, có dạng :

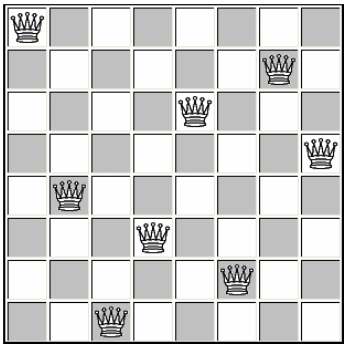
```
quicksort :: [a] -> [a]
```

Nghĩa là quicksort nhận một danh sách có kiểu bất kỳ làm tham đối và trả về một danh sách các phần tử đã sắp xếp có cùng kiểu lúc đầu. Chẳng hạn có thể gọi quicksort bởi nhiều lời gọi khác nhau như sau :

```
quicksort [5, 2, 9, 1]
--> [1, 2, 5, 9]
quicksort ['d', 'a', 'c', 'b']
--> ['a', 'b', 'c', 'd']
```

3. Bài toán tám quân hậu

Nội dung bài toán tám quân hậu (eight queens problem) như sau : hãy tìm cách đặt tám quân hậu lên một bàn cờ vua (có 8×8 ô, lúc đầu không chứa quân cờ nào) sao cho không có quân hậu nào ăn được quân hậu nào ? Theo luật cờ vua, một quân hậu có thể ăn được bắt cứ quân cờ (của đối phương) nào nằm trên cùng cột, hay cùng hàng, hay cùng đường chéo thuận, hay cùng đường chéo nghịch với nó.



Hình I.6. Một lời giải của bài toán tám quân hậu.

Bài toán tám quân hậu được nhà Toán học người Đức Call Friedrich Gauss đưa ra vào năm 1850 nhưng không có lời giải hoàn toàn theo phương pháp giải tích. Sau đó bài toán này được nhiều người giải trọn vẹn trên máy tính điện tử, theo nhiều cách khác giải nhau.

Ý tưởng chung của mỗi lời giải là phải làm sao đặt một quân hậu trên mỗi cột , do vậy có thể biểu diễn bàn cờ bởi một danh sách các số nguyên cho biết vị trí hàng của mỗi quân hậu trong mỗi cột liên tiếp. Hàm `queens n` trong chương trình sau đây trả về mọi vị trí an toàn để đặt các quân hậu trên `n` cột đầu tiên.

```
queens 0 = [[]]
queens (n+1)= [q:b | b <- queens n; q <- [0..7]; safe q b]
safe q b = and [ not checks q b i | i <- [0..#b-1] ]
checks q b i = q=b!i or abs(q - b!i) = i+1
```

Lời gọi để nhận được tất cả các phương án đặt quân hậu là `queens 8`.

4. Bài toán Hamming

Bài toán Hamming (Hamming problem) sau đây minh họa ưu thế của phương pháp tính giá trị hàm theo kiểu khôn ngoan. Yêu cầu đặt ra là tạo các số Hamming (Hamming number) theo thứ tự tăng dần (increasing order), mỗi số có thể được viết dưới dạng :

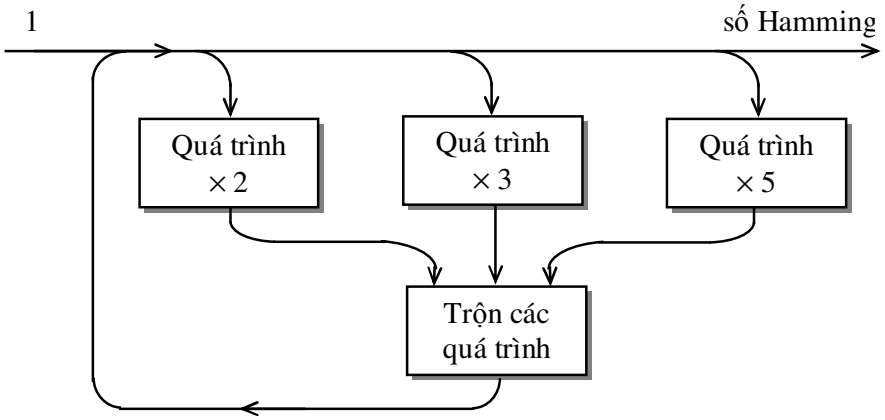
$2^i \times 3^j \times 5^k$

Như vậy, các số Hamming chỉ chứa các thừa số `2`, `3` và `5`. Do đó, kết quả chương trình sẽ bắt đầu với các số :

```
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, ...
```

và chương trình dừng khi người sử dụng không muốn tiếp tục nữa.

Thoạt nhìn, ta có thể nghĩ rằng bài toán được giải bằng cách sử dụng ba vòng lặp cho `i`, `j` và `k`. Tuy nhiên, xem xét kỹ thì bài toán trở nên hóc búa, vì rằng dãy kết quả phải được sắp xếp và không cho phép các số trùng nhau.



Hình I.7. Giải bài toán Hamming nhờ các quá trình giải song song.

Có thể giải thích bài toán Hamming một cách tinh tế hơn bằng cách sử dụng một mô hình xử lý song song như trong *Hình I.7*. Mô hình sử dụng ba quá trình nhân, mỗi quá trình nhận một dòng giá trị vào giống nhau và tạo ra một dòng ra phân biệt.

Quá trình thứ nhất nhân các số đưa vào với 2 để gửi ra dòng kết quả. Tương tự, quá trình thứ hai nhân các số với 3 và quá trình thứ ba nhân các số với 5 để gửi ra dòng kết quả. Mỗi

một dòng kết quả ra đều đã được sắp xếp theo thứ tự tăng dần. Một quá trình thứ tư làm nhiệm vụ trộn (merge) ba dòng kết quả sao cho các số đưa ra có thứ tự tăng dần và đã loại bỏ mọi số trùng nhau. Kết quả trộn được đem quay ngược lại thành dòng vào để tiếp tục quá trình nhân.

Lúc đầu, giá trị 1 được lấy làm dòng vào. Như vậy dòng vào sẽ chứa đúng các số Hamming theo dãy tăng dần và không chứa các số trùng nhau.

Giá trị khởi đầu 1 là một số Hamming, sau đó được nhận với 2, 3 hoặc 5 cũng là những số Hamming, còn các số không được tạo ra theo cách nhân này sẽ không phải là các số Hamming.

Mặc dù ngôn ngữ Miranda không có các quá trình song song, nhưng mô hình trên đây rất dễ dàng được triển khai trong ngôn ngữ này, bằng cách sử dụng kỹ thuật tính giá trị hàm theo kiểu khôn ngoan. Chương trình Miranda như sau :

```
mul :: num -> [num] -> [num]   || nhân các phần tử
mul a s = [a *x | x <- s]      || của danh sách với thừa số a đã cho
ham :: [num]                   || tạo các số Hamming
ham = 1 : merge3 (mul 2 ham) (mul 3 ham) (mul 5 ham)
```

Hàm `mul` thực hiện phép nhân bằng cách nhân tất cả các giá trị trong danh sách `s` với một thừa số đã cho `a`, là 2, 3 hoặc 5. Hàm `ham` xây dựng một danh sách chứa giá trị 1 ráp với các kết quả của phép trộn ba dòng tương ứng với ba quá trình nhân. Hàm `ham` không có tham đối, nhưng chương trình sẽ được bắt đầu bởi lời gọi `ham`. Phép trộn ba dòng (3-way merge) được xem như là phép trộn của hai dòng (2-way merge) :

```
merge3 x y z = merge2 x (merge2 y z)
merge2 (x:xs) (y:ys)         || trộn hai danh sách đã sắp xếp
= (x:merge2 xs (y:ys)), if x<y
= (x:merge2 xs ys), if x=y   || loại bỏ số trùng nhau
= (y:merge2 (x:xs) ys), if x>y
```

Hàm trộn hai dòng `merge3` loại bỏ các số trùng nhau, tương tự như hàm trộn `merge` trong thư viện của ngôn ngữ Miranda.

Chương trình trên đây chủ yếu dựa vào kỹ thuật tính giá trị hàm theo kiểu khôn ngoan và không thể chạy được nếu không dùng kỹ thuật này. Thật vậy, với *ngữ nghĩa ngặt* (strick semantics), phép nhân thứ nhất được bắt đầu với thừa số 2 là `(mul 2 ham)` sẽ gây ra một vòng lặp vô hạn. Tuy nhiên với kỹ thuật tính giá trị hàm theo kiểu khôn ngoan, hệ thống sẽ tìm cách rút gọn hàm `merge3` và chỉ tính giá trị các đối số của nó khi thực sự cần thiết.

I.3 Kết luận

Cho đến nay, các ngôn ngữ hàm đã được phát triển là Lisp, ISWIM, FP, Scheme, Common Lisp, Hope, Standard ML, Miranda, Haskell và Lucid. Phần tiếp theo của giáo trình này sẽ nghiên cứu Scheme, một ngôn ngữ hàm có nguồn gốc từ Lisp.

Ngôn ngữ Lisp (**Lis**t **p**rocessing languages) được đề xuất bởi Mc Carthy từ năm 1958. Rất nhiều khái niệm về lập trình hàm được phát triển từ ngôn ngữ Lisp. Lisp hiện đại lại không phải là ngôn ngữ hàm thuần túy, mà cho phép sử dụng các phép gán cho biến, và do vậy, có thể xảy ra hiệu ứng phụ.

Tuy nhiên, Lisp là ngôn ngữ đầu tiên mang phong cách lập trình hàm. Lisp cho phép viết các hàm đệ quy và các hàm bậc cao. Danh sách là cấu trúc dữ liệu chủ yếu của Lisp. Việc cấp phát và giải tỏa bộ nhớ được tiến hành tự động, nhờ kỹ thuật dọn rác (garbage collection).

Ngôn ngữ Lisp chủ yếu được phát triển cho *các ứng dụng ký hiệu* (symbolic applications), đặc biệt trong lĩnh vực trí tuệ nhân tạo (artificial intelligence). Đối với các phép tính số thì Lisp sử dụng không hiệu quả vì tốc độ tính toán chậm. Một điểm yếu khác của Lisp là các chương trình Lisp thường rất khó đọc, khó hiểu, do cú pháp sử dụng các dấu ngoặc là chủ yếu. Ví dụ, hàm `factorial` tính giai thừa, được viết trong chương trình Lisp như sau :

```
(defun factorial (n)
    (if (<= n 1)
        1
        (* n (factorial (- n 1)))))
```

Rõ ràng chương trình này khó đọc và khó hiểu hơn chương trình Miranda đã xét trước đây. Cú pháp của Lisp rất đơn điệu (uniform), tuy nhiên, điều này giúp cho việc lập trình và lưu giữ các chương trình Lisp trở nên dễ dàng như là dữ liệu chuẩn. Yếu tố này cũng giúp cho người lập trình dễ viết các trình sửa lỗi (debugger) và những công cụ tiện ích ngay trong Lisp.

Một vấn đề khác đối với ngôn ngữ Lisp là sự thiếu vắng hệ thống khai báo kiểu. Một danh sách có thể chứa nhiều kiểu phần tử khác nhau. Điều thoải mái này có thể có lợi trong một số trường hợp, nhưng lại gây khó khăn cho trình biên dịch khi cần phát hiện các lỗi sai về sử dụng kiểu dữ liệu. Lisp thường sử dụng kỹ thuật động để cấp phát bộ nhớ vật lý cho biến (dynamic binding) nên rất dễ gây ra sự lộn xộn, nhầm lẫn khi sử dụng biến.

Ngôn ngữ Lisp là tổ tiên trực tiếp của nhiều ngôn ngữ khác, có nhiều ảnh hưởng đến cách thiết kế các ngôn ngữ hàm. Các ngôn ngữ thừa kế Lisp sử dụng hệ thống kiểu, cho phép sử dụng các kỹ thuật tính giá trị hàm theo kiểu khôn ngoan và so khớp.

Tóm tắt chương 1

- Lập trình hàm tạo ra những hàm nhận các giá trị vào thuộc một miền xác định để cho ra kết quả là các giá trị thuộc một miền khác, giống như một hàm toán học. Các ngôn ngữ hàm thuần túy thiếu các yếu tố mệnh lệnh như phép gán và các lệnh lặp. Tuy nhiên, các hàm không gây ra hiệu ứng phụ như hầu hết các ngôn ngữ mệnh lệnh gặp phải.
- Các ngôn ngữ hàm thuần túy có tính nhất quán trong kết quả trả về của hàm, nghĩa kết quả của một phép áp dụng hàm không phụ thuộc vào thời điểm được gọi mà chỉ phụ thuộc vào các tham đối được cung cấp khi gọi hàm như thế nào.
- Tính nhất quán khi hàm trả về kết quả làm cho một chương trình hàm dễ đọc, dễ thay đổi, dễ mô phỏng song song và dễ chứng minh tính đúng đắn.
- Những khái niệm quan trọng nhất trong các ngôn ngữ hàm là các hàm đệ quy, danh sách, tính đa kiểu, các hàm bậc cao, tham đối hoá từng phần và so khớp các phương trình.
- Các ngôn ngữ hàm sử dụng phép đệ quy để thay cho phép lặp.
- Danh sách là cấu trúc dữ liệu chính của các ngôn ngữ hàm. Nhiều phép toán trên danh sách là tiền định (predefined).
- Việc quản lý bộ nhớ cho các cấu trúc dữ liệu được tiến hành tự động. Người lập trình không cần quan tâm đến sự có mặt của các địa chỉ máy (machine address) và không xử lý dữ liệu kiểu con trỏ. Việc giải tỏa bộ nhớ không còn cần dùng đến nữa cũng được tiến hành tự động theo kỹ thuật dọn rác.
- Các ngôn ngữ hàm hiện đại cho phép xử lý các hàm đa kiểu, mỗi hàm có thể nhận nhiều tham đối có kiểu khác nhau. Một số ngôn ngữ hàm sử dụng phép định kiểu ngặt, nhưng các kiểu luôn được suy đoán (inferred) bởi bộ kiểm soát kiểu và người lập trình không cần phải mô tả trước (như trong các ngôn ngữ mệnh lệnh).
- Một hàm bậc cao nhận một hàm làm tham đối. Các hàm bậc cao mang lại tính mềm dẻo đáng kể cho các ngôn ngữ hàm.
- Phép rút gọn theo thứ tự áp dụng tính giá trị các tham đối của hàm trước tiên, sau đó mới tính đến bản thân hàm. Phép rút gọn theo thứ tự chuẩn lại áp dụng hàm cho các tham đối không cần tính giá trị, và chỉ tính giá trị các tham đối khi thực sự cần thiết.
- Phép rút gọn theo thứ tự chuẩn căn cứ trên kỹ thuật tính giá trị hàm theo kiểu khôn ngoan, cho phép xử lý các cấu trúc dữ liệu vô hạn nếu có thể. Kỹ thuật này tỏ ra thân thiện hơn so với kỹ thuật tính giá trị có lợi, nhưng khó vận hành có hiệu quả.
- Các ngôn ngữ hàm hiện đại mô tả các hàm như một tập hợp các phương trình (set of equations) và sử dụng kỹ thuật so khớp để lựa chọn các phương trình thỏa mãn một mẫu đã cho.
- Các ngôn ngữ hàm không thuần túy (impure functional languages) như Lisp mang một số tính chất của ngôn ngữ hàm, nhưng không có tính nhất quán khi hàm trả về kết quả.

Bài tập chương 1

1. Viết các hàm Miranda tính gần đúng giá trị các hàm sau với độ chính xác e = 10^{-5}

   $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...$ cho đến khi $\left|\frac{1}{2n-1}\right| < \varepsilon$

   $1 + \frac{x^2}{2} + \frac{2}{3}\times\frac{x^4}{4} + \frac{2}{3}\times\frac{4}{5}\times\frac{x^6}{6} + ...$ cho đến khi phần tử thứ $n <$ e

   $S = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^n}{n!} + \ldots$ cho đến khi $\left|\frac{x^n}{n!}\right| < \varepsilon$

   $S = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + ... + \frac{x^{2n}}{(2n)!} + ...$ cho đến khi $\left|\frac{x^{2n}}{(2n)!}\right| < 10^{-5}$

   $y = \sqrt{x + \sqrt{x + \ldots + \sqrt{x}}}$ có $n > 1$ dấu căn

2. Viết một chương trình Miranda để tìm ước số chung lớn 4 của * số nguyên bất kỳ p, q.
3. Cho danh sách các số nguyên *L* và một số nguyên *K*, hãy viết chương trình bằng Miranda thực hiện các việc sau đây :

   a) Đếm các số chia hết cho *K* trong *L* ?

   b) Kiểm tra số *K* có nằm trong danh sách *L* hay không ?

   c) Cho biết vị trí phần tử đầu tiên trong danh sách *L* bằng *K* ?

   d) Tìm tất cả các vị trí của các phần tử bằng *K* trong danh sách *L* ?

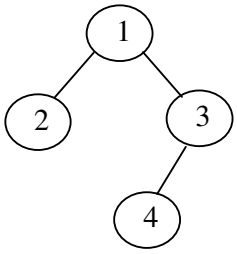
   e) Thay phần tử bằng *K* trong danh sách L bởi phần tử *K*' đã cho ?
4. Viết chương trình Miranda để xóa ba phần tử đầu tiên và ba phần tử cuối cùng của một danh sách.
5. Viết chương trình Miranda để xóa N phần tử đầu tiên của một danh sách. Thất bại nếu danh sách không có đủ N phần tử.
6. Viết chương trình Miranda để xóa N phần tử cuối cùng của một danh sách. Thất bại nếu danh sách không có đủ N phần tử.
7. Định nghĩa bằng Miranda hai hàm `even_length` và `odd_length` để kiểm tra số các phân tử của một danh sách đã cho là chẵn hay lẻ tương ứng.

   Ví dụ danh sách `[a, b, c, d ]` có độ dài chẵn,
   danh sách `[ a, b, c ]` có độ dài lẻ.

   Viết chương trình Miranda kiểm tra một danh sách có phải là một tập hợp con của một danh sách khác không ?
8. Viết chương trình Miranda để lấy ra phần tử thứ N trong một danh sách. Thất bại nếu danh sách không có đủ N phần tử.

   Viết chương trình Prolog tìm phần tử lớn nhất và phần tử nhỏ nhất trong một danh sách các số.
9. Viết chương trình Miranda để kiểm tra hai danh sách có rời nhau (disjoint) không ?

10. Viết một chương trình Miranda để giải bài toán tháp Hà Nội (Tower of Hanoi) : chuyển N đĩa có kích thước khác nhau từ một cọc qua cọc thứ hai lấy cọc thứ ba làm cọc trung gian, sao cho luôn luôn thỏa mãn mỗi lần chỉ chuyển một đĩa từ một cọc này sang một cọc khác, trên một cọc thì đĩa sau nhỏ hơn chồng lên trên đĩa trước lớn hơn và đĩa lớn nhất ở dưới cùng.
11. Viết một chương trình Miranda để tạo ra các số nguyên tố sử dụng sàng Eratosthènes. Chương trình có thể không kết thúc. Thử sử dụng kỹ thuật tính giá trị hàm theo kiểu khôn ngoan để có lời giải đơn giản và hiệu quả.
12. Cây nhị phân (binary tree) được biểu diễn như là một một danh sách gồm ba phần tử dữ liệu : nút gốc (root node), cây con bên trái (left subtree) và cây con bên phải (right subtree) của nút gốc. Mỗi cây con lại được xem là những cây nhị phân. Cây, hoặc cây con rỗng (empty tree) được biểu diễn bởi một danh sách rỗng. Ví dụ cho cây nhị phân có 4 nút `[1, [2, [], []], [3, [4, [], []], []]]` như sau :



Hình I.8. Cây nhị phân có 4 nút.

Viết chương trình duyệt cây lần lượt theo thứ tự giữa (trái-gốc-phải), trước (gốc-trái-phải) và sau (trái- phải-gốc) ?

CHƯƠNG II. NGÔN NGỮ SCHEME

A line may take us hours, yet if it does not seem a moment's thought
All our stitching and unstitching has been as nought.
Yeats - Adam's Curse

II.1 Giới thiệu Scheme

Scheme là một *ngôn ngữ thao tác ký hiệu* (symbolic manipulation) do Guy Lewis Steele Jr. và Gerald Jay Sussman đề xuất năm 1975 tại MIT (Massachusetts Institute of Technology, Hoa Kỳ), sau đó được phát triển nhanh chóng và ứng dụng rất phổ biến. Scheme là ngôn ngữ thuộc họ Lisp và mang tính sư phạm cao. Scheme giải quyết thích hợp các bài toán toán học và xử lý ký hiệu. Theo W. Clinger và J. Rees[3] :

... «*Scheme demonstrate that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a pratical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today...* ».

Tương tự các ngôn ngữ hàm khác, Scheme có cú pháp rất đơn giản nên rất dễ lập trình. Các cấu trúc dữ liệu cơ sở của Scheme là danh sách và cây, dựa trên khái niệm về *kiểu dữ liệu trừu tượng* (data abstraction type). Một chương trình Scheme là một dãy các định nghĩa *hàm* (hay *thủ tục*) góp lại để định nghĩa một hoặc nhiều hàm phức tạp hơn. Hoạt động cơ bản trong lập trình Scheme là tính giá trị các biểu thức. Scheme làm việc theo *chế độ tương tác* (interaction) với người sử dụng.

Mỗi vòng tương tác xảy ra như sau :

- Người sử dụng gõ vào một biểu thức, sau mỗi dòng nhấn enter (↵).
- Hệ thống in ra kết quả (hoặc báo lỗi) và qua dòng mới.
- Hệ thống đưa ra một dấu nhắc (prompt character) và chờ người sử dụng đưa vào một biểu thức tiếp theo...

Việc lựa chọn dấu nhắc tùy theo quy ước của hệ thống, thông thường là dấu lớn hơn (>) hoặc dấu hỏi (?)[4].

Một dãy các phép tính giá trị biểu thức trong một vòng tương tác được gọi là một *chầu làm việc* (session). Sau mỗi chầu, Scheme đưa ra thời gian và số lượng bộ nhớ (bytes) đã sử dụng để tính toán.

Để tiện theo dõi, cuốn sách sử dụng các quy ước như sau :

[3] Xem tài liệu định nghĩa ngôn ngữ Scheme tại địa chỉ http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html
[4] Trong cuốn sách này, tác giả không ghi ra dấu nhắc hệ thống (dấu >) cho dễ đọc.

- Kết quả tính toán được ghi theo sau dấu mũi tên (`-->`).
- Các thông báo về lỗi sai được đặt trước bởi ba dấu sao (`***`).
- Cú pháp của một biểu thức được viết theo quy ước EBNF kiểu chữ nghiêng đậm. Ví dụ : **<e>**

Để tiện trình bày tiếng Việt, một số phần chú thích và kết quả tính toán không in theo kiểu chữ Courier.

Chú thích trong Scheme

Chú thích (comment) dùng để diễn giải phần chương trình liên quan giúp người đọc dễ hiểu, dễ theo dõi nhưng không có hiệu lực đối với Scheme (Scheme bỏ qua phần chú thích khi thực hiện). Chú thích được bắt đầu bởi một dấu chấm phẩy (`;`), được viết trên một dòng bất kỳ, hoặc từ đầu dòng, hoặc ở cuối dòng. Ví dụ :

```
; this is a comment line
(define x 2) ; định nghĩa biến x có giá trị 2
```

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
   (lambda (n)
      (if (= n 0)
          1 ; Base case: return 1
          (* n (fact (- n 1))))))
```

II.2 Các kiểu dữ liệu của Scheme

Kiểu dữ liệu (data type) là một tập hợp các giá trị có quan hệ cùng loại với nhau (related values). Các kiểu dữ liệu được xử lý tuỳ theo bản chất của chúng và thường có tính phân cấp. Trong Scheme có hai loại kiểu dữ liệu là *kiểu đơn giản* (simple data type) và *kiểu phức hợp* (compound data type). Trong chương này, ta sẽ xét các kiểu dữ liệu đơn giản trước.

II.2.1. Các kiểu dữ liệu đơn giản

Các kiểu dữ liệu đơn giản của Scheme bao gồm kiểu *số* (number), kiểu *lôgích* (boolean), kiểu *ký tự* (character) và kiểu *ký hiệu* (symbol).

II.2.1.1. Kiểu số

Kiểu số của Scheme có thể là số nguyên (integer), số thực (real), số hữu tỷ (rational) và số phức (complex) như sau :

Kiểu số	Ví dụ
số nguyên	`52`
số thực	`3.0, -2.5`
số hữu tỷ	`6/10, 23/5`
số phức	`3+4i, 3`

Scheme không phân biệt số nguyên hay số thực. Các số không hạn chế về độ lớn, miễn là bộ nhớ hiện tại cho phép.

Với các số, Scheme cũng sử dụng các phép toán số học thông dụng +, –, *, /, max, min, phép lấy căn bậc hai $\sqrt{\ }$ và so sánh số học với số lượng đối số tương ứng :

```
(+ x1 ... xn)        -->  x1 + ... + xn
(- x1 x2)            -->  x1 - x2
(* x1 ... xn)        -->  x1 * ... * xn
(/ x1 x2)            -->  x1 / x2
(quotient x1 x2)     -->  phần nguyên của (x1 / x2)
(remainder x1 x2)    -->  phần dư của phép chia nguyên (x1 / x2), lấy dấu x1
(modulo x1 x2)       -->  phần dư của phép chia nguyên (x1 / x2) , lấy dấu x2
(max x1 ... xn)      -->  max (x1, ..., xn)
(min x1 ... xn)      -->  min (x1, ..., xn)

(sqrt x)             -->  √x
```

Các phép so sánh sau đây trả về `#t` nếu kết quả so sánh lần lượt các giá trị $x_1, ..., x_n$ được thoả mãn, ngược lại trả về `#f` :

```
(= x1 ... xn)
(< x1 ... xn)
(<= x1 ... xn)
(> x1 ... xn)
(>= x1 ... xn)
```

Ví dụ :

```
(* 1 2 3 4 5 6 7 8 9)
--> 362880

(= 1 2 3 4 5)
--> #f

(= 1 1 1 1 1)
--> #t

(< 1 2 3 4 5)
--> #t

(> 4 3 2 1)
--> #t

(<= 1 2 3 4 5)
--> #t

(>= 6 5 4 3 2 1)
--> #t
```

Thể tích hình cầu bán kính *R* :

```
(* 3 pi R R R)
```

Nhiệt độ Farenheit được biểu diễn qua nhiệt độ Celsius `C` :

```
(+ 32 (* 9/5 C))
```

Một biểu thức Scheme có thể trộn lẫn lộn các số nguyên và các số thực :

```
(+ 2.3 5)
--> 7.3
```

Phép tính trên các số hữu tỷ :

```
(* 2/3 5/2)
--> 10/6
```

Cùng lúc có thể gõ vào nhiều biểu thức (trước khi Enter) để nhận được nhiều kết quả :

```
(* 2 3)(+ 1 4)(- 7 9)
--> 6
    5
    -2
```

Ký tự

Một ký tự (character) của Scheme có dạng `#\<char>`. Ví dụ :

```
#\a
--> #\a ;chữ a thường
#\A
--> #\A ;chữ A hoa
#\(
--> #\( dấu ngoặc trái
```

Chuỗi

Chuỗi (string) là kiểu dữ liệu phức hợp của Scheme, gồm dãy các ký tự tuỳ ý đặt giữa hai dấu nháy kép, nhưng các dấu nháy kép này vẫn được giữ nguyên trong giá trị của chuỗi :

```
"Chào các bạn !"
--> " Chào các bạn !"
```

Tên

Mọi ngôn ngữ lập trình đều sử dụng tên để chỉ định các đối tượng cần xử lý. Trong Scheme, tên được tạo thành từ các chữ cái, chữ số và các dấu đặc biệt, trừ `# () []` và dấu cách (space) dùng để phân cách các đối tượng.

Tên của Scheme được bắt đầu bởi một chữ cái và không phân biệt chữ hoa chữ thường. Viết `pi` hay `PI` đều cùng chỉ một tên. Nên chọn đặt tên «biết nói» (mnemonic) và sử dụng các dấu nối (–). Chẳng hạn các tên sau đây đều hợp lệ :

```
pi          *           pi-chia-2
x           +           a34kTMNs
soup        <=?         is-this-a-very-long-name?
lambda      V19a        list->vector
```

II.2.1.2. Kiểu lôgích và vị từ

Mọi ngôn ngữ lập trình đều sử dụng các cấu trúc điều khiển sử dụng đến các giá trị lôgích và do đó, cần biểu diễn các giá trị lôgích. Trong Scheme, các hằng có sẵn kiểu lôgích là `#t` (true) và `#f` (false).

Vị từ (predicate) là một hàm luôn trả về giá trị lôgích. Theo quy ước, tên các vị từ được kết thúc bởi một dấu chấm hỏi (`?`).

Thư viện Scheme có sẵn nhiều vị từ. Sau đây là một số vị từ dùng để kiểm tra kiểu của giá trị của một biểu thức :

```
(number? s)     --> #t nếu s là một số thực, #f nếu không
(integer? s)    --> #t nếu s là một nguyên, #f nếu không
(string? s)     --> #t nếu s là một chuỗi, #f nếu không
(boolean? s)    --> #t nếu s là một lôgích, #f nếu không
(procedure? s) --> #t nếu s là một hàm, #f nếu không
```

Ví dụ :

```
(string? 10)
--> #f          ; phải viết "10"
(procedure? +)
--> #t          ; dấu + là một tên hàm
(complex? 3+4i)
--> #t
(real? -2.5+0.0i)
--> #t
(real? #e1e10)
--> #t
(rational? 6/10)
--> #t
(integer? 3)
--> #t
```

Scheme có vị từ `equal?` sử dụng hai tham đối để so sánh các giá trị :

```
(equal? s1 s2 )
--> #t nếu s1 = s2
```

Đối với các số thực, vị từ bằng nhau là dấu = và các vị từ so sánh là các dấu phép toán quan hệ <, <=, >, >=, `zero?` :

```
(< nb1 nb2)
--> #t nếu nb1 < nb2, #t nếu không.
```

Phép toán phủ định là `not` :

```
(not s)
--> #t nếu s có giá trị #f, #f nếu không.
```

Ví dụ :

```
(zero? 100)
--> #f
(not #f)
--> #t
(not #t)
--> #f
(not "Hello, World!")
--> #f
```

II.2.1.3. Ký hiệu

Ngôn ngữ Scheme không chỉ xử lý các giá trị kiểu số, kiểu lôgích và kiểu chuỗi như đã trình bày, mà còn có thể *xử lý ký hiệu* nhờ phép trích dẫn. Giá trị ký hiệu của Scheme là một tên (giống tên biến) mà không gắn với một giá trị nào khác. Chú ý Scheme luôn luôn in ra các ký hiệu trích dẫn dạng chữ thường.

Kiểu trích dẫn vẫn hay gặp trong ngôn ngữ nói và viết hàng ngày. Khi nói với ai đó rằng «*hãy viết ra tên anh*», thì người đó có thể hành động theo hai cách :

- hoặc viết *Trương Chi*, nếu người đó tên là Trương Chi và hiểu câu nói là «*viết ra tên của mình*».
- hoặc viết *tên anh*, nếu người đó hiểu câu nói là phải viết ra cụm từ «*tên anh*».

Trong ngôn ngữ viết, người ta dùng các dấu nháy (đơn hoặc kép) để chỉ rõ cho cách trả lời thứ hai là : «*hãy viết ra "tên anh"*». Khi cần gắn một giá trị ký hiệu cho một tên, người ta hay gặp sai sót. Chẳng hạn, việc gắn giá trị ký hiệu là `pierre` cho một biến có tên là `first-name` :

```
(define first-name pierre)
*** ERROR -- unbound variable: pierre.
```

Ở đây xuất hiện sai sót vì Scheme tính giá trị của tên `pierre`, nhưng tên này lại không có giá trị. Để chỉ cho Scheme giá trị chính là ký hiệu `pierre`, người ta đặt trước giá trị `pierre` một *phép trích dẫn* (quote operator) :

```
'pierre
--> pierre
```

Ký tự `'` là cách viết tắt của hàm `quote` trong Scheme :

`'<exp>` tương đương với `(quote <exp>)`

Hàm `quote` là một dạng đặc biệt tiền định cho phép trả về tham đối của nó dù tham đối là thế nào mà không tính giá trị :

```
(quote pierre)
--> pierre
```

Khái niệm trích dẫn có tác dụng quan trọng : khái niệm bằng nhau trong ngôn ngữ tự nhiên và trong Scheme là khác nhau về mặt Toán học :

1 + 2 = 3 nhưng "1 + 2" ≠ "3"

```
(= (+ 1 2) 3)
--> #t
```

nhưng :

```
(= '(+ 1 2) '3)
--> error !!!!! không cùng giá trị
```

Ta có thể định nghĩa kết quả trích dẫn cho biến :

```
(define first-name 'pierre)
```

Bây giờ, nếu cần in ra giá trị của `first-name`, ta có :

```
first-name
--> pierre
(define x 3)
```

```
x
--> 3
'x
--> x
```

Họ các ngôn ngữ Lisp rất thuận tiện cho việc xử lý ký hiệu. Một trong những áp dụng quan trọng của Lisp là *tính toán hình thức* (formal computation). Người ta có thể tính đạo hàm của một hàm, tính tích phân, tìm nghiệm các phương trình vi phân. Những chương trình này có thể giải các bài toán tốt hơn con người, nhanh hơn và ít xảy ra sai sót. Trong chương sau, ta sẽ thấy được làm cách nào để Scheme tính đạo hàm hình thức của x^3 là $3x^2$.

Một xử lý ký hiệu quan trọng nữa là xử lý chương trình : một trình biên dịch có các dữ liệu là các chương trình được viết trên một hệ thống ký hiệu là ngôn ngữ lập trình. Bản thân một chương trình Scheme cũng được biểu diễn như một danh sách (sẽ xét sau). Chẳng hạn :

```
(define (add x y)
   (+ x y))
```

là một danh sách gồm ba phần tử `define`, `(add x y)` và `(+ x y)`.

Chú ý :

Trong thực tế, người ta chỉ sử dụng `quote` trong lời gọi chính của một hàm. Định nghĩa của một hàm nói chung không chứa `quote` (đó là trường hợp của tất cả các hàm đã viết cho đến lúc này), trừ khi người ta cần xử lý ký hiệu.

Để kiểm tra giá trị một biểu thức có phải là một ký hiệu không, người ta dùng vị từ `symbol?` như sau :

```
(symbol? 'pierre)
--> #t
(symbol? #t)
--> #f
(symbol? "pierre")
--> #f
```

Ví dụ cuối (kết quả là `#f`) chỉ ra rằng không nên nhầm lẫn ký hiệu với chuỗi, dẫu rằng trong phần lớn các ngôn ngữ, các chuỗi là phương tiện duy nhất để mô hình hóa các tên gọi.

II.2.2. Khái niệm về các biểu thức tiền tố

Có nhiều cách để biểu diễn các biểu thức số học. Ngôn ngữ Scheme sử dụng một cách hệ thống khái niệm dạng ngoặc tiền tố. Nguyên tắc là viết các phép toán rồi mới đến các toán hạng và đặt tất cả trong cặp dấu ngoặc.

Ví dụ biểu thức số học `4+76` được viết thành `(+ 4 76)`. Một cách tổng quát, nếu `op` chỉ định một phép toán hai ngôi, một biểu thức số học có dạng :

```
exp1 op exp2
```

sẽ được viết dưới dạng ngoặc tiền tố là :

```
(op ~exp1 ~exp2)
```

trong đó, `~expj` là dạng tiền tố của biểu thức con `expj`, `j` = `1`, `2`.

Đối với các phép toán có số lượng toán hạng tuỳ ý, chỉ cần viết dấu phép toán ở đầu các toán hạng. Ví dụ biểu thức số học :

```
4 + 76 + 19
```

sẽ được viết thành :

```
(+ 4 76 19)
```

Chú ý đặt dấu cách hay khoảng trống (space) giữa dấu phép toán và giữa mỗi toán hạng. Người ta có thể trộn lẫn các phép toán :

`34*21 - 5*18*7` được viết thành `(- (* 34 21) (* 5 18 7))`,

`-(2*3*4)` được viết thành `(- (* 2 3 4))`.

Một trong những lợi ích của việc sử dụng các cặp dấu ngoặc trong biểu thức là thứ tự ưu tiên thực hiện của các phép toán được bảo đảm, không mập mờ và không sợ bị nhầm lẫn.

Dạng tiền tố được sử dụng cho *tất cả* các biểu thức. Ví dụ, áp dụng hàm `f` cho các đối số `2`, `0` và `18` viết theo dạng Toán học là `f(2, 0, 18)` và được biểu diễn trong Scheme là `(f 2 0 18)`. Ở đây, các dấu phẩy được thay thế bởi các dấu cách đủ để phân cách các thành phần.

Cách viết các biểu thức dạng ngoặc tiền tố làm tăng nhanh số lượng các dấu ngoặc thoạt tiên làm hoang mang người đọc. Tuy nhiên, người sử dụng sẽ nhanh chóng làm quen và rất nhiều phiên bản của Scheme hiện nay (trong môi trường cửa sổ và đồ hoạ) có khả năng kiểm tra tính tương thích giữa các cặp dấu ngoặc sử dụng trong biểu thức.

Các biểu thức có thể lồng nhau nhiều mức. Quy tắc tính giá trị theo *trình tự áp dụng* (xem mục II.7, chương 1) như sau : các biểu thức trong cùng nhất được tính giá trị trước, sau đó thực hiện phép toán là các biểu thức bên ngoài tiếp theo. Lặp lại quá trình này nhiều lần cho đến khi các biểu thức đã được tính hết.

```
(+ (* 2 3) 4)     ; = (+ 6 4)
--> 10
```

Rõ ràng các biểu thức lồng nhau làm người đọc khó theo dõi, chẳng hạn biểu thức sau đây khó theo dõi :

```
(+ 1 (* 2 3 (- 5 1)) 3) ; = (+ 1 (* 2 3 4) 3) = (+ 1 24 3)
```

Nên viết biểu thức trên nhiều dòng khác nhau theo quy ước viết *thụt dòng* (indentation) và cân thẳng đứng tương tự các dòng lệnh trong các chương trình có cấu trúc (Pascal, C, Ada...). Biểu thức trên được viết lại như sau :

```
(+ 1
   (* 2 3 (- 5 1))
   3)
--> 28
```

Người ta thường viết thẳng đứng các toán hạng của một hàm, thụt vào so với tên hàm. Chẳng hạn biểu thức `(f a b c)` được viết :

```
(f a
   b
   c)
```

Tuy nhiên, cách viết này còn tuỳ thuộc vào thói quen (hay sở thích) của người lập trình. Khi các toán hạng là những biểu thức phức tạp, người ta có thể cân thẳng đứng như vừa nói, nhưng khi các biểu thức là đơn giản, người ta có thể viết cân theo hàng ngang cho tiện.

Do giá trị của một biểu thức không phụ thuộc vào cách viết như thế nào, các dấu cách và các dấu qua dòng ↵ đều có cùng một nghĩa trong Scheme, nên người lập trình có thể vận dụng quy ước viết thụt dòng sao cho phù hợp với thói quen của họ.

Chẳng hạn biểu thức Toán học :

$$\frac{\sin(a)+\sin(b)}{\sqrt{1+a^2+b^2}}$$

có thể viết trong Scheme :

```
(/ (+ (sin a) (sin b))
   (sqrt (+ 1 (* a a) (* b b))))
```

II.2.3. S-biểu thức

Người ta gọi *s-biểu thức* (s-expression, s có nghĩa là symbolic) là tất cả các kiểu dữ liệu có thể gộp nhóm lại với nhau (lumped together) đúng đắn về mặt cú pháp trong Scheme. Ví dụ sau đây đều là các s-biểu thức của Scheme :

```
42
#\A
(1 . 2)
'(a b c)
#(a b c)
"Hello"
(quote pierre)
(string->number "16")
(begin (display "Hello,World!") (newline))
```

Mọi s-biểu thức không phải luôn luôn đúng đắn về mặt cú pháp hoặc về mặt ngữ nghĩa, nghĩa là s-biểu thức không phải luôn luôn có một giá trị. Scheme tính giá trị của một biểu thức ngay khi biểu thức đó đã đúng đắn về mặt cú pháp, hoặc thông báo lỗi sai. Ví dụ biểu thức sau đây vào đúng :

```
(+ 7 ↵                ; ↵ là dấu enter
(* 3 4)) ↵
--> 19
```

Sau khi gõ `(+ 7` rồi ↵, Scheme lùi về đầu dòng tiếp theo và tiếp tục chờ (con trỏ nhấp nháy) vì biểu thức chưa đúng đắn về mặt cú pháp. Chỉ khi gõ tiếp `(* 3 4))`↵ mới làm xuất hiện kết quả trên đầu dòng tiếp theo do biểu thức đã vào đúng. Biểu thức sau đây vào sai và gây ra lỗi :

```
(+ 3 (*6 7))
--> ERROR: unbound variable: *6
    ; in expression: (... *6 7)
    ; in top level environment.
    ; Evaluation took 0 mSec (0 in gc) 11 cells work,38 bytes other
```

II.3 Các định nghĩa trong Scheme

II.3.1. Định nghĩa biến

Biến (variable) là một tên gọi được gán một giá trị có kiểu nào đó. Một biến chỉ định một vị trí nhớ lưu giữ giá trị này. Các tên đặc biệt như `define` gọi là từ khóa của ngôn ngữ do nó chỉ định một phép toán tiền định. Chẳng hạn, khi muốn chỉ định cho tên biến `pi` một giá trị *3.14*, ta viết :

```
(define pi 3.14159)
```

khi đó, ta có

```
pi
--> 3.14159
```

Dạng tổng quát của định nghĩa biến như sau :

(define var expr)

Sau khi định nghĩa biến **var** sẽ nhận giá trị của biểu thức `expr`.

Ví dụ : Định nghĩa biến `root2` :

```
(define root2 (sqrt 2))
root2
--> 1.4142135623730951
```

khi đó `root2` có thể được sử dụng như sau :

```
(* root2 root2)
--> 2.000000000000000 ; 15 con số 0
```

Tuy nhiên, nếu sử dụng một tên chưa được định nghĩa trước đó, Scheme sẽ thông báo lỗi :

```
toto
--> *** ERROR-- unbound variable : toto ; chưa được gán giá trị
```

Sheme có sẵn một số tên đã được định nghĩa, đó là các tên hàm cơ sở :

```
sqrt              ; giá trị của tên sqrt là gì ?
--> #[procedure]  ; là hàm tính căn bậc hai
```

Sự tương ứng giữa một biến và giá trị của biến được gọi là một *liên kết* (link). Tập hợp các liên kết của một chương trình được gọi là một *môi trường* (environment). Trong ví dụ trên, ta nhận được một môi trường gồm các liên kết giữa `pi` và giá trị của `pi`, giữa `root2` và giá trị của `root2`. Người ta cũng nói môi trường gồm các liên kết tiền định như liên kết của `sqrt`. Giá trị của một biến có thể bị thay đổi do sự định nghĩa lại. Thứ tự các định nghĩa là quan trọng khi chúng tạo nên các biểu thức chứa các biến, vì mỗi biến này phải có một giá trị trước khi tính giá tự biểu thức.

```
(define one 1)
(define two 2)
(define three (+ one two))
```

Ta có thể hoán đổi thứ tự hai định nghĩa biến `one` và `two`, nhưng định nghĩa biến `three` phải đặt cuối cùng. Nếu ta quyết định tiếp theo thay đổi giá trị của `one` bởi `(define one 0)`, thì giá trị gán cho biến `three` vẫn không đổi cho đến khi tính lại nó.

II.3.2. Định nghĩa hàm

II.3.2.1. Khái niệm hàm trong Scheme

Khi cần đặt tên cho biểu thức chứa các tham biến, ta đi đến khái niệm hàm. Ví dụ ta cần định nghĩa một hàm tính thể tích của hình cầu bán kính *R*. Ta sử dụng dạng `define` có cú pháp như sau :

```
(define (sphereVolume R)
   (* 4/3 pi R R R))
```

Ở đây *R* là một tham biến, còn tên gọi `pi` không phải là một tham biến vì `pi` đã được định nghĩa trên đây.

Một cách tổng quát, một định nghĩa hàm có dạng :

```
(define (func-name x1 … xk)
   body)
```

Các x_i, `i=1..k`, được gọi là các *tham biến hình thức* (formal parameters), hay gọi tắt là tham biến. Trong Scheme, các tham biến và giá trị trả về của hàm không cần phải khai báo kiểu. Tên gọi của tham biến không quan trọng. Chẳng hạn ta có thể định nghĩa lại hàm `spherevolume` bởi tên tham biến khác như sau :

```
(define (spherevolume x)
   (* 4/3 pi x x x))
```

Để tiện theo dõi đọc chương trình, người ta thường sử dụng các quy ước viết các tham biến để gợi ra một cách *không tường minh* (implicite) các kiểu của chúng.

- Các tham biến nhận các số nguyên được viết *n, i, j,* n_1, n_2, ...
- Các tham biến nhận các số thực là *r,* r_1, *nb,* nb_1, ...
- Một biểu thức bất kỳ *s,* s_1, *e, exp*, ...

II.3.2.2. Gọi hàm sau khi định nghĩa

Lời gọi một hàm sau khi người lập trình tự định nghĩa tương tự lời gọi một hàm có sẵn trong thư viện của Scheme. Lời gọi có dạng :

```
(func-name arg1 arg2 … argk)
```

trong đó, `func-name` là tên hàm đã được định nghĩa, các arg_i, `i=1..k`, được gọi là các *tham đối thực sự* (effective arguments).

Sau khi gọi, các tham đối arg_i được tính giá trị để gán cho mỗi tham biến hình thức x_i tương ứng. Tiếp theo, thân hàm (`body`) được tính giá trị trong một *môi trường cục bộ* (local environment), trong đó, các tham đối hình thức biểu diễn giá trị của các tham đối thực sự. Điều này không làm ảnh hưởng đến môi trường trước đó. Một tham đối có thể là một biểu thức Scheme nào đó, miễn là giá trị của nó phù hợp với kiểu dự kiến trong thân hàm.

Ví dụ, lời gọi :

```
(spherevolume (+ 8 4))
```

gây ra việc tính giá trị tính của thân hàm :

```
(* 4/3 pi r r r)
```

với `r` được tính từ đối `(+ 8 4)` là `12` để có kết quả trả về là :

```
--> 7234.56
```

Nếu ta tính lại `pi` chính xác hơn :

```
(define pi 3.14159)
```

Thì việc gọi lại hàm `SphereVolum` sẽ cho kết quả khác :

```
(SphereVolime 12)
--> 7238.22
```

II.3.2.3. Sử dụng các hàm bổ trợ

Do tính thể tính hình cầu cần tính lập phương của một số, do đó ta có thể định nghĩa hàm tính lập phương :

```
(define (cube nb) (* nb nb nb))
```

và ta định nghĩa lại hàm tính thể tích :

```
(define (spherevolume x) (* 4/3 pi (cube x)))
```

Ở đây, ta đã sử dụng hàm *bổ trợ* (auxileary functions) `cube` để tham gia định nghĩa hàm chính. Nếu như một hàm chỉ dùng để tính toán trung gian cho một hàm khác, mà không dùng chung, người ta có thể đặt nó trong hàm đó và không muốn nhìn thấy bởi các hàm khác. Đó là khái niệm hàm cục bộ đối với một hàm khác. Như vậy, hàm cục bộ cũng là hàm bổ trợ.

Không có quy tắc nhất định để xác định thứ tự định nghĩa các hàm. Người ta có thể định nghĩa một (hoặc nhiều) hàm chính trước, rồi định nghĩa các hàm bổ trợ để làm rõ các hàm chính, rồi tiếp tục định nghĩa các hàm bổ trợ của các hàm bổ trợ, v.v... Hoặc ngược lại, người ta có thể bắt đầu bởi các hàm bổ trợ trong cùng nhất trước rồi định nghĩa đến các hàm chính sau cùng. Đây chỉ là vấn đề phong cách lập trình, cơ bản là lúc nào thì cần tính giá trị của biểu thức, khi đó các hàm liên quan đã được định nghĩa rồi.

Ví dụ : Hàm sau đây tính cạnh huyền của một tam giác vuông cạnh `a`, `b` :

```
(define (hypotenuse a b)
   (sqrt (+ (square a) (square b))))
```

Ở đây sử dụng hàm bổ trợ `square` được định nghĩa tiếp theo :

```
(define (square x) (* x x))
```

Thông thường khi lập trình, người ta thường có thói quen định nghĩa lại các hàm do có sai sót về thiết kế, thái độ của bộ diễn dịch Scheme là tránh định nghĩa lại tất cả các hàm.

Các định nghĩa hàm có thể lồng nhau. Các định nghĩa bên trong của một hàm không thể tiếp cận được từ bên ngoài hàm : đó là những định nghĩa có tính cục bộ (local definitions). Điều này hoàn toàn có lợi khi cần định nghĩa các đối tượng hay các hàm phụ thuộc nhau.

Ví dụ ta cần tính $x^4 - x^2$ với một hàm được định nghĩa như sau :

```
(define (x4-minus-x2 x)
   (define x2 (square x))
      (* x2 (- x2 1)))
(x4-minus-x2 2)
--> 12

x2
--> ***ERROR --- unbound variable x2
```

Định nghĩa bên trong của một hàm khác với lệnh gán trong các ngôn ngữ mệnh lệnh (chẳng hạn Pascal) :

```
(define x 1)
(define (f)
   (define x 2) ; định nghĩa x trong môi trường cục bộ đối với f
      (+ x 3))
x
--> 1
```

```
(f)
--> 5
x
--> 1
; lời gọi f không làm thay đổi định nghĩa toàn cục của x
```

Người ta khuyên không nên sử dụng các định nghĩa hàm hay biến cục bộ bên trong thân của một hàm : kết quả nhiều khi không dự kiến trước được và phụ thuộc vào bộ diễn dịch Scheme đang sử dụng.

II.3.2.4. Tính không định kiểu của Scheme

Định nghĩa hàm của Scheme không sử dụng khai báo kiểu cho tham biến và cho kết quả trả về. Sự không tương thích về kiểu chỉ có thể được phát hiện ở thời điểm gọi thực hiện. Khi đó, các biến không được định kiểu nhưng các giá trị lại được định kiểu. Vì vậy, người lập trình phải để ý đến tính tương thích về kiểu cho các giá trị của biểu thức. Điều này có lợi nhưng cũng có những bất tiện.

Nhận xét :

Sự mềm dẻo về kiểu cho phép sử dụng hàm với nhiều dữ liệu khác nhau. Mọi dữ liệu sử dụng cho các phép toán trong thân hàm đều có cùng nghĩa. Không cùng phân biệt một tham đối là số nguyên, số thực hày là một số phức. Cùng một biến có thể biểu diễn lúc thì một số, lúc thì một chuỗi, v.v... trong khi các giá trị thì lại được định kiểu.

Tuy nhiên, không định kiểu làm mất tính an toàn, chính người lập phải kiểm tra kiểu chứ không phải là bộ biên dịch. Không định kiểu cũng làm mất tính hiệu quả, vì rằng sự nhận biết ban đầu về kiểu của các đối tượng cho phép bộ biên dịch sử dụng các phép toán thích hợp.

II.3.3. Cấu trúc điều khiển

II.3.3.1. Dạng điều kiện `if`

Cấu trúc điều kiện cơ bản nhất của Scheme là `if` có cú pháp như sau :

(if e s-then s-else)

Nếu giá trị của biểu thức `e` là `#f`, dạng `if` trả về giá trị của biểu thức `s-else`, ngược lại thì `s-then`. Nghĩa là `#f` cóvai trò *false*, còn mọi giá trị khác có vai trò *true* trong phép thử.

Ví dụ :

```
(define (abs-value nb)
   (if (<= 0 nb) nb (- nb)))
```

Chú ý dạng `if` chỉ luôn tính một trong hai biểu thức `s-then`, hoặc `s-else`. Vì vậy người ta gọi `if` là một dạng đặc biệt mà không phải là một hàm. Điều này cho phép định nghĩa hàm chia hai số tránh chia cho `0`.

```
(define (div-safe r1 R2)
   (if (zero? R2)
        "Sai: khong the chia cho back !"
        (/ r1 R2)))
```

Có thể sử dụng các dạng `if` lồng nhau, chẳng hạn ta định nghĩa hàm sau đây trả về dấu của một số, 1 nếu dương, –1 nếu âm và `0` nếu bằng không :

```
(defne (sign nb)
```

```
(if (< 0 nb)
     1
     (if (zero? nb) 0 -1)))
```

Phép `not` có thể định nghĩa tính khác không của một số. Chẳng hạn :

```
(define (not-zero nb)
  (not (zero? nb)))
```

Dạng điều kiện `cond`

Thay vì sử dụng các `if` lồng nhau, Scheme có dạng `cond` rất thuận tiện :

(cond (e1 s1)
...
(eN sN)
[(else sN+1)])

Các biểu thức e_1, e_2, ... được tính liên tiếp cho đến khi gặp một giá trị đúng, giả sử e_j, khi đó giá trị trả về của `cond` là giá trị của s-biểu thức s_j.

Để đảm bảo ít nhất có một phép thử thành công, người ta thường đặt vào cuối từ khóa `else`. Khi đó, s_{N+1} được tính nếu mọi e_1, ... , e_N đều sai.

Ví dụ, hàm `mention` sau đây trả về một kết quả tùy theo điểm thi `note` :

```
(define (mention note)
  (cond((>= note 8) "Gioi")
       ((>= note 7) "Kha")
       ((>= note 6) "Trung binh kha")
       ((>= note 5) "TB")
       (else "Kém")))
```

Bằng cách biểu diễn các kiểu giá trị được biết qua tên gọi của chúng, sau đây là một ví dụ về hàm trả về kiểu của một biểu thức Scheme.

```
(define (type-of s)
  (cond ((symbol? s) 'symbol)
        ((number? s) 'number)
        ((lôgích? s) 'boolean)
        ((string? s) 'string)
        (else 'unknowtype)))
(type-of 2376)
--> number
(type-of 'kiki)
--> symbol
(type-of "c'est une chaine.")
--> string
(type-of (> 1 2))
--> boolean
```

Các phép toán logic `and` và `or`

Các phép toán `and` và `or` dùng để tổ hợp các biểu thức Scheme (tương ứng với cách tính"ngắn mạch"). Giá trị :

(*and s1 s2 … sN*)

nhận được bằng cách tính lần lượt dãy các biểu thức s_1, s_2, ... và dừng lại ngay khi gặp một giá trị `#f` và trả về kết quả `#f`, nếu không, trả về giá trị cuối là s_N.

Ví dụ, nếu các số thực `a`, `b`, `c` là ba cạnh của một tam giác, thì biểu thức :

```
(and (< a (+ b c)) (< b (+ c a)) (c (+ a b))
```

trả về giá trị `#t`. Biểu thức sau tránh được lỗi chia cho `0` :

```
(and (< 2 8) (number? "yes") (+ 1 (/ 2 0)))
--> #f
```

Một cách tương tự, giá trị tương tự của :

(*or s1 s2 … sN*)

Nhận được bằng tính lần lượt biểu thức s_1, s_2, ... và dừng lại khi gặp một giá trị đúng `#t` để trả về giá trị `#t` này, nếu không, trả về giá trị `#f`.

```
(or (= 2 3) 10 (number? "yes"))
--> 10              ; vì 10 là giá trị đầu tiên khác #f
```

II.3.3.2. Biến cục bộ

1. Định nghĩa biến cục bộ nhờ dạng let

Khi tính một biểu thức, các kết quả trung gian cần phải được lưu giữ để tránh tính đi tính lại nhiều lần. Ví dụ, khi tính diện tích tam giác cạnh `a`, `b`, `c` theo công thức :

$\sqrt{p(p-a)\,(p-b)\,(p-c)}$ với p là nửa chu vi, p = (a+b+c)/2

ta cần lưu giữ giá trị p để chỉ phải tính một lần. Nếu dùng một hàm bổ trợ để tính p thì sẽ không tiện nếu như p chỉ sử dụng để tính diện tích tam giác, các hàm khác không sử dụng p sẽ không cần biết đến p.

Scheme có các dạng đặc biệt `let` cho phép lưu giữ các giá trị trung gian trong thân hàm chính. Cú pháp của `let` như sau :

(*let link body*)

phần liên kết `link` có dạng :

((*v1 s1*)…(*vk sk*))

Có nghĩa rằng trong suốt quá trình tính toán, thân hàm `body`, tên biến `vj` chỉ định giá trị biểu thức `sj`.

Ví dụ : Diện tích tam giác cạnh `a`, `b`, `c` được tính từ nửa chu vi p như sau :

```
(let ((p (/ (+ a b c) 2)))
   (sqrt (* p (- p a) (- p b) (- p c))))
```

Do cần phải định nghĩa nhiều liên kết `link` song song, nên người ta đặt tập hợp các liên kết $(v_1\ s_1)\ldots(v_k\ s_k)$ giữa hai dấu ngoặc để phân biệt với phần thân hàm `body`. Người ta thường viết dạng `let` theo cột dọc và phần thân hàm lại lệch qua trái cho dễ đọc :

(let ((*v*$_1$ *s*$_1$)
…
(*v*$_k$ *s*$_k$)
***body*)**

Ví dụ :

```
(let ((a 20) (b (* 4 8)) (c 10)
(* c (- a b)))
--> 120
```

Chú ý rằng các biểu thức s_k được tính trước và chỉ khi đó, các biến v_k mới được định nghĩa.

```
(define (f x y) (let ((x y) (y x)) (- x y)))
(f 2 5)
--> 3                ; không phải là 0, cũng không phải là –3
```

2. Phạm vi tự động của dạng let

Cần xem xét phạm vị hoạt động của các biến đã định nghĩa trong `let` :

```
(define a 10)
(let ((a 5) (b (* 2 a)))
    (+ a b))
--> 25
```

Giá trị toàn cục của `a` được nhìn thấy trong biểu thức định nghĩa `b`, và do đó `b=20`, nhưng trong thân `(+ a b)` thì `a = 5` do `body` nhìn thấy các v_j. Tuy nhiên, giá trị của biến cục bộ `b` lại không xác định ở ngoài `let` :

```
b
--> *** ERROR -- unbound variable: b
```

Một cách tổng quát, các biến v_j được định nghĩa trong `link` của `let` các định nghĩa khác chỉ khi tính thân `body` của `let` mà thôi.

Trong thân của một định nghĩa hàm, ta có thể sử dụng `let`. Chẳng hạn xây dựng hàm tính $\sqrt{a^2+b^2}$, ta tính a^2 và b^2 nhờ `let` như sau :

```
(define (pitagore a b)
    (let ((a2 (* a a)) (b2 (* b b)))
       (sqrt (+ a2 b2))))
(pitagore 3 4)
--> 5.
```

3. Liên kết biến theo dãy : dạng let*

Nếu cần liên kết các biến theo dãy, Scheme có dạng đặc biệt `let*` :

(let* ((*v*$_1$ *s*$_1$)… (*v*$_k$ *s*$_k$)) *body*)

Sự liên kết của `let*` thể hiện ở chỗ giá trị của v_i dùng để tính s_j, nếu $i<j$ (v_i đứng trước s_j).

```
(define a 2)
```

```
(let* ((a (* a a))     ; a = 4
       (b (* 3 a))     ; b = 12
       (c (* a b)))    ; c = 48
     (+ a b c ))
--> 64
```

Thật vậy, trong thân của `let*`, biến `a=4` do trước đó, `a=2`, biến `b=12` do `a=4`, biến `c=48` do `a=4` và `b=12`.

Dạng `let*` thực ra không thật cần thiết vì có thể được biểu diễn bởi các `let` lồng nhau. Ví dụ trên đây được viết lại theo dạng `let` như sau :

```
(let ((a (* a a)))
  (let ((b (* 3 a)))
    (let ((c (* a b)))
         (+ a b c))))
--> 64
```

Dạng `let` lồng nhau được dùng để làm rõ sự phụ thuộc giữa các biểu thức. Một cách tổng quát, quan hệ giữa `let` và `let*` như sau :

(let* ((v_1 s_1)
…
(v_k s_k))
body)

$\Leftrightarrow$

(let ((v_1 s_1))
(let ((v_2 s_2))
…
(let ((v_k s_k))
body) …))

II.3.3.3. Định nghĩa các vị từ

Như đã nói, tên các vị từ trong Scheme đều đặt một dấu chấm (`?`) hỏi ở sau cùng. NSD có thể định nghĩa các vị từ và cũng phải tuân theo quy ước này. Chẳng hạn ta định nghĩa lại vị từ kiểm tra một số nguyên là chẵn (even) hay lẻ (odd) mà không sử dụng các vị từ `even?` và `odd?` có sẵn trong thư viện của Scheme :

```
(define (is-even? n)
    (if (= n 0)
      #t
      (is-odd? (- n 1))))
(define (is-odd? n)
    (if (= n 0)
      #f
      (is-even? (- n 1))))
(is-even? 4)
--> #t
(is-odd? 5)
--> #t
```

Vị từ sau đây kiểm tra một số nguyên *n* trong phạm vi 2..100 có là số nguyên tố hay không ? Ta lập luận như sau : *n* không phải là số nguyên tố nếu *n* chia hết (thương số lớn hơn 1) cho các số 2, 3, 5, 7, ... *sqrt*(*n*). Vậy nếu $10<n\leq100$ thì *sqrt*(100) = 10 và do vậy, *n* không thể là bội của các số 2, 3, 5, 7.

Trước tiên ta viết vị từ kiểm tra *n* có là bội của các số 2, 3, 5, 7 hay không :

```
(define (multiple2357? n)
  (or (zero? (remainder n 2))
      (zero? (remainder n 3))
      (zero? (remainder n 5))
      (zero? (remainder n 7))))
```

Các số 2, 3, 5, 7 và các số nguyên trong phạm vi 100 không phải là bội của các số này đều là số nguyên tố :

```
(define (prime<=100? n)
  (if (or (= n 2)
          (= n 3)
          (= n 5)
          (= n 7)
          (not (multiple? n)))
    #t #f))
```

Sau đây ta viết vị từ kiểm tra một năm dương lịch đã cho có phải là năm nhuần hay không ? Một năm là nhuần nếu chia hết cho 400 hoặc nếu không phải thì năm đó phải chia hết cho 4 và không chia hết cho 100. Trước tiên ta viết vị từ kiểm tra một số này có chia hết một số khác hay không.

```
(define (divisibleBy? number divisor)
  (= (remainder number divisor) 0))
(define (isBissextile? year)
  (or (divisibleBy? year 400)
      (and (divisibleBy? year 4)
           (not (divisibleBy? year 100)))))
(isBissextile? 1900)
--> #f
(isBissextile? 2004)
--> #t
```

II.3.4. Sơ đồ đệ quy và sơ đồ lặp

II.3.4.1. Sơ đồ đệ quy

Có nhiều *sơ đồ đệ quy* (recursive schema) được ứng dụng quen thuộc trong lập trình, ngôn ngữ Scheme sử dụng *sơ đồ đệ quy nguyên thuỷ* (primitive) có cú pháp như sau :

```
(define (<name> <arg>)
  (if (zero? <arg>)
      <init-val>
      (<func> <arg> (<name> (- <arg> 1)))))
```

Ở đây, cả <init-val> và <func> đều chưa được định nghĩa theo hàm <name>.

Sơ đồ đệ quy được giải thích một cách tổng quát như sau :

- Có một hoặc nhiều cửa ra tương ứng với điều kiện đã được thoả mãn để dừng quá trình đệ quy : nếu <arg> = 0 thì cửa ra lấy giá trị trả về <init-val>.
- Có một hoặc nhiều lời gọi đệ quy, nghĩa là gọi lại chính hàm đó, sao cho trong mỗi trường hợp, giá trị của tham đối <arg> phải hội tụ về một trong các cửa ra để dừng, thông thường giảm dần, chẳng hạn tham đối mới là <arg>-1.

Phương pháp lập trình đệ quy mang tính tổng quát và tính hiệu quả khi giải quyết những bài toán tính hàm có độ lớn dữ liệu phát triển nhanh. Để áp dụng kỹ thuật đệ quy, luôn luôn phải tìm câu trả lời cho hai câu hỏi sau đây :

1. *Có tồn tại hay không các trường hợp đặc biệt của bài toán đã cho để nhận được một lời giải trực tiếp dẫn đến kết quả.*
2. *Có thể nhận được lời giải của bài toán đã cho từ lời giải của cũng bài toán này nhưng đối với các đối tượng nhỏ hơn theo một nghĩa nào đó ?*

Nếu có câu trả lời thì có thể dùng được đệ quy. Sau đây là một số ví dụ.

II.3.4.2. Ví dụ

1. Tính tổng bình phương các số từ 1 đến n

Xét hàm `SumSquare` tính tổng bình phương các số từ 1 đến *n* :

$SumSquare = 1 + 2^2 + 3^2 + ... + n^2$

Bằng cách nhóm $n-1$ số bình phương phía bên phải, tức từ 1 đến $(n-1)^2$, ta nhận được quan hệ truy hồi như sau :

$SumSquare(n) = SumSquare\ (n-1) + n^2$,

Quy ước $SumSquare\ (0) = 0$, ta định nghĩa hàm `SumSquare` trong Scheme như sau :

```
(define (SumeSquare n)
(if (zero? n)
     0
     (+ (SumeSquare (- n 1)) (* n n))))

(SumeSquare 3)
--> 14
```

Định nghĩa hàm như trên là *đệ quy* vì hàm cần định nghĩa lại gọi lại chính nó. Đó là lời gọi đệ quy `(SumSquare (- n 1))`.

2. Tính giai thừa

Để tính $n\ ! = 1 * 2 * ... * n$ với ta có :

$0\ ! = 1$

$n\ ! = n * (n-1)\ !$ với $n > 1$

Từ đó :

```
(define (fac n)
   (if (zero? n)   ; hoặc (<= n 0) để đảm bảo xử lý mọi số n
      1
      (* n (fac (- n 1)))))
(fac 5)
--> 120
```

3. Hàm Fibonacci

Có thể cùng lúc có nhiều lời gọi đệ quy trong thân hàm. Một hàm cổ điển khác thường được minh họa cho trường hợp này là hàm Fibonacci. Hàm `fib` được định nghĩa từ quan hệ truy hồi :

$f(0) = 0$ và $f(1) = 1$. Với $n \geq 0$, thì $f(n + 2) = f(n + 1) + f(n)$.

Người ta chứng minh được rằng :

$$f(n) = \frac{(\phi^n - \psi^n)}{\sqrt{5}} \quad \text{với } \phi = \frac{(1+\sqrt{5})}{2} \sim 1.6, \quad \text{và } \psi = \frac{(1-\sqrt{5})}{2} \sim 0.6$$

Ở đây, f được gọi là *số vàng* (golden number), với ψ là nghiệm của phương trình :

$$x^2 = 1 + x.$$

Hàm `fib` trong Scheme được định nghĩa như sau :

```
(define (fib n)
   (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))))
```

Ta nhận thấy rằng có nhiều số Fibonacci được tính nhiều lần :

```
(fib 4)
   ; = (+ (fib 3) (fib 2))
   ; = (+ (+ (fib 2) (fib 1)) (+ (fib 1) (fib 0)))
   ; = (+ (+ (+ (fib 1) (fib 0)) (fib 1)) (+ (fib 1) (fib 0)))
   ; = (+ (+ (+ 1 0) 1) (+ 1 0))
   ; = (+ (+ 1 1) 1)
   ; = (+ 2 1)
--> 3
```

4. Tính các hệ số nhị thức

Tính các hệ số nhị thức hay tổ hợp : số phép chọn k trong bộ n phân tử. Gọi $(n\ k)$ là hệ số nhị thức cần tìm, với $0 \le k \le n$, $n \ge 0$, ta có :

$$(n\ k) = \frac{n!}{(n-k)! * k!}$$

với : $(n\ 0) = (n\ n) = 1$

$(n+1\ k) = (n\ k) + (n\ k-1)$

Chẳng hạn $(5\ 2) = 10$. Ta xây dựng hàm `coef-binomial` như sau :

```
(define (coef-binomial n k)
    (cond ((zero? k) 1)
          ((= k n) 1)
          (else (+ (coef-binomial (- n 1) k)
                   (coef-binomial (- n 1) (- k 1))))))

(coef-binomial 5 2)
--> 10

(coef-binomial 0 0)
--> 1
```

II.3.4.3. Tính dừng của lời gọi đệ quy

Một lời gọi đệ quy phải luôn luôn chứa ít nhất một *cửa ra*, còn được gọi là *trường hợp cơ sở*, để trả về kết quả của hàm mà ở đó không phải là một lời gọi đệ quy. Vì nếu không, sẽ xảy ra hiện tượng gọi nhau (lặp đi lặp lại) vô hạn lần. Như vậy cách định nghĩa các hàm trong các ví dụ trên đây là hợp lý vì đều có cửa ra.

Chẳng hạn cửa ra của âënh nghéa hàm `fac` trên đây là `(fac 0)` ứng với 0!=1. Tuy nhiên, một thủ tục đệ quy có cửa ra (cửa ra không gọi đệ quy) chưa đủ điều kiện để kết thúc. Để minh hoạ ta hãy xét một cách khác định nghĩa hàm `fac` là hàm `bad-fact` như sau :

```
(define (bad-fact n)
   (if (= n 5)
        120
        (quotient (bad-fact (+ n 1)) (+ n 1))))
```

Với lời gọi n=3, (n<5), ta có :

```
(bad-fact 3)
   ; = (quotient (bad-fact 4) 4)
   ; = (quotient (quotient (bad-fact 5) 5) 4)
   ; = (quotient (quotient 120 5) 4)
   ; = (quotient 24 4)
--> 6
```

Kết quả đúng, nhưng với lời gọi n=6, (hoặc với mọi n>5), ta có :

```
(bad-fact 6)
   ; = (quotient (bad-fact 7) 7)
   ; = (quotient (quotient (bad-fact 8) 8) 7)
   ; ...
```

Do vi phạm điều kiện «hàm đệ quy gọi lại chính nó nhưng có giá trị tham đối nhỏ hơn», rõ ràng việc tính giai thừa của hàm `bad-fact` sẽ không bao giờ dừng lại.

Chẳng hạn, lời gọi `(fac -2)` sẽ kéo theo lời gọi `(fac -2)`, kéo theo lời gọi `(fac -3)`, v.v... Cả hai trường hợp đều gây ra thông báo lỗi (hoặc treo máy) :

```
*** ERROR -- Stack overflow
```

Chú ý đối với hàm `fac`, nếu lấy n <0, thì cũng gây ra một vòng lặp vô hạn. Từ những quan sát trên đây, người ta đặt ra ba câu hỏi sau :

1. *Có thể suy luận ngay trên các dòng lệnh của chương trình để chứng minh một định nghĩa hàm là đúng đắn ? Câu trả lời là **có thể**.*
2. *Có phải là ngẫu nhiên mà hàm `fac` được định nghĩa đúng còn hàm `bad-fact` thì không đúng ? Câu trả lời là **không phải**.*
3. *Có tồn tại hay không những sơ đồ lập trình áp dụng được cho các định nghĩa hàm để chạy đúng đắn ? Câu trả lời là **có**.*

Tất cả những vấn đề trên đều dựa trên khái niệm *quy nạp* (induction) mà phép lặp là một trường hợp đặc biệt[5]. Để chứng minh hàm P định nghĩa đúng (tính đúng) một hàm f nào đó, cần phải chứng minh P thoả mãn hai điều kiện là :

– P đúng đắn từng phần (partial correction).
– P dừng.

Trong trường hợp tính giai thừa, ta biết rằng $0! = 1$, và $n! = n*(n-1)!$. Bây giờ ta cần chứng minh tính đúng đắn từng phần của `fac` :

---

[5] Để chứng tỏ rằng một tính chất P nào đó là đúng đối với mọi số nguyên n, người ta lập luận như sau :
- Cơ sở quy nạp : P(0) là đúng.
- Giả thiết quy nạp : nếu P(n–1) là đúng, thì P(n) cũng đúng.

- `(fact 0) = 1 = 0!`
- Nếu $n > 0$, hoặc $n1 = n-1$, thì do giả thiết quy nạp `(fact n1)` = $n1!$, do đó
  `(fact n) = (* n (fact n1))` = $n * n1! = n!$

II.3.4.4. Chứng minh tính dừng

Mỗi lời gọi đệ quy, các tham đối được dùng tới phải «nhỏ hơn» so với tham đối ban đầu. Phương pháp thông dụng nhất để đảm bảo một hàm đệ quy dừng là tìm được một số nguyên dương giảm ngặt sau mỗi lần gọi đệ quy. Trong trường hợp hàm `SumSquare`, ta chỉ cần chọn tham đối n.

Hàm `fib` dừng vì mỗi lời gọi kéo theo hai lời gọi đệ quy, mỗi một trong chúng có tham đối mỗi lúc lại nhỏ hơn. Với $n \geq 2$, các số nguyên $n-1$ và $n-2$ giảm ngặt so với n và đều ≥ 0.

Trong trường hợp hàm `fac`, dãy các giá trị của n là giảm ngặt, và `(fac n)` chỉ gọi `(fact n1)` nếu $n \neq 0$, như vậy nếu $n \geq 0$, thì n1= $n-1 \geq 0$.

Từ đó nếu giá trị ban đầu của n là ≥ 0, thì dãy này là hữu hạn. Mặt khác, `fac` chỉ gọi đến những hàm *sơ cấp* (primitive) là những hàm dừng.

Từ đó suy ra rằng `(fact n)` = $n!$ với $n \geq 0$

Chứng minh tính đúng đắn từng phần của hàm `bad-fact` :

- `(bad-fact 0)` = `(quotient (bad-fact 1) 1)` = ... = 1
- Nếu `(bad-fact n1)` = n1 ! là đúng với n1 = $n + 1$, thì :
  `(bad-fact n)` = `(quotient (bad-fact n1) n1)` = n1 ! / n1 = $n!$

Ta suy ra rằng nếu `bad-fact` dừng, thì `bad-fact` tính đúng giai thừa cho các số n thoả mãn $0 \leq n \leq 5$. Tuy nhiên `bad-fact` không dừng với mọi $n > 6$.

Thực tế, người lập trình không sử dụng cách lập luận trên đây vì tính phức tạp và dài dòng (dài hơn hàm cần chứng minh). Tuy nhiên người lập trình vẫn cần phải biết để thuyết phục người khác vì không thể chứng minh tính đúng đắn một chương trình qua một vài ví dụ minh hoạ.

II.3.4.5. Sơ đồ lặp

Ta lấy lại phép tính giai thừa : để tính `fac 3`, trước tiên bộ diễn dịch Scheme phải tính `(fact 2)`, muốn vậy cần ghi nhớ để lần sau thực hiện phép nhân. Tương tự, cần phải tính `(fact 1)`, trước khi tính `(* 2 (fact 1))`. Như vậy đã có một chuỗi các phép tính khác nhau, tăng tuyến tính với n. Người ta gọi đây là *quá trình đệ quy tuyến tính* (linear recursive processus).

Trong trường hợp tính hàm `fib`, lúc đầu Scheme hoãn thực hiện một phép cộng để tiến hành hai lời gọi đệ quy mà không thể tính toán đồng thời. Người ta gọi đây là *quá trình đệ quy dạng cây* (tree recursive processus). Rõ ràng với kiểu quá trình này, hàm `fib` đòi hỏi một chi phí tính toán đáng kể. Bây giờ ta xét một cách tính giai thừa khác :

```
(define (fact n)
  (define (i-fact p r)
    (if (= p 0)
        r
        (i-fact (- p 1) (* p r))))
  (i-fact n 1))
```

Áp dụng :

```
(fact 3)
   ; = (i-fact 3 1)
   ; = (i-fact 2 3)
   ; = (i-fact 1 6)
   ; = (i-fact 0 6)
--> 6
```

Tại mỗi giai đoạn, không có phép tính nào bị hoãn và do đó không cần ghi nhớ. Ta có thể làm dừng quá trình tính bằng cách chỉ cần ghi nhớ một số giá trị cố định (là `p` và `r`), rồi dùng lại sau đó.

Người ta gọi quá trình tính trên đây là *quá trình lặp* (iterative processus). Người ta cũng nói rằng hàm là đệ quy kết thúc.

Sử dụng quá trình lặp rất có hiệu quả về bộ nhớ, vì vậy mà trong hầu hết các ngôn ngữ lập trình cổ điển, các ngôn ngữ mệnh lệnh, người ta định nghĩa các cấu trúc cú pháp cho phép thực hiện các quá trình lặp (các vòng lặp `while`, `for`...).

Bộ diễn dịch Scheme có khả năng xử lý một quá trình lặp trong một không gian nhớ không đổi nên người lập trình nên tận dụng khả năng này.

Một quá trình lặp cũng được đặc trưng bởi một *bất biến* (invariant) : đó là một quan hệ giữa các tham đối. Quan hệ luôn được giữ không thay đổi khi tiến hành các lời gọi liên tiếp. Quan hệ này cho phép chứng minh tính đúng đắn của chương trình.

Chẳng hạn, ta tính bất biến của hàm `i-fact` :

$p = 0$ và $p! * r = constant = p_0! * r_0$

ở đây, p_0 và r_0 là các giá trị trong lời gọi chính.

Giả sử rằng lời gọi đệ quy bảo toàn quan hệ này :

$p_0! * r_0 = (p-1)! * (p! * r) = p! * r$

Hơn nữa, tính chất này còn đúng trong lời gọi chính : hàm `i-fact` dừng với giá trị $p = 0$ và do vậy ta có $r = p_0! * r_0$, chính là kết quả trả về của `i-fact`.

Lời gọi chính là `(i-fact n 1)`, `n=0`, kết quả của `(fact n)` sẽ là $n! * 1 = n!$.

Người ta luôn luôn có thể chuyển một hàm đệ quy tuyến tính thành đệ quy kết thúc, bằng cách định nghĩa một hàm phụ trợ có một tham đối bổ sung dùng để tích luỹ các kết quả trung gian (biến `r` trong hàm `i-fact`).

Sơ đồ lặp tổng quát để định nghĩa một hàm `fname` như sau :

```
(define (<fname> <arg>)
   (define (<i-name <arg> <result>)
      (if (zero? <arg>)
         <result>
         (<i-name> (- <arg> 1) (<func> <arg> <result>))))
   (<i-name> <arg> <init-val>))
```

Người ta cũng có thể chuyển một hàm đệ quy dạng cây thành đệ quy kết thúc, lúc này thời gian tính hàm được rút gọn một cách ngoạn mục ! Chẳng hạn ta viết lại hàm tính các số Fibonacci theo kiểu đệ quy kết thúc :

```
(define (fib n)
  (define (i-fib x y p) ; x=fib(n-p+1),y=fib(n-p)
    (if (= p 0)
      y
      (i-fib (+ x y) x (- p 1))))
  (i-fib 1 0 n))
(fib 100)
--> 354224848179261915075
(fib 200)
--> 280571172992510140037611932413038677189525
```

Chú ý rằng hàm tính Fibonacci bây giờ có chi phí tuyến tính !!!

II.3.5. Vào/ra dữ liệu

Cho đến lúc này, ta mới tạo ra các hàm mà chưa nêu lên cách vào/ ra dữ liệu. Thực tế ta đã lợi dụng vòng lặp tương tác như sau :

... → đọc dữ liệu → tính hàm → in ra kết quả → ...

Một trong những nguyên lý của lập trình Scheme là tránh trộn lẫn các việc in ra kết quả với việc tính toán trong cùng một định nghĩa hàm. Tuy nhiên, trong một số trường hợp, vẫn có thể trộn lẫn các quá trình vào/ra vào bên trong của một hàm.

1. Đọc vào dữ liệu : read

Hàm `(read)` đọc một dữ liệu bất kỳ của Scheme từ bàn phím (là dòng vào hiện hành) và trả về giá trị đọc được mà không xảy ra một tính toán nào.

```
(read)
```

Sau khi thực hiện lời gọi này, Scheme bắt đầu ở trạng thái chờ người sử dụng gõ vào từ bàn phím. Giả sử người sử dụng gõ vào một biểu thức có chứa nhiều dấu cách thừa (giữa + và 2, giữa 2 và 3) :

```
(+ 2 3)
```

Lập tức, Scheme sẽ trả về kết quả là một biểu thức đúng (không còn dấu cách thừa) mà không tính biểu thức này.

```
--> (+ 2 3)
```

Dạng tổng quát của hàm đọc dữ liệu của Scheme có chứa một tham biến tùy chọn là một *dòng vào* (flow) thường được dùng khi đọc các tệp dữ liệu `(read [flow])`. Khái niệm về các dòng vào/ra sẽ xét ở chương sau.

2. In ra dữ liệu : write và display

Để in ra giá trị của một s-biểu thức bất kỳ ra màn hình (là dòng ra hiện hành), Scheme có hàm `(write s)` như sau :

```
(write "Bonjour tout le monde!")
--> "Bon jour tout le monde!"
```

Giá trị in ra bởi `write` là một dữ liệu Scheme và thấy được bởi `read`, tuy nhiên, dữ liệu đưa ra phải phù hợp với thói quen của người đọc. Chẳng hạn, để đưa ra các chuỗi ký tự không có dấu nháy, Sheme có hàm `display` tương tự `write` nhưng đối với các chuỗi ký tự thì không in ra các dấu nháy :

```
(display "Hello, world!")
--> Hello, world!
```

Để qua dòng mới, Sheme có hàm (`newline`) không có tham đối :

```
(newline)
```

3. Xây dựng vòng lặp có menu

Sau đây, ta xây dựng một hàm gây ra một vòng lặp in ra một menu, và bằng cách trả lời bởi gõ vào một số, hàm thực hiện một công việc cho đến khi người ta sử dụng gõ `0` để kết thúc vòng lặp. Khi gõ sai, hàm đưa ra một câu thông báo yêu cầu gõ lại.

```
(define (menu)
   (display "Enter 0 to quit, 1 to job1, 2 to job2.")
   (let ((rd (read)))
      (cond ((equal? rd 0) (display "Good bye!"))
            ((equal? rd 1) (display "I work the job 1.")
                   (newline ) (menu))
            ((equal? rd 2) (display "I work the job 2.")
                   (newline) (menu))
            (else (display "restart :un known command.")
                   (newline) (menu)))))
(menu)
Enter 0 to quit, 1 to job1, 2 to job2.2
I work the job 2.
Enter 0 to quit, 1 to job1, 2 to job2.1
I work the job 1.
Enter 0 to quit, 1 to job1, 2 to job2.0
Good bye!;Evaluation took 13255 mSec (0 in gc) 27 cells
work, 130 bytes other
```

Chú ý các số 2, 1 và `0` sau dấu chấm là do người sử dụng gõ vào. Trong hàm `menu` trên đây, ta thấy xuất hiện hiệu ứng phụ trong thân một hàm, nghĩa là biểu thức không được sử dụng giá trị mà chỉ để in ra.

Tóm tắt chương 2

- Scheme là một ngôn ngữ lập trình hàm thuộc họ Lisp. Ngôn ngữ Scheme rất thích hợp để mô tả các khái niệm trừu tượng và các công cụ lập trình.
- Scheme có cú pháp đơn giản, dễ hiểu, dễ lập trình. Hoạt động cơ bản trong lập trình Scheme là tính giá trị các biểu thức. Scheme làm việc theo chế độ tương tác.
- Scheme (cũng như hầu hết các ngôn ngữ họ Lisp) rất thuận tiện khi giải các bài toán xử lý ký hiệu, tính toán hình thức. Sử dụng Scheme có thể tính đạo hàm của một hàm, tính tích phân, tìm nghiệm các phương trình vi phân...
- Scheme có hai loại kiểu dữ liệu là kiểu đơn giản và kiểu phức hợp. Kiểu dữ liệu đơn giản gồm kiểu số, kiểu lôgích, kiểu ký tự và kiểu ký hiệu. Kiểu dữ liệu phức hợp gồm danh sách và cây.
- Scheme sử dụng cách viết biểu thức dạng tiền tố. Nghĩa là viết các phép toán rồi mới đến các toán hạng và đặt tất cả trong cặp dấu ngoặc.
- Scheme tính giá trị của một biểu thức ngay khi biểu thức đó đã đúng đắn về mặt cú pháp, hoặc thông báo lỗi sai.
- Trong Scheme, người ta đưa vào khái niệm s-biểu thức là tất cả các kiểu dữ liệu có thể nhóm lại với nhau đúng đắn về mặt cú pháp.
- Scheme sử dụng nhiều kiểu định nghĩa cho biến, cho hàm.
- Trong Scheme phép đệ quy được sử dụng triệt để. Người lập trình có thể sử dụng sơ đồ đệ quy hoặc phép lặp để định nghĩa một hàm. Trong một số trường hợp, sử dụng phép lặp sẽ có chi phí thấp hơn.
- Một chương trình Scheme là một dãy các định nghĩa hàm góp lại để định nghĩa một hoặc nhiều hàm phức tạp hơn.
- Trong Scheme, người ta thường khó phân biệt chương trình (hàm) và dữ liệu (danh sách) do cả chương trình và dữ liệu đều có cách viết sử dụng các cặp dấu ngoặc lồng nhau nhiều mức.
- Trong một số phiên bản trình thông dịch, mỗi danh sách hay ký hiệu thường được đặt ở phía trước một dấu nháy đơn (quote) để dễ phân biệt với hàm.
- Khả năng vào/ra dữ liệu trong Scheme rất hạn chế.

Bài tập chương 2

1. Giải thích các biểu thức số học sau đây, sau đó tính giá trị và so sánh kết quả :

```
(+ 23 (- 55 44 33) (* 2 (/ 8 4)))
(define a 3)
a
(/ 6 a)
(define b (+ a 1))
(+ a b (* a b))
```

2. Giải thích các biểu thức lôgic sau đây, sau đo tính giá trị và so sánh kết quả (có thể sử dụng hai biến *a* và *b* trong bài tập 1) :

```
(= 2 3)
(= a b)
(not (or (= 3 4) (= 5 6)))
(+ 2 (if (> a b) a b))
```

3. Giải thích các biểu thức điều kiện sau đây, sau đo tính giá trị và so sánh kết quả :

```
(if (= 1 1) "waaw" "brrr")
(if (= 4 4) 5 6)
(if (> a b) a b)
(if (and (> b a) (< b (* a b))) b a)
(+ 2 (if (> a b) a b))
((if (< a b) + -) a b)
(cond ((= 1 1) "waaw 1")
      ((= 2 2) "waaw 2")
      ((= 3 3) "waaw once more")
      (else "waaw final"))
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

4. Viết dạng ngoặc tiền tố của các biểu thức :

   a) $(p-a)\,(p-b)\,(p-c)$

   b) $1 + 2x2 + 3x3$

   c) $\dfrac{\sin(x+y)\cos(x-y)}{1+\cos(x+y)}$

5. Các biểu thức sau đây có đúng về mặt cú pháp hay không?

```
f (x y z)    (f) (x y z)    (f)            ((f f))
()           ff             ((a) (b) (c))
```

6. Giải thích các s-biểu thức sau đây, sau đo tính giá trị và so sánh kết quả :

```
(and #f (/ 1 0))
(if #t 2 (/1 0))
(if #f 2 (/ 1 0))
(and #t #t #f (/ 1 0))
(and #t #t #t (/ 1 0))
```

7. Viết hàm yêu cầu người sử dụng gõ vào một số nằm giữa 0 và 1000 để trả về giá trị bình phương của số đó. Đặt hàm này vào trong một vòng lặp với menu.
8. Viết hàm sử dụng menu để giải hệ phương trình đại số tuyến tính :
   $ax + by = 0$
   $cx + dy = 0$
9. Viết hàm tính giá trị tiền phải trả từ giá trị không thuế (duty-free). Biết rằng hệ số thuế VAT là 18,6%.
10. Viết hàm tính chiều cao *h* của tam giác theo các cạnh *a*, *b*, c cho biết diện tích tam giác được tính :
    $$S = \sqrt{p(p-a)\,(p-b)\,(p-c)}$$
    với p là nửa chu vi (sử dụng hàm bổ trợ tính để tính p).
11. Viết biểu thức tính nghiệm phương trình bậc hai $ax^2 + bx + c = 0$.
12. Cho biết giá trị của a=10, hãy tính :
```
(let ((a (* a a)) (b (* 4 5)) (c (* a 5)))
   (+ a b c))
```
13. Tính giá trị hai biểu thức sau :
```
(let ((x 5))
     (let* ((y (+ x 10)) (z (* x y)))
        (+ x y z)))
(let ((x 4))
   (if (= x 0) 1 (let ((x 10)) (* x x))))
```
14. Viết biểu thức Scheme để tính giá trị :
    $$\frac{\sqrt{x^2 + y^2} - \sqrt{x^2 - y^2}}{1 + \sqrt{x^2 + y^2} + \sqrt{x^2 - y^2}}$$
    khi biết giá trị của *x, y*
15. Viết hàm `(sum n)` = 1 + 1/2 +...+ 1/*n* với *n* nguyên, $n > 0$
16. Viết hàm `(power n x)` = x^n với *x* bất kỳ và *n* nguyên. Cho $x^n = x * x^{n-1}$. Mở rộng cho trường hợp $n < 0$.
17. Tương tự bài tập 16 nhưng sử dụng phương pháp chia đôi :
    $x^0 = 1$, $x^n = x * x^{n-1}$ nếu *n* lẻ và $x^n = (x^{n/2})^2$ nếu *n* chẵn.
18. Viết vị từ kiểm tra một năm đã cho có phải là năm nhuần không ?
19. Viết hàm `(nbsec h m s)` tính ra số giây từ giờ, phút, giây đã cho. Ví dụ :
```
(nbsec 10 3 45)
--> 36225
```
20. Viết hàm `(Hanoi n A B C)` giải bài toán «Tháp Hà Nội». Ví dụ :
```
(Hanoi 2 'A 'B 'C)
-->  Move A to B
     Move A to C
     Move B to C
```

CHƯƠNG III. KIỂU DỮ LIỆU PHỨC HỢP

If worms have the power of acquiring some notion,
however rude, of the shape of an object and of their burrows,
as seems to be the case, they deserve to be called intelligent.
Charle R. Darwin (*Vegetable Mould*, 1881)

Kiểu dữ liệu phức hợp trong Scheme gồm kiểu *chuỗi* (string), kiểu *vectơ* (vector), kiểu *bộ đôi* (doublet), kiểu danh sách. Ngoài ra, Scheme còn một số kiểu dữ liệu phức hợp khác. Kiểu dữ liệu *thủ tục* (procedure) chỉ định các biến chứa giá trị trả về của hàm. Kiểu dữ liệu *cổng* (port) chỉ định các cổng vào-ra tương ứng với các tệp và các thiết bị vào-ra (bàn phím, màn hình). Cuối cùng, tất cả các kiểu dữ liệu vừa xét trên đây, kể cả kiểu đơn giản, đều được Scheme gom lại thành một giuộc được gọi là kiểu s-biểu thức.

Sau đây, ta sẽ lần lượt trình bày các kiểu dữ liệu chuỗi, vectơ, bộ đôi và danh sách. Trong phần trình bày kiểu dữ liệu bộ đôi, chúng ta sẽ nghiên cứu khái niệm *trừu tượng hoá dữ liệu* (data abstraction).

III.1 Kiểu chuỗi

Chuỗi là một dãy các ký tự bất kỳ được viết giữa một cặp dấu nháy đôi (double-quotes). Giá trị của chuỗi chính là bản thân nó. Ví dụ sau đây là một chuỗi :

```
"Cha`o ba.n !"
--> "Cha`o ba.n !"
```

Có thể đưa vào trong chuỗi dấu nháy đôi, hay dấu \ (reverse solidus), bằng cách đặt một dấu \ phía trước, chẳng hạn :

```
(display "two \"quotes\" within.")
--> two "quotes" within.
```

Thư viện Scheme có hàm `string` cho phép ghép liên tiếp các ký tự thành một chuỗi :

```
(string #\S #\c #\h #\e #\m #\e)
--> "Scheme"
```

Ta có thể định nghĩa một biến nhận giá trị kiểu chuỗi :

```
(define greeting "Scheme ; cha`o ba.n !")
```

; Chú ý dấu ; trong một chuỗi không có vai trò là dấu chú thích.

Để tiếp cận đến một ký tự của chuỗi ở một vị trí bất kỳ, cần sử dụng hàm `string-ref`. Chỉ số chỉ vị trí ký tự là một số nguyên dương, tính từ 0. Chỉ số hợp lệ lớn nhất của chuỗi đã cho là chiều dài của chuỗi trừ đi 1 :

```
(string-length greeting)
--> 21
```

```
(string-ref greeting 0)
--> #\S
(string-ref greeting 20)
--> #\!
(define Str     ; sử dụng hàm ghép liên tiếp các chuỗi
    (string-append "Chuoi \""
                greeting
                " \" co do dai 21."))
(display Str)
--> Chuoi "Scheme ; cha`o ba.n ! " co do dai 21.
```

Vị từ `string?` dùng để kiểm tra kiểu chuỗi :

```
(string? greeting)
--> #t
```

Sử dụng hàm `make-string`, ta có thể tạo ra một chuỗi có độ dài bất kỳ và chứa các ký tự bất kỳ :

```
(define a-5-long-string (make-string 5))
a-5-long-string
--> "?????"
(define a-5-asterisk-string (make-string 5 #\*))
a-5-asterisk-string
--> "*****"
```

Hàm `string-set!` dùng để thay thế một ký tự trong chuỗi đã cho tại một vị trí bất kỳ cho bởi chỉ số :

```
(string-set! a-5-long-string 0 #\c)
a-5-long-string
--> "c????"
```

Trên đây ta đã sử dụng các hàm xử lý chuỗi :

(string? Str)

Trả về `#t` nếu `str` là một chuỗi, nếu không trả về `#f`.

(string-append str$_1$ str$_2$ ...)

Trả về chuỗi ghép liên tiếp các chuỗi `str`$_1$, `str`$_2$.

(make-string k [char])

Trả về một chuỗi mới có độ dài *k*. Nếu có tham biến `char`, thì tất cả các ký tự của chuỗi sẽ là `char`, nếu không nội dung của chuỗi sẽ không được xác định.

(string-length str)

Trả về độ dài của chuỗi `str`.

(string-ref str k)

Trả về ký tự thứ `k` của chuỗi `str`. Giá trị của `k` phải hợp lệ.

(string-set! string k char)

Đặt giá trị ký tự thứ `k` của chuỗi `str` bởi `char`. Giá trị của `k` phải hợp lệ.

Sau đây là một số hàm xử lý chuỗi khác của Scheme :

Các vị từ so sánh chuỗi :

(string=? str$_1$ str$_2$)

(*string<? str₁ str₂*)
(*string>? str₁ str₂*)
(*string<=? str₁ str₂*)
(*string>=? str₁ str₂*)

trả về `#t` nếu thỏa mãn quan hệ thứ tự từ vựng =, <, >, <=, >= giữa các chuỗi (có phân biệt chữ hoa chữ thường), trả về `#f` nếu không thỏa mãn. Phép so sánh dựa trên các vị từ so sánh ký tự `char=?`, `char<?`, `char>?`, `char<=?`, và `char>=?`. Hai chuỗi được xem như tương đương về mặt từ vựng nếu chúng có cùng chiều dài và bao gồm dãy các ký tự giống nhau tương ứng với `char=?`.

Các vị từ sau đây cũng cho kết quả tương tự nhưng phép so sánh chuỗi không phân biệt chữ hoa chữ thường :

(*string-ci=? str₁ str₂*)
(*string-ci<? str₁ str₂*)
(*string-ci>? str₁ str₂*)
(*string-ci<=? str₁ str₂*)
(*string-ci>=? str₁ str₂*)

Phép so sánh được dựa trên các vị từ so sánh ký tự `char-ci=?`, `char-ci<?`, `char-ci>?`, `char-ci<=?`, và `char-ci>=?`.

Hàm :

(*string-copy str*)

trả về chuỗi mới là bản sao (copy) của `str` :

```
(string-copy "abc")
--> "abc"
```

Hàm :

(*substring str k l*)

trả về bản sao của `str` kể từ ký tự có chỉ số `k` (kể cả `k`) đến ký tự có chỉ số `l` (không kể `l`). Các giá trị của `k` và `l` phải hợp lệ.

```
(substring "Hello, World!" 0 1)
--> "H"
(substring "Hello, World!" 7 12)
--> "World"
```

Hàm `substring` có thể định nghĩa lại như sau :

```
(define (my-substring s1 m n)
    (let ((s2 (make-string (- n m))))
       (do ((j 0 (+ j 1)) (i m (+ i 1)))
           ((= i n) s2)
         (string-set! s2 j (string-ref s1 i)))))
(my-substring "Hello, World!" 7 12)
--> "World"
```

Trong hàm `my-substring` sử dụng cấu trúc lặp `do` có cú pháp như sau :

(*do* ((*var₁ init₁ step₁*)
...)
(*test expr* ...) *command* ...)

Cấu trúc lặp `do` hoạt động tương tự như lệnh `do` trong các ngôn ngữ mệnh lệnh : biến điều khiển var_1 nhận giá trị khởi động $init_1$ có bước thay đổi $step_1$ (nếu có) cho mỗi lần lặp. Khi điều kiện lặp `test` chưa được thoả mãn (false), biểu thức `command` (nếu có) được thực hiện. Khi `test` được thoả mãn (true), các biểu thức `expr` được tính (từ trái qua phải) đồng thời là giá trị trả về.

```
(define L '(1 3 5 7 9))
(do ((L L (cdr L))
     (sum 0 (+ sum (car L))))
   ((null? L) sum))
--> 25
```

III.2 Kiểu dữ liệu vectơ

Vectơ là kiểu dữ liệu gồm một dãy các phần tử tương tự kiểu chuỗi, nhưng mỗi phần tử có thể có kiểu bất kỳ, không hoàn toàn ký tự. Mỗi phần tử của vectơ lại có thể là một vectơ, từ đó tạo thành vectơ có nhiều chiều (multidimensional vector). Trong Scheme, các phần tử của vectơ được đặt trong một cặp dấu ngoặc `()`, ngay sát trước cặp dấu ngoặc có đặt một dấu `#` (number sign).

```
'#(999 #\a "Kiki" (1 2 3) 'x)
--> '#(999 #\a "Kiki" (1 2 3) 'x)
```

Sử dụng hàm `vector` để tạo ra một vectơ các số nguyên như sau :

```
(vector 1 2 3 4 5)
--> '#(1 2 3 4 5)
(vector 'x 'y 'z)
--> '#(x y z)
```

Hàm `make-vector` để tạo ra một vectơ có độ dài ấn định trước :

```
(make-vector 3)
--> '#(#{Unspecific} #{Unspecific} #{Unspecific})
(make-vector 3 "Kiki")
--> '#("Kiki" "Kiki" "Kiki")
```

Tương tự kiểu chuỗi, các hàm `vector-ref` và `vector-set!` dùng để tiếp cận và thay đổi tương ứng từng phần tử của vectơ. Vị từ `vector?` dùng để kiểm tra kiểu vectơ :

```
 (vector? '#(x y z))
--> #t
```

Sử dụng cấu trúc `do`, ta tạo ra một vectơ các số tự nhiên như sau :

```
(do ((v (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) v)
   (vector-set! vec i i))
--> '#(0 1 2 3 4)
```

III.3 Kiểu dữ liệu bộ đôi

III.3.1. Khái niệm trừu tượng hoá dữ liệu

Trừu tượng hoá thủ tục (procedure abstraction) là xây dựng các ứng dụng phức tạp từ những thao tác đơn giản, bằng cách che dấu trong chừng mực có thể những chi tiết xử lý. *Trừu tượng hoá dữ liệu* cũng nhằm mục đích định nghĩa một lớp dữ liệu phức tạp và cách thao tác trên các dữ liệu đó mà không quan tâm đến cách biểu diễn và cài đặt chúng trong máy vật lý như thế nào. Phương pháp trừu tượng hoá dữ liệu được ứng dụng rộng rãi trong lập trình hướng đối tượng.

Một *cấu trúc dữ liệu trừu tượng* hay *kiểu dữ liệu trừu tượng* được định nghĩa, hay được *đặc tả* (specification) bởi 4 thành phần : tên kiểu dữ liệu trừu tượng (types), các định nghĩa hàm (functions), các điều kiện đầu (preconditions) nếu có và các tiên đề (axioms). Hai thành phần đầu mô tả cú pháp về mặt cấu trúc thuộc tính của kiểu dữ liệu trừu tượng, hai thành phần sau mô tả ngữ nghĩa.

Thành phần **`functions`** liệt kê các khuôn mẫu hàm (function pattern). Mỗi khuôn mẫu hàm, còn được gọi là một *ký pháp* (signature), có dạng một ánh xạ cho biết kiểu của các tham đối và của kết quả như sau :

Tên-hàm : miền-xác-định –> miền-trị

Người ta thường phân biệt ba loại hàm là *hàm kiến tạo* (constructor) để tạo ra kiểu dữ liệu mới, *hàm tiếp nhận* (accessor) để trích ra các thuộc tính và *hàm biến đổi* (transformator) để chuyển kiểu dữ liệu.

Do các hàm không phải luôn luôn xác định với mọi dữ liệu nên người ta cũng phân biệt các *hàm toàn phần* (total functions) và các *hàm bộ phận* (partial functions). Trong trường hợp các hàm là bộ phận, người ta cần cung cấp các điều kiện đầu là các ràng buộc trên miền xác định của hàm. Một lời gọi hàm không tuân theo điều kiện đầu đều dẫn đến sai sót (chia cho *0*, tính căn bậc hai của một số âm, v.v...). Chẳng hạn, điều kiện đầu của hàm tạo số hữu tỷ `(create-rat n d)` là mẫu số $d \neq 0$. Thành phần **`preconditions`** có thể vắng mặt nếu không có điều kiện đầu.

Thành phần **`axioms`** mô tả các chức năng vận dụng hàm, phải được khai báo rõ ràng, đầy đủ và dễ hiểu. Mỗi vận dụng hàm có dạng một mệnh đề lôgích hợp thức luôn có giá trị `true`, nghĩa là một hằng đúng (tautology). Khi cần sử dụng các biến, người ta sử dụng từ khoá **`var`** để khai báo trước tên các biến, tương tự trong các ngôn ngữ lập trình mệnh lệnh như C, Pascal...

Chú ý rằng kiểu tiền định `number` cũng là một kiểu trừu tượng : bởi vì ta có thể biết các phép toán số học trên các số như +, –, *, ... nhưng ta lại không biết cách biểu diễn chúng trong bộ nhớ. Chẳng hạn sau đây là đặc tả kiểu số tự nhiên $\mathcal{N}$(natual number) :

```
types nat
functions
   0 : -> nat   ; 0 là số tự nhiên đầu tiên, là một hằng (constant)
   succ : nat -> nat    ; hàm successor lấy phần tử kế tiếp
   + : nat × nat -> nat
   - : nat × nat -> nat
   * : nat × nat -> nat
   ^ : nat × nat -> nat
   = : nat × nat -> boolean
```

```
axioms var X, Y : nat
   X + 0 == X
   X + succ(Y) == succ(X + Y)
   0 - X ⇔ 0        ; chỉ làm việc với các số tự nhiên
   X - 0 ⇔ X
   succ(X) - succ(Y) ⇔ X - Y
   X * 0 ⇔ 0
   X * succ(Y) ⇔ X + (X * Y)
   X ^ 0 ⇔ succ(0)
   X ^ succ(Y) ⇔ X * (X ^ Y)
   0 = 0 ⇔ true
   succ(X) = 0 ⇔ false
   0 = succ(X) ⇔ false
   succ(X) = succ(Y) ⇔ X = Y
```

III.3.2. Định nghĩa bộ đôi

Trong ngôn ngữ Scheme, một bộ đôi, hay còn được gọi là *cặp có dấu chấm* (dotted pair), là kiểu dữ liệu phức hợp tương tự bản ghi gồm một cặp giá trị nào đó có thứ tự. Có một dấu chấm để phân cách hai giá trị lưu giữ trong bộ đôi và các dấu cách để phân cách giữa dấu chấm và các giá trị. Phần tử (trường) thứ nhất được gọi là `car`, phần tử thứ hai `cdr`.

Các tên `car` và `cdr`[1] xuất hiện khi John Mc. Carthy đề xuất ngôn ngữ Lisp chạy trên IBM-704 năm 1956 và được giữ lại theo truyền thống.

Giả sử `any` là một kiểu dữ liệu nào đó của Scheme, ký pháp của cấu trúc bộ đôi `doublet` được đặc tả như sau :

```
types doublet
functions
   cons  : any × any    -> doublet
   car   : doublet -> any
   cdr   : doublet -> any
   pair? : any     -> boolean
axioms var a, y : any
   car(cons(x, y)) = x
   cdr(cons(x, y)) = y
```

Từ kiểu trừu tượng bộ đôi, để tạo ra bộ đôi, người ta sử dụng `cons` :

(cons s1 s2)

--> (<*giá trị của* s1> . <*giá trị của* s2>)

Sau đây là một ví dụ về một bộ đôi : :

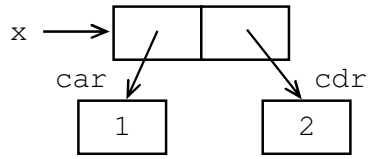
```
(cons 'year 2004)       ; gồm một ký hiệu và một số
--> (year . 2004)
(cons 29 #t)            ; gồm một số và một trị lôgích
--> (29 . #t)
```

Định nghĩa biến có giá trị là một bộ đôi :

[1] **car** = **c**ontent of the **a**ddress **r**egister, **cdr** = **c**ontent of the **d**ecrement **r**egister.

```
(define x (cons 1 2))
x
--> (1 . 2)
(car x)
--> 1
(cdr x)
--> 2
```

Trong máy, bộ đôi được biên dịch nhờ một cặp con trỏ trỏ đến `car` và `cdr`.



Hình III.1. Biểu diễn các bộ đôi nhờ con trỏ.

Các thành phần `car` và `cdr` của bộ đôi có vai trò đối xứng nhau : hàm `car` tương ứng với thành phần đầu tiên và hàm `cdr` tương ứng với thành phần thứ hai. Ta có quan hệ tổng quát như sau :

```
(car (cons a b)) = <giá trị của a>
(cdr (cons a b)) = <giá trị của b>
(car x)
--> 1   x = (1 . 2)
(cdr x)
--> 2
(cdr (car y))
--> 2
```

Vị từ `pair?` dùng để kiểm tra s-biểu thức có phải là một bộ đôi hay không ?

```
(define x '(1 . 2))
(pair? x)
--> #t          ; x = (1 . 2) là một bộ đôi
(pair? (cdr x))
--> #f          ;(cdr x) = 2 không phải là một bộ đôi
```

Vị từ `eq?` hay được dùng để kiểm tra hai bộ đôi có đồng nhất với nhau không. Thực chất, `eq?` kiểm tra tính đồng nhất của hai con trỏ.

Ngữ nghĩa của hàm này như sau :

```
axioms var x, y : doublet ; a, b : any
   (x = cons(a, b)) ∧ (y = x) ⇒ (eq?(x) = eq?(y))
```

Ví dụ :

```
(define x (cons 1 2))
(define y x) ; cả x và y cùng trỏ đến một bộ đôi
(eq? x y)
--> #t
```

Tuy nhiên :

```
(eq? x (cons 1 2))
--> #f
```

bởi vì x không trỏ đến bộ đôi mới được tạo ra bởi cons !

III.3.3. Đột biến trên các bộ đôi

Nhờ cons, car, cdr, ta đã định nghĩa các bộ đôi như là dữ liệu sơ cấp. Từ đó, ta có thể thay đổi bộ đôi nhờ các hàm đột biến set-car! và set-cdr!. Các hàm này không tạo ra bộ đôi mới, nhưng làm thay đổi bộ đôi đang tồn tại. Scheme quy ước các hàm làm thay đổi trạng thái một đối tượng có tên gọi kết thúc bởi dấu chấm than !. Người ta gọi đó là những *hàm đột biến* (mutator functions). Giả sử kiểu bộ đôi được tạo ra từ hai kiểu dữ liệu nào đó của Scheme là T1 và T2, ta viết quy ước doublet(T1, T2), khi đó các hàm set-car! và set-cdr! được bổ sung vào phần đặc tả hàm như sau :

functions

set-car! : doublet(T1, T2) × T1 → any

set-cdr! : doublet(T1, T2) × T2 → any

Ngữ nghĩa của các hàm này như sau :

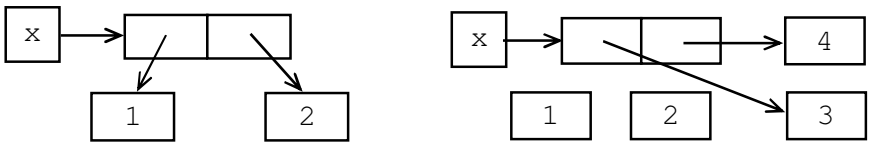
axioms var x : doublet(T1, T2) ; a, a1 : T1, b, b1 : T2

(x = cons(a, b)) ∧ (set-car!(x, a1)) ⇒ (car(x) = a1)

(x = cons(a, b)) ∧ (set-cdr!(x, b1)) ⇒ (cdr(x) = b1)

Giả sử ta có bộ đôi :

```
(define x (cons 1 2))
x
--> (1 . 2)
```



Hnh III.3.1. Biểu diễn các bộ đôi đột biến.
Trước (a) *và sau khi* (b) *thay đổi bộ đôi*

Khi đó nếu thực hiện :

```
(set-car! x 3)
(set-cdr! x 4)
```

thì giá trị của con trỏ x không thay đổi, cả hai trường hợp đều cùng trỏ đến một bộ đôi. Tuy nhiên trong trường hợp sau, nội dung của bộ đôi đã thay đổi, lúc này bộ đôi chứa hai con trỏ trỏ đến hai vị trí biểu diễn lần lượt 3 và 4. Riêng hai vị trí chứa lần lượt 1 và 2 vẫn như cũ.

Kiểm tra giá trị của con trỏ x :

```
x
--> (3 . 4)
```

III.3.4. Ứng dụng bộ đôi

1. Biểu diễn các số hữu tỷ

Scheme sử dụng bộ đôi để biểu diễn kiểu dữ liệu danh sách. Trước khi trình bày các phép xử lý trên danh sách, ta có thể sử dụng bộ đôi để minh hoạ một cách biểu diễn các số hữu tỷ, là một cặp số nguyên (trong Scheme, số hữu tỷ có dạng tiền định `n/d`).

Giả sử ta cần xây dựng một tập hợp các hàm cho phép xử lý các số hữu tỷ như : cộng (+), trừ (–), ... Giả sử kiểu số hữu tỷ `rational` được đặc tả như sau :

```
types rational
functions
   create-rat : integer × integer  -> rational
   numer :      rational           -> integer
   denom :      rational           -> integer
   =rat :       rational × rational    -> boolean
var r : rational ; n, n1, n2, d, d1, d2 : integer
preconditions
   (r=create-rat(n, d)) ⇒ (numer(r)*d=denom(r) *n) ∧
                        (denom(r) ≠ 0)
axioms
   numer(create-rat(n, d)) = n
   denom(create-rat(n, d)) = d
   (=rat(create-rat(n1, d1), create-rat(n2, d2)) ⇔ (n1*d2 =
                        n2*d1)
```

Hàm `create-rat` có vai trò tạo số hữu tỷ từ tử số n và mẫu số `d`. Các hàm `numer` trả về tử số (numerator) và `denom` trả về mẫu số (denominator) của số hữu tỷ là các hàm tiếp nhận. Hàm `=rat` là hàm kiểm tra hay chuyển kiểu, kiểm tra nếu hai số hữu tỷ là bằng nhau.

Khi *r* là một số hữu tỷ, *r=n/d*, thì tử số của *r* là *n* và mẫu số là *d≠0*. Hai số hữu tỷ *r1=n1/d1* và *r2=n2/d2* bằng nhau khi và chỉ khi *n1.d2 =n2/d1*.

Chú ý rằng phần **functions** mới chỉ định nghĩa cú pháp của hàm, chưa xây dựng ngữ nghĩa cho nó. Các tên phép toán có mặt trong ký pháp là chưa đủ, mà cần có thêm phần **axioms**. Ví dụ các định nghĩa trong Scheme sau đây tuy đúng đắn về mặt cú pháp, nhưng hoàn toàn không có nghĩa sử dụng đối với kiểu trừu tượng `rational` :

```
(define (create-rat n d) n)
(define (numer r) r)
(define (denom r) r)
(define (=rat r1 r2) (= r1 r2))
```

`(create-rat n d)`	tạo số hữu tỷ từ tử số *n* và mẫu số `d`
`(numer x)`	trả về tử số (numerator) của số hữu tỷ *x*
`(denom x)`	trả về mẫu số (denominator) của số hữu tỷ *x*
`(=rat r1 r2)`	kiểm tra nếu *r1* và *r2* đều cùng là một số hữu tỷ

Sử dụng bộ đôi, ta định nghĩa lại bốn hàm trên đây theo cách hoạt động được mô tả trong thành phần **axioms** như sau :

```
(define (create-rat n d) (cons n d))
```

```
(define (numer r) (car r))
(define (denom r) (cdr r))
(define (=rat r1 r2)
   (= (* (car r1) (cdr r2))
      (* (cdr r1) (car r2))))
```

Ví dụ : Ta áp dụng các định nghĩa trên để tạo ra các số hữu tỷ như sau :

```
(define R1 (create-rat 2 3))
(define R2 (create-rat 4 5))
R1
--> '(2 . 3)
R2
'(4 . 5)
(numer R1)
--> 2
(denom R2)
--> 5
(=rat R1 R2)
--> #f
```

Ta tiếp tục định nghĩa các hàm mới trên các số hữu tỷ `+rat`, `-rat`, `*rat`, `/rat` (tương ứng với các phép toán +, –, *, /) và hàm chuyển một số hữu tỷ thành số thực `rat->real` để bổ sung vào kiểu dữ liệu trừu tượng `rational` như sau :

functions

```
   +rat :      rational × rational     -> rational
   -rat :      rational × rational     -> rational
   *rat :      rational × rational     -> rational
   /rat :      rational × rational     -> rational
   rat->real :rational           -> real
```

Đến đây, người đọc có thể tự mình viết bổ sung phần ngữ nghĩa cho các hàm mới. Các hàm này được định nghĩa trong Scheme như sau :

```
(define (+rat x y)
   (create-rat (+ (* (numer x) (denom y)) (* (denom x) (numer y)))
      (* (denom x) (denom y))))
(define (-rat x y)
   (create-rat (- (* (numer x) (denom y)) (* (denom x) (numer y)))
      (* (denom x) (denom y))))
(define (*rat x y)
   (create-rat (* (numer x) (numer y))
      (* (denom x) (denom y))))
(define (/rat x y)
   (create-rat (* (numer x) (denom y))
      (* (denom x) (numer y))))
```

; chuyển một số hữu tỷ thành số thực :

```
(define (rat->real r) (/ (numer r) (denom r)))
```

```
(+rat R1 R2)
--> (22 . 15)
(rat->real R1)
--> 0.666666666666667
(rat->real R2)
--> 0.8
```

Ví dụ bài toán xử lý số hữu tỷ dẫn đến nhiều mức lập trình có độ trừu tượng giảm dần như sau :

– Xử lý các số hữu tỷ

– Xử lý các phép toán trên các số hữu tỷ : `+rat`, `-rat`, `*rat`, `/rat`, `=rat`

– Xử lý sơ khởi trên các số hữu tỷ : `create-rat`, `numer`, `denom`

– Xử lý sơ khởi trên các bộ đôi : `cons`, `car`, `cdr`

Khi thiết kế chương trình định hướng cấu trúc trong phương pháp tinh chế từng bước, người lập trình quan tâm đến các mức trừu tượng cao hơn, bằng cách ưu tiên chức năng xử lý như thế nào, mà chọn chậm lại (trì hoãn) cách biểu diễn các cấu trúc dữ liệu tương ứng. Đối với các số hữu tỷ, ta có thể biểu diễn chúng dưới dạng phân số rút gọn như sau :

```
(define (create-rat n d)
  (define g (gcd (abs n) (abs d)))
  ; Chú ý hàm thư viện gcd (greatest common divisor)
  ; chỉ tính ước số chung lớn nhất cho các số nguyên dương
  (cons (quotient n g) ( quotient d g)))
(define (numer r) (car r))
(define (denom r) (cdr r))
(define (=rat x y)
  (and (= (car x) (car y))
     (= (cdr x) (cdr y))))
(create-rat 6 14)
--> (3 . 7)
(gcd 12 3)
--> 3
```

Các định nghĩa hàm trên đây không làm thay đổi tính chất số học của các số hữu tỷ, cũng như không làm thay đổi mức độ trừu tượng của bài toán. Hai cách định nghĩa thủ tục `create-rat` vừa trình bày là đáng tin cậy vì trước khi gọi chúng, ta đã giả thiết rằng điều kiện đầu (`d` ≠ `0`) đã được thoả mãn.

Dự phòng và khắc phục những sai sót có thể xảy ra, trong khi lập trình hay trong quá trình khai thác của người sử dụng, bằng cách kiểm tra điều kiện đầu, được gọi là phương pháp *lập trình dự phòng* (defensive programming). Phương pháp lập trình này làm cho chương trình trở nên đáng tin cậy và dễ thuyết phục người sử dụng.

Ta viết lại thủ tục `create-rat` như sau :

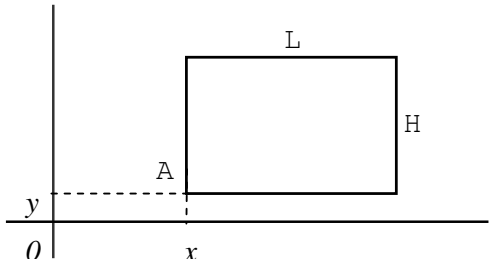
```
(define (create-rat n d)
  (if (zero? d)   ; kiểm tra điều kiện đầu
      (display "ERROR: *** Mẫu số bằng 0 !")
      (cons n d)))
(create-rat 4 0)
--> ERROR: *** Mẫu số bằng 0 !
```

2. Biểu diễn hình chữ nhật phẳng

Giả sử cần mô hình hóa các hình chữ nhật trong tọa độ phẳng *xoy* có các cạnh song song với các trục toạ độ, ta có thể có nhiều phương pháp mô tả hình chữ nhật qua tọa độ như sau :

1. Tọa độ của hai đỉnh đối diện.
2. Tọa độ tâm điểm và hai độ dài cạnh.
3. Tọa độ của đỉnh dưới cùng bên trái và hai độ dài cạnh.

Ta có thể chọn phương pháp thứ 3. Giả sử *x*, *y* là tọa độ của đỉnh *A* (xem hình vẽ), *y* là độ dài của cạnh song song với *0x* và *H* là độ dài của cạnh song song với *0y*. Ta cần định nghĩa một hàm để ánh xạ một hình chữ nhật thành một đối tượng Scheme, gọi là *biểu diễn trong* (internal representation) của hình chữ nhật.



Hình III.2. Biểu diễn các thành phần của một hình chữ nhật.

Đến đây, ta có thể nghĩ đến nhiều cách biểu diễn như sau :

- Sử dụng bộ đôi `((x y) . (L H))`,
- hoặc sử dụng bộ đôi `((x . y) . (L . H))`,
- hoặc sử dụng một danh sách gồm các thành phần `(x y L H)` (cấu trúc danh sách sẽ được trình bày trong mục sau),
- v. v...

Tạm thời ta chưa chọn cách biểu diễn hình chữ nhật (dựa theo quan điểm của phương pháp lập trình cấu trúc : cố gắng trì hoãn việc khai báo dữ liệu trong chừng mực có thể). Ta xây dựng hàm `cons-rectangle` để tạo mới một hình chữ nhật theo các thành phần *x*, *y*, *y*, *H*. Từ một cách biểu diễn trong nào đó của hình chữ nhật, ta cần xây dựng các hàm tiếp cận đến các thành phần này là `value-x`, `value-y`, `value-L`, `value-H`. Các hàm này thực hiện các phép toán trên hình chữ nhật một cách độc lập với biểu diễn trong đã cho.

Đầu tiên, ta cần xây dựng vị từ `(in? xo yo R)` kiểm tra nếu điểm *M* có toạ độ x_o, y_o có nằm bên trong (hay nằm trên các cạnh) của một hình chữ nhật *R* đã cho. Ta thấy điểm *M* phải có tọa độ lớn hơn đỉnh dưới cùng bên trái và thấp hơn đỉnh cuối cùng bên phải của *R*. Ta có chương trình như sau :

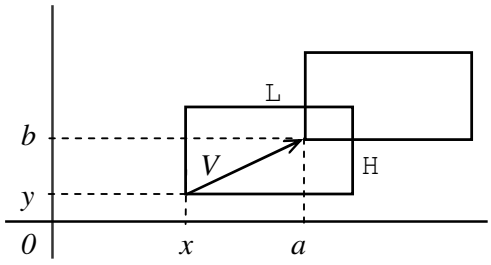
```
(define (in? xo yo R)
   (let  ((x1 (value-x R))
          ((y1 (value-y R))
          ((L (value-L R))
          ((H (value-H R)))
     (and  (<= x1 xo) (<= y1 yo)
           (<= xo (+ x1 L))
           (<= yo (+ y1 H)))))
```

Bây giờ xây dựng vị từ `(inclus? R1 r2)` để kiểm tra hình chữ nhật *R1* có nằm bên trong hình chữ nhật *R2* không ? Nếu đúng, trả về `#t`. Muốn vậy, chỉ cần *R2* chứa hai đỉnh đối diện của *R1*. Ta có chương trình như sau :

```
(define (inclus? R1 R2)
   (let  ((x1 (value-x R1))
          ((y1 (value-y R1))
          ((L1 (value-L R1))
          ((H1 (value-H R1)))
      (and  (in? x1 y1 R2)
            (in? (+ x1 L2) (+ y1 H1) R2))))
```

Để tịnh tiến (translate) một hình chữ nhật *R* theo một véctơ *V* có các thành phần *a*, *b*, chỉ cần tịnh tiến đỉnh dưới cùng bên trái của nó. Ta có hàm tịnh tiến như sau :

```
(define (translate R a b)
   (let  ((x (value-x R))
          ((y (value-y R))
          ((L (value-L R))
          ((H (value-H R)))
      (cons-rectangle (+ x a) (+ y b) L H)))
```



Hình III.3. Tịnh tiến một hình chữ nhật.

Như đã trình bày, việc xử lý trên các hình chữ nhật không phụ thuộc vào cách biểu diễn dữ liệu. Khi cần thay đổi cách biểu diễn, chỉ cần viết lại các hàm tạo mới hình chữ nhật và các hàm tiếp cận đến các thành phần, mà không cần thay đổi các hàm `in?`, `inclus?`, và `translate`.

Chẳng hạn ta có thể biểu diễn hình chữ nhật bởi bộ đôi :

```
((x . y) . (L . H))
```

Khi đó, cách biểu diễn trong của hình chữ nhật được xây dựng như sau :

```
(define (cons-rectangle x y L H)
     (cons (cons x y) (cons L H)))
(define (value-x R) (car (car R)))
(define (value-y R) (cdr (car R)))
(define (value-L R) (car (cdr R)))
(define (value-H R) (cdr (cdr R)))
```

Ví dụ :

```
(define R (cons-rectangle 1 2 3 4))
R
--> '((1 . 2) 3 . 4) ; tương đương với '((1 . 2) . (3 . 4))
```

```
(value-L R)
--> 3
(value-H R)
--> 4
```

III.4 Kiểu dữ liệu danh sách

III.4.1. Khái niệm danh sách

Khi giải một số bài toán, người ta thường phải nhóm nhiều dữ liệu thành một dữ liệu tổ hợp duy nhất. Để nhóm các ngày trong tuần là *Monday, Tuesday, Wednesday, Thursday, Friday, Saturday* và *Sunday* thành một tổ hợp dữ liệu, người ta sử dụng cấu trúc danh sách có dạng một s-biểu thức như sau :

```
(mon tue wed thu fri sat sun)
```

Tuy nhiên, nếu định nghĩa :

```
(define week-list)
     (mon tue wed thu fri sat sun))
-->  *** ERROR-unbound variable: mon
```

thì ta gặp lại sự sai sót về các ký tự. Scheme xem danh sách này như là một lời gọi hàm có tên `mon` với các tham đối là `tue`, ... `sun` và, như đã thấy, ta chưa hề định nghĩa hàm `mon` (!). Để định nghĩa danh sách trên, Scheme sử dụng phép toán trích dẫn :

```
(define week-list
    '(mon tue wed thu fri sat sun))
```

Khi đó ta có `week-list` là biến có giá trị là một danh sách các ký tự :

```
week-list
--> (mon tue wed thu fri sat sun))
```

Để xử lý thuận tiện, người ta sử dụng biến *L* để chỉ định một danh sách, ta có :

```
(define L '(1 2 3 4 5))
L
--> '(1 2 3 4 5)
```

Có thể sử dụng các danh sách phức tạp hơn để biểu diễn các biểu thức. Có thể xây dựng các biểu thức số học, biểu thức chứa các dữ liệu ký hiệu, hay chưa các danh sách lồng nhau. Một biểu thức khi được trích dẫn được gọi là một *biểu thức trực kiện* (literal expression) vì rằng giá trị của nó chính là văn bản (text) tạo ra nó. Biểu thức trực kiện có vai trò là một hằng :

Cho trước một danh sách, khi ta nói độ dài hay số phần tử của danh sách (length) là số phần tử của danh sách đó. Sau đây là một số ví dụ :

```
'(1 2 2 3)          ; là một danh sách gồm 4 số nguyên
--> '(1 2 2 3)
'(one + two = three)   ; là một danh sách gồm 5 ký hiệu tuỳ ý
--> (one + two = three)
```

```
'()                ; danh sách rỗng (empty list) không có phần tử nào
--> '()
'(* 4 5)           ; không phải là một biểu thức tính được
--> (* 4 5)
```

Nếu ta cần biểu diễn các thông tin về một trang lịch công tác trong ngày, ta có thể sử dụng một danh sách dạng :

```
(define L-congviec-10-10-2000
    '(thu-hai
        (sang
           (9h doc-thu)
           (9h-30 chuan-bi-len-lop)
           (10h len-lop)
           (12h an-trua))
        (chieu
           (14h tham-gia-chuyen-de)
           (16h den-cuoc-hen))))
```

Kiểu dữ liệu danh sách cần có thủ thuật để có thể đọc vào và in ra. Scheme cho phép xây dựng các danh sách *không thuần nhất* (heterogeneous list) về kiểu. Chẳng hạn :

```
(define L '(1 (2 3) x "bonjour"))
(list-ref L 0)
--> 1
(list-ref L 3)
--> "bonjour"
```

Trừ trường hợp ngược lại, ở đây ta chỉ nên xử lý các danh sách *thuần nhất* (homogeneous list) là danh sách chỉ chứa cùng một kiểu dữ liệu T đã cho : chứa toàn các số, hoặc chỉ chứa toàn các ký hiệu, v.v... Thứ tự của các phần tử xuất hiện trong một danh sách là có ý nghĩa.

Một trong những đặc trưng của các chương trình họ Lisp là có cùng bản chất với dữ liệu. Chẳng hạn, định nghĩa hàm `(define...)` cũng là một danh sách. Người ta gọi một *danh sách con* (sub-list) là một phần tử của danh sách đã cho. Phần tử này có thể là một danh sách, hay là một danh sách con của một danh sách con.

Ví dụ, danh sách :

```
'(a (b c) () ((c)) d)
```

có các danh sách con là : `(b c)`, `()`, `((c))` và `(c)`. Một danh sách không có danh sách con được gọi là một *danh sách phẳng* (plate list).

Người ta nói rằng một phần tử là ở *mức một* (first level) của một danh sách nếu phần tử này không ở trong một danh sách con.

Các phần tử ở mức một của danh sách vừa nêu trên đây là :

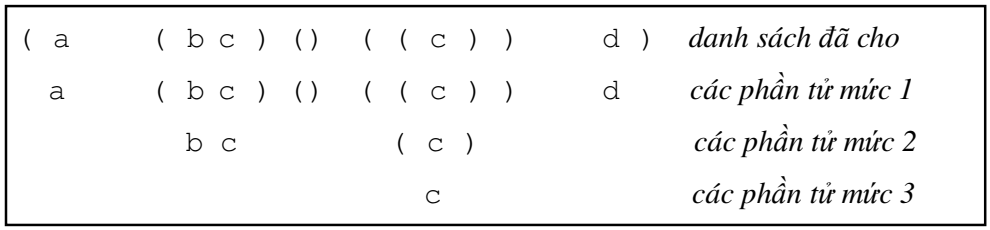
```
a, (b c), (), ((c)), c
```

Người ta gọi các phần tử ở *mức thứ n* là những phần tử ở mức một của một danh sách con được đặt ở *mức n-1*.

Độ sâu (depth) của một danh sách là mức của một phần tử ở mức cao nhất. Chẳng hạn danh sách trên đây có độ sâu là 3.

Độ dài (length) của một danh sách là số các phần tử ở mức thứ nhất, được tính nhờ hàm tiền định `length` của Scheme :

```
(length '((a b) (c d) () (e)))
--> 4
```



Hình III.4. Biểu diễn các mức của một danh sách phức hợp.

Định nghĩa kiểu dữ liệu trừu tượng danh sách

Có thể có nhiều cách khác nhau để định nghĩa một danh sách. Sau đây là đặc tả kiểu dữ liệu trừu tượng danh sách `list(T)` sử dụng các phép toán cơ bản `cons`, `car` và `cdr`, trong đó, `T` là kiểu phần tử của danh sách :

```
types list(T)
functions
     () :  list(T)
     null? :        list(T) -> boolean
     pair? :        list(T) -> boolean
     cons :T × list(T) -> list(T)
     car : list(T)  -> T
     cdr : list(T)  -> list(T)
preconditions
     car(x) chỉ xác định với x ≠ 0
     cdr(x) chỉ xác định với x ≠ 0
axioms var L : list(T), x : T
     car(cons (x,L))   = x
     cdr(cons (x,L))   = L
     null?(())     = true
     null?(cons (x,L)) = false
     pair?(())     = false
     pair?(cons (x,L)) = true
```

III.4.2. Ứng dụng danh sách

III.4.2.1. Các phép toán cơ bản cons, list, car và cdr

Scheme sử dụng bộ xây `cons` để tạo một danh sách :

(*cons s L*)

trả về một danh mới bằng cách thêm giá trị của `s` trước danh sách `L`.

```
(cons 'a '())
--> (a)
(cons (+ 5 8) '(a b))
--> (13 a b)
```

Scheme có thể nhận biết một danh sách và đưa ra theo cách riêng :

```
(cons 1 2)
--> (1 . 2)
(cons 1 '())        ; chú ý dấu quote do danh sách rỗng làm tham đối
--> (1)             ; không phải (1 . ())
(cons 1 (cons 2 (cons 3 ())))
-> (1 2 3)          ; không phải (1 . (2 . (3 . ())))
```

Để xây dựng danh sách, Scheme thường sử dụng hàm `list`. Quan hệ giữa bộ xây `cons` và cấu trúc `list` được tóm tắt bởi đẳng thức sau :

(*list* <e_1> ... <e_N>) =
(*cons* <e_1> (*cons* <e_2> ... (*cons* <e_N> '())...))

Ví dụ :

```
(list (cons 1 2) (cons 3 4))
--> ((1 . 2) (3 . 4))  ; (1 . 2) và (3 . 4) là các bộ đôi
```

Hàm `list` nhận một số tham đối tùy ý là các s-biểu thức để trả về một danh sách mới tổ hợp từ các giá trị của các tham đối đó. Ví dụ :

```
(list 1 2 3)
--> (1 2 3)
(list '(a b) '(c d) '(c d) '() '((e)))
--> ((ab) (cd) () (e))
```

Danh sách sau đây gồm 3 phần tử, phần tử thứ nhất là dấu + :

```
'(+ 1 2)
--> (+ 1 2)
```

Chú ý kiểu dữ liệu của các phần tử danh sách :

```
(list a b c)
--> Error: undefined variable a
(list 'a 'b 'c)
' (a b c)
```

Các dạng `car` và `cdr` dùng để tiếp cận đến các phần tử của một danh sách khác rỗng. Hàm `(car L)` trả về phần tử đầu tiên của `L`

```
(car '(a b c))
--> a
(car '(1 2 3)) == (car (quote 1 2 3))
--> 1
(car '(+ 1 2))
--> +
(car ''1)
--> quote
```

Hàm `(cdr L)` trả về danh sách `L` đã lấy đi phần tử đầu tiên

```
(cdr '(a b c))
--> (b c)
```

Ta cũng có các đẳng thức sau :

```
(car (cons s L)) = s
(cdr (cons s L)) = L
```

Với quy ước dấu gạch trên tên chỉ định giá trị của một biểu thức. Trong Scheme, `car` hay `cdr` của một danh sách rỗng đều dẫn đến một lỗi sai, trong khi trong các ngôn ngữ phát triển từ Lisp thì thường có kết quả trả về là một danh sách rỗng. Nếu muốn thay đổi tên các hàm `car` và `cdr` này, chỉ cần định nghĩa các hàm đồng nghĩa `head` và `tail` như sau :

```
(define head car)
(define tail cdr)
```

Tuy nhiên, không nên định nghĩa như vậy, vì rằng sẽ gặp khó khăn khi công việc lập trình có sự tham gia của nhiều người. Bằng cách tổ hợp các hàm `car` hoặc `cdr`, ta có thể tiếp cận đến bất kỳ một phần tử nào của một danh sách :

```
(car (cdr week-list))
--> mardi
(car (cdr (cdr week-list))))
--> mercredi
```

Cách viết tổ hợp trên đây sẽ làm người đọc khó theo dõi. Chính vì vậy người ta đưa ra một hệ thống viết tắt để chỉ định sự tổ hợp tối đa là bốn `car` hay `cdr`. Scheme quy ước :

$cx_1x_2x_3x_4r$ với x_j là một **a** hoặc một **d**, $j=1..4$

là dãy «hàm» ghép : $cx_1r \circ cx_2r \circ cx_3r \circ cx_4r$

Scheme có tất cả 28 hàm ghép như vậy. Ví dụ :.

```
(caadr '(a (b) c d e)) ; là hàm car∘car∘cdr
--> b
(cddddr '(a (b) c d e)) ; là hàm cdr∘cdr∘cdr∘cdr
--> (e)
(cddddr week-list)
--> (vendredi samedi dimanche)
```

Sau đây, ta có thể sử dụng hai sơ đồ tổng quát để định nghĩa các hàm xử lý danh sách :

Sơ đồ đệ quy tuyến tính :

```
(define (<fname> <arg>)
   (if (null? <arg>)
       <init-val>
       (<func> (car <arg>) (<fname> (cdr <arg>)))))
```

Sơ đồ lặp :

```
(define (<fname> <arg>)
   (define (<i-name> <arg> <result>)
      (if (null? <arg>)
          <result>
          (<i-name> (cdr <arg>)
                    (<func> (car <arg>) <result>))))
   (<i-name> <arg> <init-val>))
```

Vị từ `null?` của Scheme cho phép kiểm tra một danh sách rỗng.

```
(null? '())
--> #t
```

III.4.2.2. Các hàm xử lý danh sách

Thư viện của Scheme có sẵn nhiều hàm xử lý danh sách. Sau đây ta đặc tả bổ sung một số hàm xử lý danh sách thông dụng vào kiểu dữ liệu trừu tượng `list(T)` như sau :

```
functions
     length :  list (T)          -> Integer
     append :  ist(T) × list(T)  -> list(T)
     reverse : list(T)           -> list(T)
```

1. Các hàm length, append và reverse

Hàm `length` trả về độ dài của danh sách được định nghĩa như sau :

```
(define (length L)
   (if (null? L)
      0
      (+ 1 (length (cdr L)))))
```

Hàm `length` có chi phí tuyến tính. Ví dụ :

```
(length (list 1 2 3))
--> 3
(= (length (list 1 2 3 4)) 0)
--> #f
```

Các danh sách rỗng có thể được tạo ra bởi hàm `(list)` không có tham đối, hoặc sử dụng trực kiện `'()`. Ví dụ :

```
(= (length (list)) 0)
--> #t
(= (length '()) 0)
--> #t
(null? (list 1 2 3 4))
--> #f
(null? (list))
--> #t
(null? '())
--> #t
```

Scheme không thừa nhận dạng `'()` là một biểu thức, mà xem `'()` là một giá trị. Vì vậy, để sử dụng danh sách rỗng, ta cần trích dẫn :

```
'()
--> '()
```

Hàm `append` nhận một số tham đối tùy ý, mỗi tham đối có giá trị là một danh sách để trả về một danh sách mới, là *ghép* (concatenation) tất cả các danh sách tham đối này. Hàm `append` được định nghĩa như sau :

```
(define (append L1 L2)
   (if (null ? L1)
        L2
        (cons (car L1) (append (cdr L1) L2))))
```

Hàm `append` có chi phí tuyến tính theo `L1`. Ví dụ :

```
(append (list 1 2 3) (list 4 5 6 7))
--> (1 2 3 4 5 6 7)
(append '(a b) '(c d) '() '((e)))
--> (a b c e (e))
```

Không nên nhầm lẫn cách sử dụng của các hàm `cons`, `list`, và `append` :

```
(cons '(a b) '(c d))
--> ((a b) (c d))
(list '(a b) '(c d))
--> ((a b) (c d))
(append '(a b) '(c d))
--> (a b c d)
```

Hàm `reverse` trả về danh sách nghịch đảo (reverse list) của tham đối, được định nghĩa đệ quy như sau :

```
(define (reverse L)
   (if (null? L)
        '()  ; hoặc (list)
        (append (reverse (cdr L)) (list (car L)))))
```

Chi phí của hàm `reverse` là tuyến tính theo `L`. Ví dụ :

```
(reverse '(dog cat cock duck pig))
'(pig duck cock cat dog)
(reverse (list 1 2 3))
--> (3 2 1)
(reverse (append '(a b) '(c d)))
--> '(d c b a)
```

Chú ý người lập trình thường gặp lỗi sai về kiểu giữa `car`, `cdr` và `list`. Nên thường xuyên thực hiện việc phân tích kiểu. Chẳng hạn nếu dòng cuối trong định nghĩa hàm `reverse` viết lại là :

```
(append (reverse (cdr L)) (car L)))) ; quên đặt tiền tố list
```

thì sẽ gây ra lỗi sai vì `(car L)` là một phần tử, không phải là một danh sách. Ta có thể định nghĩa lại hàm `reverse` nhờ phép lặp như sau :

```
(define (reverse L)
   (define (rev-iter L R)
      (if (null? L)
           R
           (rev-iter (cdr L) (cons (car L) R))))
   (rev-iter L '())) ; hoặc (list)
(reverse (list 1 2 3 4))
--> (4 3 2 1)
```

Chi phí của hàm `reverse` vẫn tuỳ thuộc tuyến tính vào *L*.

2. Các hàm tham chiếu danh sách

(list-ref L n)

(list-tail L n)

(list? L)

Hàm `(list-ref L n)` trả về phần tử thứ *n* của danh sách *L*, với quy ước rằng trong một danh sách, các phần tử được đánh số từ 0.

```
(define (list-ref L n)
   (if (zero? n)
            (car L)
            (list-ref (cdr L) (- n 1))))
(list-ref '(a b c d e) 3)
--> d
```

Hàm `(list-tail L n)` trả về phần tử còn lại của danh sách *L* sau khi bỏ đi *n* phần tử đầu tiên :

```
(define (list-tail L n)
   (if (zero? n)
       L
       (list-tail (cdr L) (- n 1))))
(list-tail '(a b c) 0)
--> '(a b c)
(list-tail '(a b c) 2)
--> '(c)
(list-tail '(a b c) 3)
--> '()
(list-tail '(a b c . d) 2)
--> '(c . d)
(list-tail '(a b c . d) 3)
--> d
(define week-end (list-tail week-list 5))
week-end
--> (samedi dimanche)
```

Vị từ `(list? L)` kiểm tra *L* có phải là một danh sách không :

```
(list? '())
--> #t
(list? '(a b c))
--> #t
(list? 'a)
--> #f
(list? '(3 . 4))
--> #f
(list? 3)
--> #f
```

3. Các hàm chuyển đổi kiểu

(string->list str)

(list->string L)

(string->number str [radix])

Hàm `(string->list str)` chuyển một chuỗi thành một danh sách. Hàm này có thể được định nghĩa như sau :

```
(define (tring->list str)
    (do ((i (- (string-length str) 1) (- i 1))
         (ls '() (cons (string-ref str i) ls)))
       ((< i 0) ls)))

(string->list "")
--> '()
(string->list "abc")
--> '(#\a #\b #\c)
(apply char<? (string->list "abc"))
--> #t
(map char-upcase (string->list "abc"))
--> '(#\A #\B #\C)
(map char-downcase (string->list "ABC"))
--> ' (#\a #\b #\c)
```

Chú ý hàm `(char-upcase ch)` chuyển một chữ thường thành chữ hoa, còn hàm `(char-downcase ch)` chuyển một chữ hoa thành chữ thường.

Hàm `(list->string L)` chuyển một danh sách thành một chuỗi. Hàm này có thể được định nghĩa như sau :

```
(define (list->string ls)
    (let ((s (make-string (length ls))))
       (do ((ls ls (cdr ls)) (i 0 (+ i 1)))
           ((null? ls) s)
         (string-set! s i (car ls)))))

(list->string '())
--> ""
(list->string '(#\a #\b #\c))
--> "abc"
(list->string
    (map char-upcase
         (string->list "abc")))
--> "ABC"
```

Hàm `(string->number str [radix])` chuyển một chuỗi `str` thành một số hệ 10 (mặc nhiên). Nếu có thêm thành phần *radix* là một số nguyên chỉ cơ số hệ đếm 2, 8, 10 hay 16 thì số trong hệ đếm tương ứng sẽ được đổi sang hệ 10 :

```
(string->number "9999")
--> 100
(string->number "1e3")
--> 1000.0
(string->number "1ABC" 16)
--> 6844
```

4. Các hàm so sánh danh sách

Để so sánh các danh sách, người ta dùng vị từ `equal?` :

```
(equal? '(a b 3) (list 'a 'b (+ 1 2)))
--> #t
(equal? '(()) '())
--> #t
```

Ta đã gặp có các vị từ xác định kiểu dữ liệu danh sách của một s-biểu thức :

`(list? s)--> #t` nếu giá trị của biểu thức `s` là một danh sách bất kỳ
`(null? s)--> #t` nếu giá trị của biểu thức `s` là một danh sách rỗng
`(pair? s)--> #t` nếu giá trị của biểu thức `s` là một danh sách khác rỗng.

Đôi khi người ta cần kiểm tra nếu một s-biểu thức không phải là một danh sách và kiểm tra nếu một danh sách có phải được rút gọn thành một phần tử duy nhất không. Ta thêm các vị từ sau đây vào thư viện các tiện ích vốn chưa có trong thư viện tiền định của Scheme :

Vị từ `atom?` kiểm tra nếu một s-biểu thức không phải là một danh sách :

```
(define (atom? s)
    (not (pair? s)))
```

Vị từ `singleton?` kiểm tra một danh sách được rút gọn thành một phần tử duy nhất :

```
(define (singleton L)
    (null? (cdr L)))
```

Người ta cũng thường phải kiểm tra nếu một đối tượng là một phần tử ở mức thứ nhất của một danh sách. Muốn vậy, người ta sử dụng một hàm tiền định `(member S L)`. Hàm này trả lại một phần của danh sách `L` chứa phần đầu tiên (first ocurrence) của giá trị `s`, hoặc trả về `#f` nếu danh sách `L` không có. Sự so sánh được thực hiện nhờ kiểm tra `equal?` :

```
(member '(b) '(a b c d))
--> #f
(member '(b) '(a b c (b) d))
--> ((b) d)
```

Scheme xem các giá trị bằng nhau ở các mức độ chính xác khác nhau. Vị từ `equal?` so sánh các cấu trúc nhưng không phải là sự so sánh tên của các đối tượng. Tên của các đối tượng được kiểm tra nhờ vị từ `eq?` và tồn tại một vị từ so sánh ít chặt chẽ hơn là `eqv?`.

Như vậy, để so sánh các danh sách hoặc các chuỗi, ta sử dung `equal?`, để so sánh các số, sử dụng = và để so sánh các ký hiệu, sử dụng `eq?`. Scheme có ba hàm so sánh hoạt động như sau :

`member` sử dụng phép kiểm tra `equal?`
`memv` sử dụng phép kiểm tra `eqv?`
`memq` sử dụng phép kiểm tra `eq?`

Sau đây là định nghĩa của hàm `member` :

```
(define (memq s L)
    (cond
      ((null? L) #f)
      ((eq? (car L) s) L)
      (else (memq s (cdr L)))))
```

Ví dụ :

```
(memq '(b) '(a b c (b) d))
--> #f
(member "tue" '("mon" "tue" "wed"))
--> ("tue" "wed")
(memq 'a '(b c a d e))
--> (a d e)
(memq 'a '(b c d e g))
--> #f
(memq 'a '(b a c a d a))
--> (a c a d a)

(memv 3.4 '(1.2 2.3 3.4 4.5))
--> (3.4 4.5)
(memv 3.4 '(1.3 2.5 3.7 4.9))
--> #f

(member '(b) '((a) (b) (c)))
--> ((b) (c))
(member '(d) '((a) (b) (c)))
--> #f
(member "b" '("a" "b" "c"))
--> ("b" "c")
```

III.4.2.3. Dạng case xử lý danh sách

Dạng `case` tương tự `cond` nhưng thường được dùng để so sánh trên các danh sách. Cú pháp của dạng `case` như sau :

```
(case s
   (L1 body1)
   ...
   [ (else bodyN) ] )
```

Khi gặp `case`, s-biểu thức `s` được tính và được hàm `memv` kiểm tra nếu giá trị cúa `s` thuộc danh sách `L1`. Nếu đúng, `case` trả về giá trị của `body1`, nếu không, tiếp tục kiểm tra nếu giá trị cúa `s` thuộc về danh sách `L2`, v.v...

Tương tự `cond`, các có thể là những dãy s-biểu thức và `elle` có thể vắng mặt. Ví dụ :

```
(case (* 2 3)
    ((2 3 5 7 9) 'prime)
    ((1 4 6 8 9 11) 'composite))
--> 'composite
(case (car '(c d))
    ((a e i o u) 'vowel)
    ((w y) 'semivowel)
    (else 'consonant))
--> 'consonant
```

```
(case (car '(c d))
    ((a) 'a)
    ((b) 'b))
--> không trả về kết quả gì (unspecified)
```

Ví dụ minh hoạ sau đây định nghĩa một hàm dịch các đại từ nhân xưng từ tiếng Pháp qua tiếng Anh :

```
(define (pronoun-traduit pronounF)
  (case pronounF
    ((je) 'I)
    ((tu vous) 'you)
    ((il) 'he)
    ((elle) 'she)
    ((nous) 'we)
    ((ils) 'they)
    (else 'pronoun-inconnu)))
(pronoun-traduit 'vous)
--> you
```

Cấu trúc danh sách của các ngôn ngữ họ Lisp rất tổng quát. Do vậy nó thường được sử dụng để mô hình hoá tất cả các kiểu dữ liệu thuộc loại ký hiệu hoặc không.

Giả sử cần tính nghiệm phương trình bậc hai $x^2 + 2bx + c = 0$ với hệ số thực. Vấn đề là cần tính biệt thức $delta = b^2 - c$. Ta quy ước sử dụng danh sách để biểu diễn các lời giải khác nhau căn cứ vào giá trị *delta :*

delta > 0, hai nghiệm thực phân biệt : `(real x1 x2)`

delta = 0, nghiệm kép :`(double x)`

delta < 0, nghiệm phức : `(complex realpart imaginarypart)`

Hàm tính nghiệm phương trình bậc hai như sau :

```
(define (trinomialroot b c)
  (let ((delta (- (* b b) c)))
    (cond ((< 0 delta)
      (let ((delta2root (sqrt delta)))
        (list 'real
              (+ (- b) delta2root)
              (- (- b) delta2root))))
      ((zero? delta) (list 'double (- b)))
      (else (let ((delta2root (sqrt (- delta))))
          (list 'complex (- b) delta2root))))))
```

Để không lẫn lộn các tính toán với các kết quả in ra, ta viết thủ tục `solution-display` dưới đây sử dụng `case` :

```
(define (solution-display solution)
  (case (car solution)
    ((real) (display "2 real roots: ")
            (display (cadr solution))
            (display " and ")
            (display (caddr solution)))
    ((double) (display "1 racine reelle double: ")
            (display (cadr solution)))
```

```
      (else (display " 2 racines complex conjuguees: ")
            (display (cadr solution))
            (display " +/- i ")
            (display (caddr solution))))
      (newline))   ; Qua dòng mới
(solution-display (trinomialroot 2 2))
--> 2 real roots: -0.585786 and -3.41421
(solution-display (trinomialroot -1 5))
--> 2 racines complex conjuguees: 1 +/- i 2
(solution-display (trinomialroot 1 1))
--> 1 racine reelle double: -1
```

III.4.2.4. Kỹ thuật đệ quy xử lý danh sách phẳng

Khi viết một chương trình nhận danh sách làm tham biến, người ta có thể sử dụng phép đệ quy bằng cách *chuyển phép tính hàm về tham biến* `cdr` *của danh sách*. Sau đây, ta xét một số ví dụ đơn giản đối với các danh sách phẳng (chỉ có các phần tử ở mức một).

1. Tính tổng các phần tử của một danh sách

Cho *Lnb* là một danh sách các số. Cần xây dựng hàm `ListSum` tính được tổng các phần tử của *Lnb* sao cho khi danh sách rỗng, giá trị trả về là 0 ?

Nếu $Lnb=(n_1\ n_2\ \ldots\ n_j)$, thì $n_1+n_2+\ldots+n_j = n_1 + (n_2+\ldots+n_j)$. Từ đó :

```
(ListSum n1 n2 … nj) = n1 + (ListSum '(n2 + … + nj))
```

```
(define (ListSum Lnb)
  (if (null? Lnb)
    0
    (+ (car Lnb) (ListSum (cdr Lnb)))))
(ListSum '(1 2 3 4 5))
--> 15
```

2. Danh sách các số nguyên từ 0 đến n

Thêm số *n* vào cuối danh sách các số nguyên :

(0 ... *n*-1)

ta nhận được danh sách :

(0 ... *n*-1 *n*)

Để thêm phần tử *s* vào cuối danh sách *L*, ta xây dựng hàm `append1` sau đây :

```
(define (append1 s L)
  (append L (list s)))
```

Gọi `iota` là hàm trả về danh sách các số nguyên từ `0` đến n :

```
(define (iota n)
  (if (zero? n)
    '(0)
    (append1 (iota (- n 1)) n)))
```

```
(iota 10)
--> '(0 1 2 3 4 5 6 7 8 9 10)
```

3. Nghịch đảo một danh sách

Do thư viện Scheme đã có hàm nghịch đảo một danh sách của là `reverse` :

```
(reverse (iota 9))
--> '(9 8 7 6 5 4 3 2 1 0)
```

nên ta có thể định nghĩa một hàm khác để nghịch đảo một danh sách. Nguyên lý hoạt động tương tự hàm `iota` : nghịch đảo phần còn lại của danh sách và thêm vào cuối danh sách phần tử đầu tiên. Hàm `myreverse` được định nghĩa như sau :

```
(define (myreverse L)
   (if (null? L)
      '()
      (append1 (myreverse (cdr L)) (car L))))

(myreverse '(a (b c d) e)
--> (e (b c d) a)
```

4. Hàm append có hai tham đối

Giả sử ta không sử dụng hàm `append` tiền định mà tự xây dựng một hàm khác nhờ hàm `cons`. Hàm `append2` ghép hai danh sách L_1 và L_2 để trả về một danh sách như sau :

```
(define (append2 L1 L2)
   (if (null? L1)
      L2 ; Nếu danh sách L1 là rỗng, thì kết quả trả về là L2
       ; Nếu L1 ≠ (), đặt car vào đầu phép ghép phần còn lại của L1 với L2.
      (cons (car L1) (append2 (cdr L1) L2))))

(append2 '(a b c) '((c d)))
--> (a b c (c d))
```

Lời gọi đệ quy chỉ có tác dụng đối với danh sách L_1 (làm độ dài giảm dần) nhưng đảm bảo tính dừng. Hàm `append2` không gọi đến hàm `append`. Về nguyên tắc, người lập trình có thể thay đổi các hàm tiền định trong thư viện, nhưng gặp nguy cơ không quản lý hết các sai sót xảy ra.

5. Loại bỏ các phần tử khỏi danh sách

Cho s-biểu thức *s* và danh sách *L*, cần xây dựng hàm `remove` sao cho loại bỏ hết khỏi *L* mọi phần tử có giá trị *s* ở mức một. Xảy ra ba trường hợp như sau :

- Nếu danh sách rỗng, thì kết quả trả về cũng rỗng.
- Nếu danh sách bắt đầu bởi một phần tử bằng *s*, thì chỉ cần loại bỏ *s* khỏi phần còn lại của danh sách.
- Nếu danh sách không bắt đầu bởi một phần tử bằng `s`, thì bắt đầu bởi việc loại bỏ *s* khỏi phần còn lại của danh sách.

Sau đây là hàm `remove` :

```
(define (remove s L)
   (cond ((null? L) '())
         ((equal? s (car L)) (remove s (cdr L)))
         (else (cons (car L) (remove s (cdr L))))))
```

```
(remove 'a '(a b c a d))
--> (b c d)
```

6. Bài toán tính tổng con

Cho một số nguyên *N* và một danh sách *Lnb* gồm các số nguyên dương tăng dần. Hãy tìm một tập hợp các phần tử của *Lnb* có tổng bằng *N*.

Ví dụ, cho *N* = 21, từ danh sách `(1 1 3 5 7 10 12 15)`, ta có thể chọn các phần tử `(1 1 7 12)`. Nếu danh sách rỗng thì ta quy ước tổng trả về bằng 0. Còn nếu danh sách khác rỗng thì giả sử phần tử đầu tiên của danh sách là x_0 (là phần tử bé nhất), ta sẽ xét các trường hợp sau đây :

- Nếu x_0 lớn hơn *N* thì sẽ không có lời giải và kết quả sẽ là `#f`.
- Nếu $x_0 = N$ thì kết quả trả về sẽ là danh sách chỉ có mỗi phần tử x_0.
- Nếu x_0 nhỏ hơn *N* thì ta kiểm tra x_0 có thuộc vào lời giải hay không. Muốn vậy, ta giải lại bài toán này cho các phần tử khác của danh sách và với tổng bây giờ là $N - x_0$:
  (a) Nếu nhận được một lời giải bộ phận, chỉ cần thêm vào x_0.
  (b) Ngược lại không có lời giải và tiếp tục đối với các phần tử khác x_0.

Sau đây là chương trình :

```
(define (goodcount N Lnb)
  (cond ((null? Lnb) (if (zero? N) '() #f))
        ((< N (car Lnb)) #f)
        ((= N (car Lnb)) (list N))
        (else (let ((partresult
                     (goodcount (- N (car Lnb)) (cdr Lnb))))
                (if partresult
                   (cons (car Lnb) partresult)
                   (goodcount N (cdr Lnb))))))))
(goodcount 21 '(1 1 3 5 7 10 12 15))
--> (1 1 7 12)
```

7. Lập danh sách các số nguyên tố

Cho một số nguyên *n*, cần tìm các số nguyên tố trong khoảng 2..*n*. Có nhiều thuật toán tìm số nguyên tố, sau đây là thuật toán cổ sử dụng sàng (sieve) Eratosthènes viết bằng ngôn ngữ mệnh lệnh :

1. Khởi động tập hợp kết quả `S` chứa các số nguyên tố.
2. Xây dựng sàng `sieve` chứa danh sách các số 2..*n*.
3. Repeat
   Tìm số nguyên tố `prime`.
   Thêm `prime` vào tập hợp kết quả `S`.
   Loại trừ khỏi `sieve` các bội số của `prime` là `2*prime`, `3*prime`...
   until `sieve` = rỗng

Sau đây là chương trình Scheme :

```
(define  (interval-list m n)
  (if (> m n)
      '()
      (cons m (interval-list (+ 1 m) n))))
```

```
(define (sieve L)
   (define (remove-multiples n L)
       (if (null? L)
          '()
          (if  (= (modulo (car L) n) 0)      ; division test
            (remove-multiples n (cdr L))
            (cons (car L)
              (remove-multiples n (cdr L))))))
   (if (null? L)
       '()
       (cons (car L)
       (sieve (remove-multiples (car L) (cdr L))))))
(define (primes<= n)
   (sieve (interval-list 2 n)))
(primes<= 500)
--> '(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
     71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
     149 151 157 163 167 173 179 181 191 193 197 199 211 223
     227 229 233 239 241 251 257 263 269 271 277 281 283 293
     307 311 313 317 331 337 347 349 353 359 367 373 379 383
     389 397 401 409 419 421 431 433 439 443 449 457 461 463
     467 479 487 491 499)
```

III.4.2.5. Kỹ thuật đệ quy xử lý danh sách bất kỳ

Khi chương trình nhận một danh sách tổng quát làm tham biến, ta có thể sử dụng kỹ thuật đệ quy để chuyển phép tính của hàm đang định nghĩa đối với tham biến `cdr` và/hoặc `car` của danh sách. Ta tiếp tục xét một số ví dụ đơn giản sau đây.

1. Làm phẳng một danh sách

Ta cần xây dựng một danh sách mới từ danh sách đã cho mà không còn các cặp ngoặc bên trong như sau :

```
(aflatten '(a (b c) () ((d)) e))
--> (a b c d e)
```

Ta có :

```
(define (aflatten L)
   (cond ((null? L) '())
      ; Nếu car của danh sách đã cho là một danh sách, thì làm phẳng nó
      ; rồi ghép kết quả với kết quả làm phẳng của cdr.
      ((list? (car L)) (append (aflatten (car L))
                               (aflatten (cdr L))))
      ; Nếu car của danh sách đã cho không phải là một danh sách,
      ; thì thêm phần tử này vào đầu kết quả làm phẳng của cdr.
      (else (cons (car L) (aflatten (cdr L))))))
```

2. Tính tổng các số có mặt trong danh sách

Xảy ra bốn trường hợp sau :

- Nếu danh sách đã cho là rỗng, thì kết quả cũng là danh sách rỗng
- Nếu `car` là một số, thì cộng số này với tổng của tất cả các số của `cdr`.
- Nếu `car` là một danh sách khác rỗng, thì tính tổng của tất cả các số của `car` rồi cộng kết quả này với tổng của tất cả các số của `cdr`.
- Nếu `car` là một danh sách rỗng hoặc là một đối tượng khác số thì chỉ tính tổng của tất cả các số của `cdr`.

Sau đây là chương trình :

```
(define (Listsum* L)
   (cond ((null? L) 0)
         ((number? (car L)) (+ (car L) (Listsum* (cdr L))))
         ((pair? (car L)) (+ (Listsum* (car L))
                             (Listsum* (cdr L))))
         (else (Listsum* (cdr L)))))
(Listsum* '(a (4) 5 ((6 b)) 8))
--> 23
```

3. Loại bỏ khỏi danh sách một phần tử ở các mức khác nhau

Cho một danh sách *L* và một biểu thức *s*. Cần loại bỏ khỏi *L* các phần tử có giá trị bằng *s*. Ta xây dựng hàm `remove1` xử lý 4 trường hợp như sau :

- Nếu danh sách đã cho là rỗng, thì kết quả cũng là danh sách rỗng
- Nếu phần tử đầu tiên là *s*, thì loại bỏ nó.
- Nếu phần tử đầu tiên là một danh sách thì xử lý nó và phần còn lại `cdr` sau đó ghép kết quả lại.
- Nếu phần tử đầu tiên không phải là *s*, thì xử lý phần còn lại `cdr`.

Chương trình như sau :

```
(define (remove1 s L)
    (cond ((null? L) '())   ; danh sách rỗng không chứa s
          ((equal? s (car L)) ; phần tử đầu tiên là s
              (remove1 s (cdr L)))
          ((list? (car L))          ; phần tử đầu tiên là một danh sách
              (cons (remove1 s (car L)) (remove1 s (cdr L))))
          (else (cons (car L )      ; phần tử đầu tiên khác s
              (remove1 s (cdr L))))))
(remove* 'a '(b a ((a) b) c (a) d))
--> (b (() b) c () d)
```

4. Nghịch đảo danh sách

Xây dựng hàm `reverse1` nghịch đảomọi phần tử ở mọi mức của một danh sách :

```
(reverse1 '(a (b c d) e)
--> (e (d c b) a)
```

Trong ví dụ trên, ta đã nghịch đảo các phần tử của danh sách con `(b c d)`. Xảy ra ba trường hợp như sau :

- Trường hợp danh sách rỗng thì kết quả trả về là `'()`.

– Nếu phần tử đầu tiên của danh sách là một danh sách, thì ta nghịch đảo các phần tử của phần còn lại `cdr` rồi ghép kết quả này với kết quả nghịch đảo của phần tử đầu tiên `car`.
– Nếu phần tử đầu tiên không phải là một danh sách, thì ta nghịch đảo các phần tử của `cdr` rồi ghép kết quả này với phần tử đầu tiên.

Hàm `reverse1` gọi hàm `append1` để thêm một phần tử vào cuối danh sách :

```
(define (reverse1 L)
    (cond ((null? L) '())
        ((list? (car L))
            (append1 (reverse1 (car L)) (reverse1 (cdr L))))
        (else (append1 (car L) (reverse1 (cdr L))))))
(reverse1 '(a b ((e f) g) c (i j) d))
--> '(d (j i) c (g (f e)) b a)
```

5. So sánh bằng nhau

Kiểu trừu tượng của dạng thức so sánh bằng nhau (equality) được định nghĩa như sau :

```
=        : number × number       → boolean
eq?      : symbol × any          → boolean
```

hoặc :

```
eq?      : any × symbol          → boolean
char=?   : character × character → boolean
string=? : string × string       → boolean
```

Sau đây là bảng so sánh bằng nhau theo kiểu dữ liệu Scheme.

Dữ liệu	*Phép so sánh*	*Ví dụ*
Kiểu số `number`	`=, <, <=, >, >=`	`(= 2 2) --> #t`
Kiểu ký tự `character`	`char=?`	`(char=? #\A #\a)` `--> #f`
Kiểu chuỗi `string`	`string=?`	`(string=? "123" "123")` `--> #t`

Kiểu bất kỳ :
`boolean, symbol, number, character, empty list, pair, vector, string, procedure`
	`eqv?, eq?`	`(eqv? #\A #\A)`
		`--> #t`
		`(eqv? 'toto 'toto)`
		`--> #t`
		`(eqv? "123" "123")`
		`--> #t`

Kiểu `pair, vector, string`	`equal?`	`(equal? "123" "123")` `--> #t`

Vị từ `eqv?`, hay `eq?`, dùng để so sánh các kiểu dữ liệu bất kỳ, kết quả sẽ là `#t` nếu chúng cùng một đối tượng. Riêng kiểu dữ liệu danh sách nên sử dụng vị từ so sánh `string=?`.

```
(eq? 123 123)
--> #t
```

```
(eq? 123456789012345678901 123456789012345678901)
--> #f
(eq? '(1 . 2) '(1 . 2))
--> #t
(define (f L) (eq? L L))
(f '(#\a . #\a))
--> #t
```

Ta có thể lại định nghĩa vị từ so sánh tổng quát `equal?` một cách đơn giản hơn như sau :

```
(define (equal? V1 V2)
   (cond
       ((eq? V1 V2) ; xử lý các trường hợp boolean, symbol và '()
            #t)
       ((and (number? V1) (number? V2))
            (= V1 V2))
       ((and (char? V1) (char? V2))
            (char=? V1 V2))
       ((and (string? V1) (string? V2))
            (string=? V1 V2))
       ((and (pair? V1) (pair? V2))
            (and (equal? (car V1) (car V2))
                 (equal? (cdr V1) (cdr V2))))
       (else #f)))      ; các đối tượng có bản chất khác nhau
(equal? #\a #\a)
--> #t
(equal? "123" "123")
--> #t
(equal? '(1 . 2) '(1 . 2))
--> #t
(equal? '(1 2 3 4 5) '(1 2 3 4 5) )
--> #t
```

III.4.3. Biểu diễn danh sách

III.4.3.1. Biểu diễn danh sách bởi kiểu bộ đôi

Một danh sách khác rỗng là một bộ đôi đặc biệt : chẳng hạn danh sách có một phần tử `(a)` chính là bộ đôi `'(a . ())` có phần tử thứ nhất `car` là `a` và phần tử thứ hai `cdr` là một danh sách rỗng. Một cách tổng quát, danh sách :

(s1 s2 ... sn)

được biểu diễn bởi kiểu dữ liệu bộ đôi :

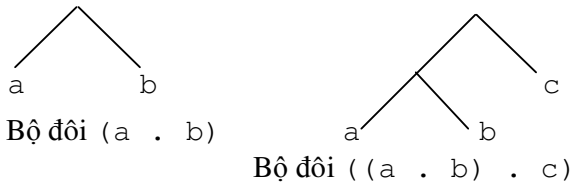
(s1 . (s2 . (... (sn . ( ))) ...)

Như vậy, một danh sách hoặc có thể rỗng, hoặc có thể là một bộ đôi có `cdr` lại là một bộ đôi khác. Các thành phần của một bộ đôi cũng có thể là các bộ đôi. Ví dụ :

```
(define x (cons 'a 'b))
(define y (cons x 'c))
```

```
y ;  xem nội dung của y
--> ((a . b ) . c)
```

Biểu diễn dạng cây (vẽ đơn giản) của các bộ đôi như sau :



Hình III.5. Biểu diễn dạng cây các bộ đôi.

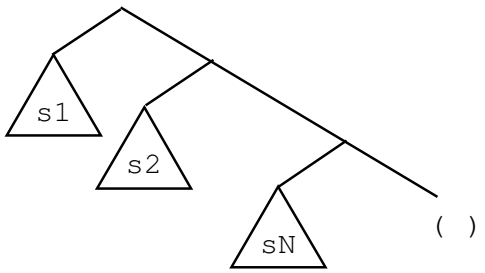
Nói cách khác, ta đã mở rộng định nghĩa của s-biểu thức là những bộ đôi. Lấy lại giá trị của `x = (1 . 2)` trong ví dụ trên đây là một bộ đôi, ta có :

```
(pair? y)
--> #t
(pair? '())  ;  danh sách rỗng không phải là một bộ đôi
--> #f
(pair? '(a b))      ; danh sách không rỗng là một bộ đôi
--> #t
```

Nghĩa là phần tử của bộ đôi có thể kiểu bộ đôi. Để in ra đơn giản, mỗi lần một cặp tạo thành một danh sách, Scheme hạn chế số dấu chấm đưa ra :

```
'(a . (1 2))
--> '(a 1 2)
'(a . (b . (c . d)))
--> (a b c . d)
```

Khi một danh sách được biểu diễn dưới dạng cây hình «cái cào» (rake), các phần tử của lần lượt là trường `car` của các bộ đôi, trường `cdr` là danh sách rỗng cuối cùng.



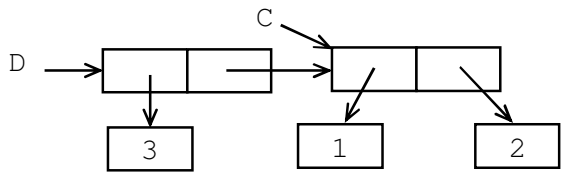
Hình III.6. Biểu diễn danh sách bởi các bộ đôi.

Các bộ đôi có thể được tiếp cận bởi nhiều con trỏ (mũi tên) khác nhau. Chẳng hạn, bộ đôi trỏ bởi `C` trong ví dụ sau đây cũng được trỏ bởi một phần tử của bộ đôi mới được xây dựng (là 5 và C) do con trỏ `D` trỏ tới :

```
(define C (cons 1 2))
(define D (cons 3 C))
D  ;  xem nội dung của D
--> (3 1 . 2)
```

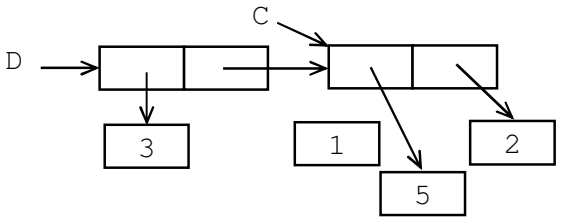
Nếu xảy ra đột biến giá trị của bộ đôi trỏ bởi `C` thì cũng dẫn đến sự đột biến dữ liệu trỏ bởi `D` :

```
(set-car! C 5)
D ; xem nội dung của D
--> (3 5 . 2)
```



Hình III.7. Nhiều mũi tên cùng biểu diễn một con trỏ.

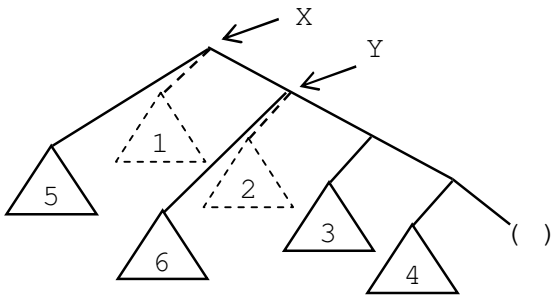
Ta có thể làm thay đổi bất kỳ đối tượng nào được xây dựng từ kiểu dữ liệu bộ đôi : một danh sách, một cây, v.v... Trong các ứng dụng của lập trình hàm, người ta thường sử dụng bộ đôi để mô hình hoá các đối tượng có trạng thái thay đổi theo thời gian.



Hình III.8. Đột biến của C cũng là đột biến của D.

Người ta cũng có thể tiết kiệm được nhiều bộ nhớ bằng cách biểu diễn mỗi bộ đôi trong một đơn vị nhớ. Sau đây ta xét một ví dụ về khả năng xảy ra hiệu ứng phụ khi thực hiện các phép đột biến trên các danh sách.

```
(define X (list 1 2 3 4))
(define Y (cdr X))      ; X và Y có cùng danh sách con (2 3 4)
(set-car! X 5) ; thay đổi X mà không ảnh hưởng đến Y
X
--> (5 2 3 4)
Y
--> (2 3 4)
```



Hình III.9. Đột biến gây ra hiệu ứng phụ.

```
(set-car! Y 6) ; đột biến gây ra hiệu ứng phụ đối với X
X
--> (5 6 3 4)
```

```
Y
--> (6 3 4)
```

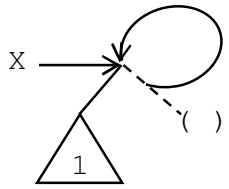
Tuy nhiên, sau khi gán lại phần `cdr` của `X` cho một danh sách mới, thì con trỏ `Y` vẫn trỏ về danh sách cũ, không thay đổi :

```
(set-cdr! X (list 7 8))   ;X và Y độc lập (không chung nhau bộ đôi nào)
X
--> (5 7 8)
Y
--> (6 3 4)
```

Sử dụng hàm đột biến `set-cdr!`, ta có thể tạo ra các danh sách nối vòng quanh với nhau như sau :

```
(define X (list 1))
X
--> '(1)
(set-cdr! X X)
(car X)
--> 1
X
--> 1 1 1 1 1 ... 1 1 1 ... ; vô hạn lần !
```

Các hàm `append`, `reverse`, `map` (sẽ xét ở chương sau), ... thường rơi vào trạng thái quẩn nếu một trong các tham đối là một danh sách nối vòng.



Hình III.10. Đột biến tạo danh sách nối vòng có thể gây ra quẩn vô hạn.

Sau đây ta xét một ví dụ sử dụng đột biến để ghép danh sách. Giả sử cho hai danh sách `L1` và `L2`, ta cần ghép chúng để nhận được một danh sách mới. Gọi `|L1|` là độ dài (length) của danh sách `L1`, ta có hai giải pháp, một giải pháp không sử dụng đột biến, như sau :

Cách 1 : tạo ra |L1| bộ đôi

```
; concat : List(T) ´ List(T) ® List(T)
(define (concat L1 L2)
  (if (null? L1)
      L2
      (cons (car L1) (concat (cdr L1) L2))))
(concat '(1 2) '(3 4 5))
--> (1 2 3 4 5)
```

Cách 2 : sử dụng đột biến.

Giả sử để đơn giản, `L1` khác rỗng và kết quả không tạo ra bộ đôi mới. Nhờ đột biến, bộ đôi cuối cùng của `L1` trỏ tới `L2`.

```
; concat! : List(T) ´ List(T) ® List(T)
(define (concat! L1 L2)
   (define (last-doublet L) ; trả về bộ đôi cuối cùng của L là null
      (if (null? (cdr L))
            L
            (last-doublet (cdr L))))
   (set-cdr! (last-doublet L1) L2))
(define x '(1 2 3))
(define y '(4 5))
(concat! x y)
x
--> (1 2 3 4 5)
y
--> (4 5)
```

III.4.3.2. Danh sách kết hợp

1. Khái niệm danh sách kết hợp

Các bộ đôi thường được dùng để định nghĩa *danh sách kết hợp* (association list), viết tắt là *alist*. Đó là một danh sách gồm các phần tử có dạng bộ đôi đồng nhất như sau :

(***(c1 . v1)... (cn . vn***)

Các alist thường được dùng để biểu diễn các cấu trúc dữ liệu như *bảng* (table), *từ điển* (dictionary), các hàm, v v... Khi muốn kết hợp một khóa ***cj*** với một giá trị ***vj***, người ta viết (***cj . vj***) để tạo ra một phần tử của danh sách kết hợp. Thư viện Scheme có hàm `assq` :

(***assq s alist***)

sử dụng phép so sánh `eq?` để trả về phần tử đầu tiên của danh sách kết hợp `alist` thoả mãn điều kiện có `car` là khóa `s`. Nếu không tìm thấy khoá nào như vậy, giá trị trả về là `#f` :

```
(assq 'a '((b . 2) (a . 1) (c . 3) (a . 0)))
--> '(a . 1)
```

Một danh sách là một trường hợp đặc biệt của bộ đôi nên kết quả trả về có thể là một phần tử không phải bộ đôi :

```
(assq 'a '((b . 2) (a 1) (c 3) (a . 0)))
--> '(a 1)
```

Do `assq` sử dụng `eq?` nên có thể phép so sánh không thành công :

```
(assq '(a) '((b . 2) (a . 1) (c . 3) ((a) . 0)))
--> #f
```

Để so sánh như vậy, Scheme còn có hàm tương tự là `assv` sử dụng phép so sánh `eqv?` và `assoc` sử dụng các phép so sánh `equal?` :

```
(assv '(a) '((b . 2) (a . 1) (c . 3) ((a) . 0)))
--> #f
(assoc '(a) '((b . 2) (a . 1) (c . 3) ((a) . 0)))
--> '((a) . 0)
```

Chú ý rằng các hàm `memq`, `memv`, `member`, `assq`, `assv`, và `assoc` đều không có dấu chấm hỏi (**?**) phía sau tên vì chúng trả về không chỉ các giá trị kiểu boolean `#f` và `#t` mà còn trả về các giá trị kiểu khác của Scheme.

Sau đây là một số ví dụ khác sử dụng các hàm `assq`, `assv`, và `assoc` để tìm kiếm trên danh sách :

```
(define e ' ((a 1) (b 2) (c 3)))
(assq 'a e)
--> '(a 1)
(assq 'b e)
--> '(b 2)
(assq 'd e)
--> #f
(assq (list 'a)'(((a)) ((b)) ((c))))
--> #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
--> '((a))
(assq 5 '((2 3) (5 7) (11 13)))
--> '(5 7)
(assv 5 '((2 3) (5 7) (11 13)))
--> '(5 7)
```

2. Sử dụng danh sách kết hợp

Do `assq` trả về một bộ đôi, ta xây dựng hàm `valof` để tìm giá trị kết hợp (là `cdr`) với khóa cần tìm (là `car`) :

```
(define (valof key alist)
   (let ((doublet (assq key alist)))
        (if doublet
             (cdr doublet)
             #f)))
(define L
    '((pluto . 9) (jerry . 8) (mickey . 10) (pony . 7)
     (tom . 8)))
(valof 'tom L)
--> 8
(valof 'iago L)
-->#f
```

Xây dựng hàm `delkey` để xóa bỏ hết các bộ đôi có khoá đã cho trong một danh sách kết hợp :

```
(define (delkey key alist)
   (cond ((null? alist) '())
         ((eq? key (caar alist))
             (delkey key (cdr alist)))
         (else (cons (car alist)
             (delkey key (cdr alist))))))
(delkey 'b '((a . 1) (b . 2) (c . 3) (b . 4)))
--> '((a . 1) (c . 3))
```

```
(delkey 'd '((a . 1) (b . 2) (c . 3) (b . 4)))
--> '((a . 1) (b . 2) (c . 3) (b . 4)))
```

Sau đây ta xây dựng hàm `sublis` nhận vào hai tham đối `alist` và `s` để thay thế các xuất hiện trong biểu thức `s` là khóa `car` trong mỗi phần tử của `alist` bởi giá trị tương ứng :

```
(define (sublis alist s)
  (cond ((pair? s)
         (cons (sublis alist (car s))
               (sublis alist (cdr s))))
        ((null? s) '())
        (else (let ((doublet (assoc s alist)))
                (if doublet
                    (cdr doublet)
                    s)))))

(sublis
   '((mot . one) (hai . two) (ba . three)))
   '(mot + hai = ba))
--> (one + two = three)
```

Để bổ sung vào danh sách kết hợp một phần tử mới, ta xây dựng hàm `acons(s1 s2 alist)` trả về kết quả là danh sách `alist` đã được thêm vào đầu một bộ đôi được xây dựng từ `s1` và `s2` :

```
(define (acons s1 s2 alist)
   (cons (cons s1 s2) alist))
(acons 'a 1 '((b . 2) (c . 3) (b . 4)))
--> '((a . 1) (b . 2) (c . 3) (b . 4))
```

III.4.3.3. Dạng quasiquote

Ngoài dạng viết tắt `'`(quotation mark) của phép trích dẫn `quote`, trong Scheme còn có dạng viết tắt ` (grave accent) của phép `quasiquote` (tạm dịch «tương tự trích dẫn»).

```
(quasiquote s)
--> 's
's
--> 's
`(a b)
--> (a b)
```

Sự khác nhau giữa `quote` và `quasiquote` liên quan đến các phần tử có đặt dấu phẩy ở trước trong một danh sách : những phần tử có dấu phẩy đặt trước như vậy được tính toán do không còn «bị trích dẫn» nữa (unquote).

Ví dụ :

```
(define x 9999)
(define y '(a b c))
`(x y ,x ,y)          ; chú ý dấu phẩy đặt trước x và y
--> '(x y 9999 (a b c))
```

Tuy nhiên vẫn có thể dùng `list` để có cùng kết quả như vậy :

```
(list 'x 'y  x y)
--> '(x y 9999 (a b c))
```

Như vậy, dùng trích dẫn `quote` cho s-biểu thức khi không cần tính giá trị các thành phần của nó, còn khi cần tính giá trị, cần đặt một dấu phẩy trước những thành phần của s-biểu thức cần tính. Nghĩa là :

`'s = `s` nếu không tồn tại các dấu phẩy trong `s`.

`'s ≠ `s` nếu tồn tại các phần tử có dấu phẩy đặt trước được tính trong `s`.

Chẳng hạn :

```
`(x y)
--> '(x y)
'(,a)
--> '((unquote a))
```

Giả sử ta muốn trả về giá trị trích dẫn `quote` của một s-biểu thức s, ta viết :

```
(list 'quote y)
--> ''(a b c)
```

hoặc viết gọn hơn :

```
`',y          ; 3 dấu liên tiếp trước y
--> ''(a b c)
```

Ta có thể viết hàm `kwote` để thực hiện việc trên như sau :

```
(define (kwote s)
    `',s)
(kwote y)
--> ''(a b c)
```

Trong Scheme còn sử dụng hai kí tự liên tiếp `,@` (dấu phẩy rồi dấu commercial at) để đặt trước một số phần tử trong một s-biểu thức có quasiquote. Khi một phần tử có hai dấu này đứng trước, nó được tính giá trị nhưng giá trị phần tử này phải là một danh sách và nội dung của danh sách kết quả sẽ thay thế nó.

```
`(x y ,x ,@y)
--> '(x y 9999 a b c)
```

Ký tự `@` tượng trưng cho chữ `a` trong `append` vì ta có thể nhận được cùng một kết quả nhờ `append` :

```
(append `(x y ,x) y)
--> '(x y 9999 a b c)
```

Kỹ thuật này thường được dùng khi cần tạo sinh các biểu thức hàm.

III.4.4. Một số ví dụ ứng dụng danh sách

1. Tìm phần tử cuối cùng của danh sách

Xây dựng hàm trả về phần tử cuối cùng của một danh sách, trả về `#f` nếu danh sách rỗng.

; Cách thứ nhất

```
(define (last1 L)
```

; hoặc `(define last (lambda (L) … )`

```
 (cond ((null? L) #f) ; danh sách rỗng
          ; danh sách chỉ có một phần tử duy nhất
          ((null? (cdr L)) (car L))
          (else (last1 (cdr L)))))

(last1 '(1 2 3 -5 -6 0 7 a))
--> a

(last1 '())
--> #f
```

; Cách thứ hai : sử dụng các hàm có sẵn `list-ref` và `length`
; để lấy phần tử cuối cùng nếu danh sách khác rỗng

```
(define (last2 L)
(if (null? L)
        #f
        (list-ref L (- (length L) 1))))

(last2 '(1 2 3 -5 -6 0 7 8))
--> 8
```

2. Liệt kê các vị trí một ký hiệu có trong danh sách

; Cách thứ nhất : sử dụng sơ đồ lặp và hàm `reverse`
; Vị trí các phần tử được tính từ 0 trở đi

```
(define (listocc1 s L)
   (define (ilo s L i R)
        (cond ((null? L) R)
              ((equal? (car L) s)
              (ilo s (cdr L) (+ i 1) (cons i R)))
              (else (ilo s (cdr L) (+ i 1) R))))
   (reverse (ilo s L 0 '())))

(listocc1 'a '(1 a 3 a 5 a 7 a))
--> (1 3 5 7)
```

; Cách thứ hai : sử dụng sơ đồ lặp và hàm `append`

```
(define (listocc2 s L)
   (define (ilo s L i R)
        (cond ((null? L) R)
              ((equal? (car L) s)
              (ilo s (cdr L) (+ i 1)
                      (append R (list i))))
              (else (ilo s (cdr L) (+ i 1) R))))
   (ilo s L 0 '()))

(listocc2 'a '(1 0 a 3 a 5 a 7 2 1 a))
--> (2 4 6 10)
```

3. Tìm tổng con lớn nhất trong một vector

Cho một danh sách *L* gồm *N* số thực, cần tìm tổng lớn nhất trong tất cả các tổng của các danh sách con của *L* (gọi tắt là tổng con lớn nhất). Chẳng hạn danh sách :

L = (*18 -20 35 12 28 -5 -42 30 -50 45*)

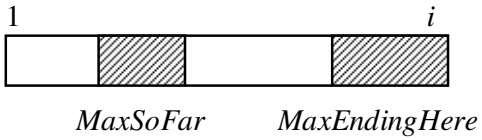
có tổng con lớn nhất là *75* tương ứng với danh sách con (*35 12 28*)

Bài toán này xuất hiện trong thủ tục so mẫu hai chiều để xử lý ảnh số do Ulf Grenander, Brown University, đề xuất năm 1977 : cho trước ma trận $N \times N$ số thực, cần tìm ma trận con có tổng lớn nhất trong tất cả các ma trận con có thể.

Vì độ phức tạp tính toán quá lớn, bài toán được đưa về bài toán một chiều. Tuy nhiên, bài toán một chiều chỉ tỏ ra đơn giản khi mọi phần tử là số dương, khi đó kết quả chính là danh sách đã cho. Nếu mọi phần tử là số âm thì kết quả là 0, tương ứng với danh sách rỗng. Nhưng khi danh sách có cả số âm và số dương tuỳ ý, bài toán lại trở nên rất phức tạp. Đã có nhiều lời giải cho bài toán này :

- Thuật toán lập phương với chi phí $O(N^3)$.
- Thuật toán bình phương với chi phí $O(N^2)$.
- Thuật toán đệ quy «chia để trị» do M. Shamos đề xuất với chi phí $O(NlogN)$.
- Thuật toán quét (tốt nhất) do Jay Kadane đề xuất với chi phí tuyến tính $O(N)$.

Ý tưởng của thuật toán quét như sau : để tìm tổng con lớn nhất của danh sách *L[1..i]*, *i=1..N*, giả thiết rằng đã có kết quả (đã xử lý) cho *L[1..i-1],* là *MaxSoFar*. Khi đó, tổng con lớn nhất của *L[1..i]*, hoặc là *MaxSoFar*, hoặc là tổng con lớn nhất kết thúc tại *i*, gọi là *MaxEndingHere* (phần gạch gạch trong hình vẽ sau đây).



Hình III.11. Thuật toán quét tìm tổng con lớn nhất.

Thuật toán quét được viết bằng giả ngữ phỏng Pascal như sau :

```
MaxSoFar := MaxEndingHere := 0
for i := 1 to N do begin
   MaxEndingHere ← max(MaxEndingHere+L[i], 0)
   MaxSoFar ← max(MaxSoFar, MaxEndingHere)
end
```

Sau đây là thủ tục Scheme tìm tổng con lớn nhất của danh sách sử dụng phép lặp :

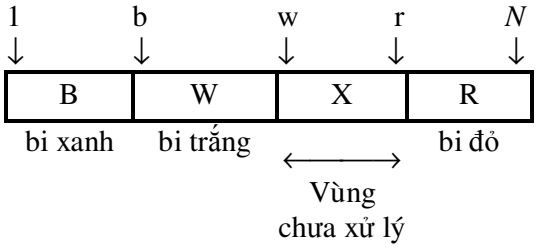
```
(define (maxsumlist L)
    (define (msl-it L Ms Me)
        (if (null? L)
             Ms
             (let ((m (max (+ Me (car L)) 0)))
                  (msl-it (cdr L) (max m Ms) m))))
    (msl-it L 0 0))
(maxsumlist '(18 -20 35  12  28  -5 -42  30 -50 45))
--> 75
```

4. Bài toán sắp xếp dãy viên bi ba màu

Bài toán sắp xếp dãy các viên bi ba màu, hay còn được gọi là bài toán cờ ba màu (drapeau français) được phát biểu như sau : Cho trước một dãy các viên bi đánh số từ 1 đến *N*, mỗi viên bi mang một màu hoặc xanh, hoặc trắng, hoặc đỏ. Cần sắp xếp lại các viên bi theo thứ tự

lần lượt xanh, đến trắng, rồi đến đỏ sao cho chỉ được hoán vị (đổi chỗ các viên bi) ngay trên dãy đã cho, hơn nữa, các hoán vị phải càng ít càng tốt.

Để giải bài toán này, ta đưa vào các vị từ *B* (blue), *W* (white) và *R* (red) với quy ước *B*(*i*), *W*(*i*) và *R*(*i*) là đúng nếu và chỉ nếu viên bi thứ *i* ($1 \le i \le N$) là xanh, trắng và đỏ tương ứng. Ta cũng sử dụng thủ tục hoán vị *permute*(*i, j*) để đặt viên bi thứ *i* thành *j*, viên bi thứ *j* thành *i*, $\forall\ i, j \in 1..N$, không loại trừ trường hợp $i = j$.



Hình III.12. Bài toán sắp xếp dãy các viên bi ba màu.

Sử dụng 3 chỉ số *b, w* và *r* để phân cách 4 vùng của mảng (xem hình), thuật toán sắp xếp các viện bi là xử lý vùng chưa được sắp xếp *X* :

```
w:= 1; b := 1; r := n;
while w <= r do
     if W(w) then w:= w+1 elsif B(w) then begin
       permute(b, w); b:= b+1; w:= w+1
     end else begin while (R(r) and w < r) then r:= r-1;
       permute(r, w); r:= r-1
     end
```

Thủ tục Scheme sắp xếp dãy các viên bi ba màu sử dụng phép lặp :

```
(define (threecolor L)
   (define (blue? x)  (if (equal? 'b x) #t #f))
   (define (white? x) (if (equal? 'w x) #t #f))
   (define (red? x)   (if (equal? 'r x) #t #f))
   (define (threecolor_itr L b w r)
    (if (null? L)
       (append b w r)
       (let ((x (car L)) (y (cdr L)))
          (cond
             ((blue? x)
                (threecolor_itr y (cons x b) w r))
             ((white? x)
                (threecolor_itr y b (cons x w) r))
             (else
                (threecolor_itr y b w (cons x r)))))))
(threecolor_itr L '() '() '()))
(threecolor '(b w w r b r b r w w r b b w))
--> '(b b b b b w w w w w r r r r)
```

5. Sắp xếp nhanh quicksort

Trong chương 1, ta đã xây dựng thuật toán quicksort sắp xếp nhanh các phần tử của một danh sách theo thứ tự không giảm viết bằng Miranda. Sau đây ta sẽ minh hoạ bằng Scheme.

Đầu tiên, ta xây dựng hàm phân chia `partition` cho phép chuyển một danh sách thành một danh sách khác (theo thứ tự ngược lại) mà phần tử đầu của nó là danh sách con chứa các giá trị nhỏ hơn hoặc bằng phần tử trục `pivot` đã cho, danh sách con còn lại chứa các giá trị lớn hơn phần tử trục. Hàm có sử dụng hai danh sách «mồi» lúc đầu cả hai đều rỗng.

```
(define (partition L pivot Linf Lsup)
   (cond ((null? L) (cons Linf Lsup))
         ((< (car L) pivot)
          (partition
             (cdr L) pivot (cons (car L) Linf) Lsup))
         (else (partition
             (cdr L) pivot Linf (cons (car L) Lsup)))))
(partition '(6 4 7 8 5 9 2 3) 7 '() '())
--> '((3 2 5 4 6) 9 8 7)
```

Bây giờ xây dựng hàm `quicksort` gọi đệ quy việc chọn phần tử trục là phần tử đầu danh sách để áp dụng hàm phân chia, các danh sách kết quả và phần tử trục được ráp lại với nhau và trả về kết quả là danh sách đã được sắp xếp.

```
(define (quicksort L)
   (if (or (null? L) (null? (cdr L)))
       L
       (let* ((pivot (car L))
              (L1-L2
                  (partition (cdr L) pivot '() '()))
              (L1 (car L1-L2))
              (L2 (cdr L1-L2)))
         (append (quicksort L1)
                 (cons pivot (quicksort L2))))))
(quicksort '(7 6 4 7 8 5 9 2 3))
--> '(2 3 4 5 6 7 7 8 9)
```

Tóm tắt chương 3

- Kiểu dữ liệu phức hợp trong Scheme gồm kiểu chuỗi, kiểu vectơ, kiểu bộ đôi, kiểu danh sách, kiểu dữ liệu thủ tục và kiểu dữ liệu cổng.
- Tất cả các kiểu dữ liệu của Scheme đều được gọi là kiểu s-biểu thức.
- Có tất cả 9 kiểu dữ liệu được gọi là đối tượng của Scheme :

Tên kiểu	*Ví dụ*	*Vị từ kiểm tra kiểu*
Số	9	`number?`
Ký tự	`#\a` ou `#\space`	`char?`
Lôgic	`#t` hoặc `#f`	`boolean?`
Ký hiệu	`'chip`	`symbol?`
Bộ đôi	`( 5 . 7 )`	`pair?`
Chuỗi	`"Tom and Jerry"`	`string?`
Vectơ	`#( 5 3 12 "Mickey" 32 )`	`vector?`
Thủ tục	---	`procedure?`
Cổng	---	`port?`

- Trong Scheme cũng như trong ngôn ngữ lập trình hướng đối tượng, người ta có thể sử dụng kiểu dữ liệu trừu tượng để định nghĩa các cấu trúc dữ liệu phức tạp.
- Một kiểu dữ liệu trừu tượng gồm 4 thành phần : tên kiểu, các định nghĩa hàm, các điều kiện đầu nếu có và các tiên đề. Hai thành phần đầu mô tả cú pháp về mặt cấu trúc thuộc tính của kiểu dữ liệu trừu tượng, hai thành phần sau mô tả ngữ nghĩa.
- Scheme sử dụng kiểu dữ liệu bộ đôi tương tự bản ghi gồm một cặp phần tử nào đó có thứ tự. Phần tử thứ nhất được gọi là `car`, phần tử thứ hai `cdr`.Có một dấu chấm để phân cách hai giá trị phần tử.
- Kiểu dữ liệu bộ đôi được sử dụng để xây dựng kiểu danh sách : một danh sách hoặc có thể rỗng, hoặc có thể là một bộ đôi có `cdr` lại là một bộ đôi khác. Các thành phần của một bộ đôi cũng có thể là các bộ đôi.
- Kiểu danh sách được sử dụng phổ biến trong Scheme. Một danh sách được xem là một cấu trúc gồm hai thành phần : phần tử đầu và phần danh sách còn lại.
- Khi xử lý một danh sách, người ta xử lý phần tử đầu, sau đó sử dụng phép đệ quy để xử lý phần danh sách còn lại.

Bài tập chương 3

1. Giải thích các biểu thức sau đây, sau đó tính giá trị và so sánh kết quả :

```
(cons 1 2)
(car (cons (cons 1 2) (cons 3 4)))
(cons (cons (cons (cons 1 2) 3) 4) 5)
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 ()))))
(list 1 2 3 4 5)
(car (list 1 2 3 4 5))
(cdr (list 1 2 3 4 5))
(cadr (list 1 2 3 4 5))
(caddr (list 1 2 3 4 5))
```

2. Viết hàm tạo các danh sách sau :

```
(a b c d)
(a ((b c) d (e f)))
(a (b (c d) . e) (f g) . h)
```

3. Cho biết những biểu thức nào có cùng kết quả trong số các biểu thức sau đây :

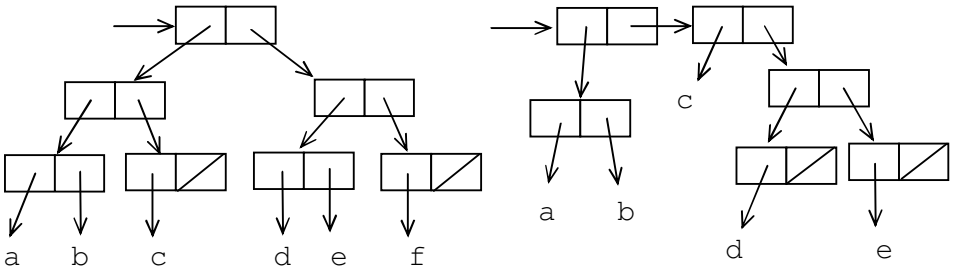
```
(list 1 '(2 3 4))
(append '(1) '(2 3 4))
(list 1 2 3 4)
(cons '1 '(2 3 4))
(cons '(1) '(2 3 4))
(cons '1 '((2 3 4)))
(append '(1) '((2 3 4)))
```

4. Cho biết giá trị của các biểu thức sau :

```
'(+ 4 7)                    '(a b)
'5                          (cons 'a '((b c)))
```

```
(cdr '(a))                    (cdr week-list)
(car '((+ 4 1)))              (cdr '((+ 4 1)))
(cdr '((a) b))                (cdr '((a) (b)))
(cdr ''(a b))
```

5. Cho biết các danh sách tương ứng với các sơ đồ sau :



1) 2)

6. Từ hàm `member`, hãy định nghĩa vị từ `member?`.

7. Định nghĩa một hàm để phá hết các ngoặc trong một danh sách.

   Chẳng hạn, đối với danh sách `((a b c) (d (e f) g) h (i j))`
   thì hàm trả về : `(a b c d e f g h i j)`

8. Hàm `concat` sau đây dùng để ghép hai danh sách tương tự `append` :

```
(define (concat list1 list2)
  (define (iter response remaining)
    (if (null? remaining)
        (reverse response)
        (iter (cons (car remaining) response)
              (cdr remaining))))
  (iter list2 list1))
```

   Tuy nhiên hàm không trả về kết quả đúng, hãy viết lại cho phù hợp, sao cho :

```
(concat '(1 2 3 4 5) '(6 7 8 9))
--> '(1 2 3 4 5 6 7 8 9)
```

9. Viết biểu thức trích danh sách để trả về kết quả là danh sách con `'(sat, sun)` : `'(mon tue wed thu fri sat sun)`.

10. Viết các hàm trả về phần tử thứ hai, thứ ba và thứ tư của một danh sách.

11. Viết dạng tổ hợp của `car` và `cdr` để nhận được giá trị là ký hiệu `a` từ các danh sách : `((b a) (c d))`, `(() (a d))`, `(((a)))`

12. Cho biết giá trị của `(car ''a)` và `(cdr ''a)` (chú ý hai dấu quote).

13. Viết định nghĩa tương tự định nghĩa của kiểu `list` cho kiểu `plate-list`.

14. Viết hàm `(count s L)` đếm số lượng ký hiệu `s` là chữ cái xuất hiện trong danh sách chữ cái `L`. Ví dụ :

```
(count 'R '(T O M A N D J E R R Y))
--> 2
```

15. Viết hàm `(double L)` nhận vào một danh sách các ký hiệu `L` để trả về danh sách mà các ký hiệu đã được viết lặp lại. Ví dụ :

```
(double '(TOM AND JERRY))
--> '(TOM TOM AND AND JERRY JERRY)
```

16. Viết hàm `(undouble L)` nhận vào một danh sách các ký hiệu `L` trong đó các ký hiệu đều bị viết lặp lại để trả về danh sách chỉ chứa mỗi ký hiệu một lần. Ví dụ :

```
(undouble (double '(TOM AND JERRY)))
--> '(TOM AND JERRY)
```

17. Từ ví dụ xử lý hình chữ nhật trình bày trong lý thuyết, viết vị từ `disjoint?` trả về `#t` nếu hai hình chữ nhật rời nhau, nghĩa là không có điểm nào chung.

18. Xây dựng các hàm xử lý hình chữ nhật sử dụng biểu diễn các thành phần bởi danh sách.

19. Cho biết giá trị các biểu thức dạng quasiquode sau đây :

```
`(1 + 2 = ,(+ 1 2))
`(the car of the list (a b) is ,(car '(a b)))
`(cons ,(+ 2 5) ,(list 'a 'b))
(let ((L '(1 2 3)))
    `((+ ,@L) = ,(+ 1 2 3)))
```

20. Dùng kiểu bộ đôi (pair-doublet) để biểu diễn số phức (a + b*i*). Hãy tính cộng, nhân và luỹ thừa bậc n nguyên dương của một số phức. Cho biết :

Cộng: $(a + bi) \pm (c + di) = (a \pm c) + (b \pm d)i$

Trừ : $(a + bi) - (c + di) = (a - c) + (b - d)i$

Nhân : $(a + bi) \times (c + di) = (ac - bd) + (ad \pm bc)i$

Chia : $\dfrac{(a+bi)}{(c+di)} = \dfrac{(ac+bd)}{(c^2+d^2)} + \dfrac{(bc-ad)}{(c^2+d^2)}i$, với điều kiện $c^2 + d^2 \neq 0$.

Luỹ thừa : $(a + bi)^n = r^n(\cos n\varphi + i\sin n\varphi)$, trong đó :

$$r = \sqrt{a^2 + b^2}, \quad \varphi = \operatorname{arctg}\frac{b}{a}$$

Căn bậc hai : $\sqrt{a+bi} = x + yi$, trong đó :

$$x = \sqrt{\frac{a}{2} + \sqrt{\left(\frac{a}{2}\right)^2 + \left(\frac{b}{2}\right)^2}}, \quad y = \sqrt{-\frac{a}{2} + \sqrt{\left(\frac{a}{2}\right)^2 + \left(\frac{b}{2}\right)^2}}$$

Nếu $a > 0$, tính x và lúc đó, $y = \dfrac{b}{2x}$, nếu $a < 0$, tính y và lúc đó, $x = \dfrac{b}{2y}$.

CHƯƠNG IV. KỸ THUẬT XỬ LÝ HÀM

Chương này sẽ trình bày chi tiết hơn một số kỹ thuật sử dụng và xử lý hàm như : dùng tên hàm làm tham đối, dùng hàm làm giá trị trả về của một hàm khác, định nghĩa hàm nhờ phép tính lambda, tham đối hoá từng phần, kỹ thuật lập trình nhờ trao đổi thông điệp giữa các hàm, kỹ thuật đệ quy cục bộ, phương pháp tổ hợp các hàm, định nghĩa các hàm có số lượng tham đối bất kỳ, v.v…

IV.1 Sử dụng hàm

Cho đến lúc này, ta đã sử dụng `define` để định nghĩa biến và hàm Scheme như sau :

(*define v s*)

trong đó : ***v*** (variable) là một biến
s là một biểu thức

(*define (f L) s*)

trong đó : ***L*** dãy từ *0.. n* biến,
f là tên hàm hay tên biến,
s là biểu thức đóng vai trò thân của hàm

Một hàm trong Scheme được xem như một kiểu dữ liệu hay một s-biểu thức, do đó, hàm có thể được dùng làm tham đối và cũng có thể làm giá trị trả về. Ta sẽ xét lần lượt hai khả năng này và một cách khác định nghĩa hàm nhờ *phép tính lambda* (λ-calculus).

IV.1.1. Dùng tên hàm làm tham đối

Giả sử ta định nghĩa một hàm tuỳ ý như sau :

```
(define (f x) (list x x))
```

```
(f 'a)
--> '(a a)
```

Bây giờ ta định nghĩa hàm `g` qua `f` như sau :

```
(define g f)
```

Nếu ta gọi hàm `g` như là một biến thì sẽ gây ra lỗi sai :

```
g
--> ERROR: '#{Procedure 6703 f}
```
(báo lỗi trong Scheme48)

Hàm `g` phải được gọi tương tự hàm `f` để có cùng kết quả :

```
(g 'a)
--> '(a a)
```

Giả sử ta định nghĩa lại hàm `car` bởi một tên khác như sau :

```
(define first car)

first
--> ERROR: #<primitive-procedure car>
; sai vì first là hàm, không phải biến

(first (list 1 2 3))
--> 1
```

Như vậy, nếu một hàm có một tham đối `f1` nào đó đã được định nghĩa trước đó, thì mọi định nghĩa dạng :

```
(define (f2 x) (f1 x))
```

đều có thể viết ngắn gọn thành :

```
(define f2 f1)
```

Ta có thể mở rộng cho các hàm nhiều tham đối :

```
(define (g2 x y z) (g1 x y z))
(define (g1 x y z) (+ x y z))
(procedure? g2)   ; g2 là một hàm
--> #t
(g2 (+ 1 2) 4 5)
--> 12
```

Trong kiểu dữ liệu trừu tượng, hai lời gọi thực hiện `(* 2 2)` và `(* 3 3)` của dạng hàm nhân `(* num num)` có nguồn gốc từ khai báo hàm :

```
* : number × number → number
```

để từ đó ta có định nghĩa hàm `square` :

```
(define (square x) (* x x))
```

Giả sử ta cần xây dựng hàm `sum-integer` tính tổng các số nguyên và hàm `sum-square` tính tổng các bình phương các số nguyên giữa hai số nguyên `a` và `b` đã cho. Ta có :

```
; Integer × Integer → Integer
(define (sum-integer a b)
  (if (> a b)
      0
      (+ a (sum-integer (+ a 1) b))))
; Integer × Integer → Integer
(define (sum-square a b)
  (if (> a b)
      0
      (+ (square a) (sum-square (+ a 1) b))))
```

Cả hai hàm trên đều có cùng một «giuộc» là :

```
(define (sum-any a b)
   (if (> a b)
        0
        (+ (func a) (sum-any (+ a 1) b))))
```

Ta có thể xem hàm `sum-any` sử dụng `func` như là tham đối hình thức của hàm. Từ đó ta có thể định nghĩa lại hàm này nhưng có 3 tham đối như sau :

```
(define (sum-any func a b)
; (Integer → Integer) × Integer × Integer → Integer
   (if (> a b)
        0
        (+ (func a) (sum-any func (+ a 1) b))))

(sum-any square 9 16)
--> 1292

(sum-any sqrt 16 25)
--> 45.1646
```

Do `func` là một hàm bất kỳ nên ta có thể mở rộng hàm `sum-any` để thực hiện phép cộng các số thực, có dạng tổng quát hơn như sau :

```
(number → number) × integer × integer → number

(sum-any sqrt 1.2 4.5) ; tính tổng các căn bậc hai các số nguyên
--> 6.41693
```

Hàm `sum-any` cho phép định nghĩa các hàm riêng biệt bằng cách thêm một định nghĩa hàm thực hiện chức năng của `func` để không sử dụng `func` như là tham đối nữa :

```
(define (sum-square a b)
   (define (square x) (* x x)) ; định nghĩa hàm func= square
   (sum-any square a b))

(sum-square 4 9)
--> 271

(define (sum-integer a b)
   (define (id x) x)    ; định nghĩa hàm func= id
   (sum-any id a b))

(sum-integer 4 9)
--> 39

(define (sum-sqrt a b)
   (sum-any sqrt a b))

(sum-sqrt 4 9)
--> 15.1597

(sum-any sqrt 4 9)
--> 15.1597
```

Sau đây là một ví dụ khác sử dụng kỹ thuật dùng tên hàm làm tham đối. Giả sử ta cần tính đạo hàm f' của một hàm f tại một điểm x theo công thức :

$$f'(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}$$

Số gia *dx* phải tương đối nhỏ sao cho ước lượng là đúng. Hàm tính đạo hàm được định nghĩa như sau :

```
; derivative: (number → number) = number × number → number
(define (derivative f x dx)
   (/ (- (f (+ x dx)) (f x)) dx))
```

Trong định nghĩa hàm `derivative`, ta đã sử dụng tên hàm để tính toán một cách hình thức, chỉ khi có lời gọi, tên hàm mới được nhận hàm cụ thể. Tên hàm xuất hiện như là một trong 3 tham đối :

```
(derivative sqrt 5 .00001)
--> 0.223607
(define (cube x) (* x x x))
(derivative cube 5 .00001)
--> 75.0001
```

IV.1.2. Áp dụng hàm cho các phần tử của danh sách

Giả sử cần xây dựng hàm có các tham đối là kết quả áp dụng một hàm *f* nào đó lên các phần tử x_i của danh sách :

$(x_1, \ldots, x_n)$

để trả về giá trị là một danh sách :

$(f(x_1) \ldots f(x_n))$

ta có thể định nghĩa hàm `mapto` như sau :

```
(define (mapto f L)
   (if (null? L)
       '()
       (cons (f (car L))
             (mapto f (cdr L)))))
(mapto odd? '(4 1 8 5 0 -5 ))
--> '(#f #t #f #t #f #t)
```

Thư viện Scheme có các hàm `map`, `for-each`, `apply` nhận tham đối là kết quả áp dụng một hàm nào đó cho các phần tử của danh sách.

Cú pháp hàm `map` như sau :

```
; map : (T1×T2×...×Tn → T)×List(T1)×...×List(Tn) → List(T)
(map f L1 L2 ... )
```

Hàm `map` đưa ra lời gọi áp dụng một hàm ***f***, hay một phép toán, cho mỗi phần tử lần lượt là của các danh sách `L1`, `L2`, ... , sau đó trả về một danh sách của tất cả các kết quả. Ví dụ :

```
(map cadr '((a b) (d e) (g h)))
--> (b e h)
(map square (list 1 2 3 4)) ; square tính bình phương một số
--> (1 4 9 16)
(map list (list 1 2 3 4))
--> ((1) (2) (3) (4))
```

```
(map + (list 1 2 3 4) (list 4 5 6))
--> (5 7 9)
(map append (list) (list))
--> '()
(map cons '(1 2 3) '(10 20 30))
--> ((1 . 10) (2 . 20) (3 . 30))
(map + '(1 2 3) '(10 20 30))
--> (11 22 33)
```

Chú ý rằng một số dạng đặc biệt không thể sử dụng như hàm để làm tham đối `f` của `map`, chẳng hạn :

```
(map or '(#f #t #f) '(#t #f #t))
--> ERROR
```

Muốn sử dụng những dạng đặc biệt như vậy cần sử dụng biểu thức lambda (sẽ xét trong mục tiếp theo) :

```
(map (lambda (x y) (or x y))
         '(#f #t #f) '(#t #f #f))
--> '(#t #t #f)
```

Cú pháp của `for-each` như sau :

(***for-each f L1 L2 ...***)

Hàm `for-each` thực hiện tương tự `map`, áp dụng hàm `f` lần lượt cho mỗi phần tử trong các danh sách nhưng chỉ khi xảy ra hiệu ứng phụ (side-effects).

```
(for-each display
    (list "one " "two " "buckle my shoe"))
--> one two buckle my shoe
```

Cú pháp hàm `apply` :

(***apply f L1 L2 ...***)

Hàm `apply` của Scheme cho phép áp dụng một hàm cho một danh sách các tham đối (chú ý tham đối đầu tiên phải được viết tách riêng) :

```
(apply + '(2 3 5))
--> 10
(apply * 2 '(3 5))
--> 30
(apply * '(2 3 5))
--> 30
```

Sử dụng thủ tục lặp `map` có thể tác động xuống các mức sâu hơn của một danh sách. Chẳng hạn ta cần tính độ sâu của một s-biểu thức có dạng danh sách, ta xây dựng hàm `depth` hoạt động như sau : nếu danh sách khác rỗng, cộng 1 vào độ sâu tối đa của mỗi phần tử của danh sách.

```
(define (depth s)
    (cond ; kiểu nguyên tử có độ sâu 0
          ((atom? s) 0)
          ; danh sách phẳng có độ sâu 1
          ((null? s) 1)
          ; xử lý danh sách khác rỗng
```

```
            (else (+ 1
                     (apply max (map depth s))))))
(depth 'a)
--> 0
(depth '(a))
--> 1
(depth '(a (b c) ((d (e))) "yes" ()))
--> 4
```

IV.1.3. Kết quả trả về là hàm

Trong Scheme, hàm có thể được dùng như giá trị kết quả trả về của một hàm khác. Chẳng hạn hàm tính đạo hàm *f* ' của một hàm *f* nào đó vừa được định nghĩa trên đây được sửa lại chỉ còn 2 tham đối, gồm tên hàm cần tính đạo hàm *f* và số gia *dx*. Sau khi thực hiện, hàm trả về một hàm khác để chờ tham đối thứ ba, là *x* :

```
;derivative: (number → number) × number → (number → number)
(define (derivative f dx)
   (define (estimation-derivative x)
      (/ (- (f (+ x dx)) (f x)) dx))
   estimation-derivative) ; chờ tham đối x mới thực hiện tiếp

(define cube-prime (derivative cube .001))
(cube-prime 5)
--> 75.0150010000254
```

Tuy nhiên ta có thể gọi trực tiếp :

```
((derivative cube .001) 5)
--> 75.0150010000254
```

Chú ý rằng trong lời gọi trên, hàm `derivative` được thực hiện để trả về hàm «chờ» `estimation-derivative` một cách «dở dang». Chỉ khi có tham đối *x*, nó mới thực hiện trọn vẹn. Một cách tổng quát, khi có một lời gọi :

(***f arg1 ... argn***)

thì hàm `f` cần được thực hiện, `f` có thể là một dạng hàm (hay một thủ tục) cần tính toán để trả về một hàm không chứa tham đối. Một hàm không chứa tham đối có thể dùng để ghi nhớ một phép tính chưa thực hiện ngay, mà chờ đợi sau đó tuỳ theo yêu cầu có thể gọi sử dụng bằng cách cung cấp tham đối thực sự. Người ta gọi đây là *lời hứa tính toán* (evaluation promise).

Chẳng hạn `f` là dạng hàm :

```
((if (= 1 2) + -) 3 4)
--> -1
((if #t + -) 3 4)
--> 7
```

Ở đây, dạng hàm (`if e1 e2 e3`) kiểm tra biểu thức điều kiện `e1` để trả về `e2` (tương ứng với phép toán `+` theo ý đồ) hoặc trả về `e3` (tương ứng với phép toán –). Giả sử ta cần xây dựng hàm `increment` một tham đối *x*, trả về một hàm để gia thêm một lượng vào *x* như sau :

```
; increment: (number → number) → (number → number)
```

```
(define (increment x)
   (define (inc y)
      (+ x y))
   inc) ; trả về tên hàm chờ tham biến thứ hai
((increment -7) 2)
--> -5
((increment 7) -2)
--> 5
```

Trong hai lời gọi trên, *x* lần lượt lấy các giá trị 2 và –2, còn y lấy –7 và 7.

IV.2 Phép tính lambda

IV.2.1. Giới thiệu phép tính lambda

Phép tính lambda (λ-calculus) do A. Church đề xuất và phát triển vào những năm 1930 của thế kỷ trước. Phép tính lambda là một hệ thống quy tắc nhưng có khả năng biểu diễn, cho phép mã hoá tất cả các hàm đệ quy (recursive functions). Phép tính lambda xem một hàm như là một phương pháp tính toán, khác với tiếp cận truyền thống của toán học xem một hàm là một tập hợp các điểm có mối quan hệ với nhau. Cùng với lý thuyết máy Turing, phép tính lambda là mô hình tính toán thực hiện tính toán trên lớp các *hàm tính được* (calculable functions).

Có hai cách tiếp cận trong phép tính lambda : tiếp cận *thuần tuý* (pure) dùng để nghiên cứu các chiến lược tính hàm và tiếp cận có *định kiểu* (typed) dùng để định kiểu các biểu thức. Mỗi cách tiếp cận đều có thể dùng để biểu diễn cách tiếp cận kia.

Về mặt cú pháp, phép tính lambda sử dụng một tập hợp *Var* gồm các *biến* : *x, y, z*, ..., một phép trừu tượng ký hiệu λ và một *áp dụng* (application) để âënh nghéa các *hạng* (term). Người ta gọi *t* là một hạng λ nếu :

- $t = x$, với $x \in Var$ là một biến,
- $t = \lambda x.M$, với $x \in Var$ là một biến, *M* là một hạng λ, được gọi là một trừu tượng,
- $t = (M\ N)$ với *M* và *N* là các hạng λ, được gọi là một áp dụng.

Ví dụ sau đây là các hạng λ hay biểu thức λ :

$(x\ x)$
$\lambda x.\lambda y.(x\ (y\ z))$
$\lambda x.(((x\ y)\ (x\ x))$
$((\lambda x.(x\ x))(\lambda x.(x\ x)))$

Trong ngôn ngữ hình thức, giả sử gọi <λ-*t*> là một hạng λ, ta có thể định nghĩa một văn phạm *G* trên bảng chữ Σ như sau :

<λ-*t*> ::= <*x*> | λ<*x*>.<λ-*t*> | (<λ-*t*> <λ-*t*>)

<*x*> ::= *Var*

Một cách trực giác, phép trừu tượng $\lambda x.M$ thể hiện hàm :

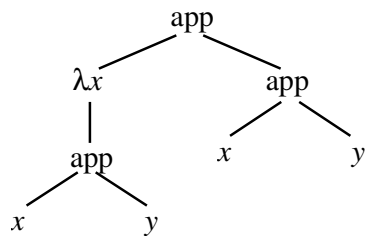
$x \rightarrow M$

còn áp dụng (M *N*) thể hiện việc áp dụng hàm *M* cho tham đối *N*.

Người ta cũng biểu diễn hạng λ bởi một cây cú pháp. Chẳng hạn hạng :

$(\lambda x.(x\ y)\ (x\ x))$

được biểu diễn bởi cây cú pháp như sau (app là các áp dụng) :



Hình IV.1. Cây biểu diễn một biểu thức lambda.

Phép trừu tượng luôn tác động lên một biến duy nhất. Khi cần biểu diễn một hàm có nhiều tham đối :

$x_1, x_2, \ldots, x_n \rightarrow M$

người ta sử dụng kỹ thuật tham đối hoá từng phần (currying) để đưa về dạng trừu tượng chỉ sử dụng một biến :

$\lambda x_1.(\lambda x_2. \ldots (\lambda x_n.M) \ldots)$

Để thuận tiện cho việc trình bày và dễ đọc các biểu thức lambda, người ta thường dùng quy ước như sau :

- Phép trừu tượng là kết hợp phải : $\lambda x.\lambda y.\lambda z.M$ có thể được viết $\lambda xyx.M$
- Các áp dụng là kết hợp trái : $((M\ N)\ P)$ có thể được viết $M\ N\ P$

Ví dụ $\lambda x.\lambda y.((x\ y)\ z)$ có thể hiện viết : $\lambda xy.xyz$.

IV.2.2. Biểu diễn biểu thức lambda trong Scheme

Trong Scheme, để biểu diễn theo cú pháp lambda một hàm có hai tham số :

$(x, y) \rightarrow \sqrt{x + y}$

người ta viết như sau :

```
(lambda (x y) (sqrt (+ x y)))
```

Một cách tổng quát :

(***lambda arg-list body***)

trong đó, `body` là một s-biểu thức, còn `arg-list` có thể là một trong các dạng :

```
x
(x1 ...)
(x1 ... xn-1 . xn)
```

Giá trị một biểu thức lambda là một *hàm nặc danh* (anonymous function), khi sử dụng tính toán để đưa ra kết quả cần phải cung cấp tham đối thực sự :

```
((lambda (x y) (sqrt (+ x y))) 3 13)
--> 4.
((lambda (x y L) (cons x (cons y L))) 'a 'b '(c d))
--> '(a b c d)
```

Để định nghĩa hàm hằng, người ta sử dụng biểu thức lambda không có tham biến. Ví dụ :

```
(lambda ()
     (display 1)
     (display " < = ")
     (display 2)
     (newline))
```

Khi gọi sử dụng, không cần đưa vào tham đối thực sự :

```
((lambda ()
     (display 1)
     (display " < = ")
     (display 2)
     (newline)))
--> 1 < 2
```

Tuy nhiên, Scheme có dạng đặc biệt `begin` dùng để nối kết các s-biểu thức cũng cho ra kết quả tương tự :

```
(begin
     (display 1)
     (display " < = ")
     (display 2)
     (newline))
--> 1 < =2
```

IV.2.3. Định nghĩa hàm nhờ lambda

Trong nhiều trường hợp, người lập trình thường phải định nghĩa các hàm bổ trợ trung gian, hay định nghĩa hàm qua những hàm khác, mà tên gọi của chúng không nhất thiết phải quan tâm ghi nhớ. Chẳng hạn hàm `id` trong `sum-square`, hàm `inc` trong `increment`, v.v ... Để thuận tiện cho người lập trình, Scheme cho phép bỏ qua các tên này, bằng cách sử dụng dạng đặc biệt `lambla`. Chẳng hạn định nghĩa hàm :

```
(lambda (x) (+ x x))
```

cho phép gấp đôi đối số *x*. Khi gọi cần cung cấp tham đối :

```
((lambda (x) (+ x x)) 9)
--> 18
```

Khi sử dụng `define` để gán một biểu thức lambda cho một tên biến :

```
(define f (lambda (x y) (sqrt (+ x y))))
```

thì biến *f* có giá trị là một hàm, *f* có thể gọi tham số :

```
(f 3 13)
--> 4
```

Trong Scheme cú pháp định nghĩa hàm nhờ lambda như sau :

```
(define fname
    (lambda (x1 ... xn)
        body))
```

Dạng định nghĩa hàm dùng biểu thức lambda có lợi thế là không phân biệt một định nghĩa hàm với một định nghĩa giá trị nào đó của Scheme. Tuy nhiên, người ta có thể chuyển đổi từ dạng biểu thức lambda về dạng định nghĩa hàm trước đây. Ví dụ :

```
(define (member? s L)
    (if (null? L)
        #f
        (or (equal? s (car L))
            (member? s (cdr L)))))
```

có thể viết lại như sau

```
(define my-member?
    (lambda (s L)
         (if (null? L)
             #f
             (or (equal? s (car L))
                 ((define (my-member? s L)
```

Hàm sau đây cho phép chuyển đổi tự động dạng cũ thành dạng mới sử dụng lambda để định nghĩa một hàm bất kỳ (chú ý trong định nghĩa hàm có sử dụng quasiquote) :

```
(define (oldform->newform def-no-lambda)
    (let ((fname (caadr def-no-lambda))
          (paramlist (cdadr def-no-lambda))
          (Lbody (cddr def-no-lambda)))
      `(define ,fname (lambda ,paramlist ,@Lbody))))
```

Vận dụng hàm `oldform->newform`, ta có thể chuyển hàm `my-member?` trên đây về dạng định nghĩa nhờ lambda như sau :

```
(oldform->newform
    '(define (my-member? s L)
       (if (null? L)
           #f
           (or (equal? s (car L))
               (my-member? s (cdr L))))))
--> '(define my-member? (lambda (s l)
     (if (null? l) #f (or (equal? s (car l))
     (my-member? s (cdr l))))))
```

Hàm kết quả có thể được sử dụng như là các hàm thông thường. Chẳng hạn, ta xây dựng lại hàm `sum-integer` tính tổng các số nguyên giữa `a` và `b` cho ở ví dụ trước đây như sau :

```
(define (sum-integer a b)
   (sum (lambda (x) x) a b)) ; lời gọi (sum f x y)
(sum-integer 0 9)
--> 45
(define (increment x)
   (lambda (y) (+ x y)))
((lambda (x) (+ x 1)) 2)
--> 3
((lambda (x) (* x 5)) 10)
--> 50
```

Nhờ phép tính lambda, định nghĩa hàm trở nên gọn hơn về mặt cú pháp :

```
(define (double x) (+ x x))
```

là tương đương với định nghĩa sử dụng lambda :

```
(define double
   (lambde (x) (+ x x)))
(double 5)
--> 10
```

Ví dụ sau đây liệt kê các phần tử một danh sách :

```
(define (display-list L)
   (for-each (lambda (x)
       (display x) (display " ")) L))
(display-liste '(a b c d e))
--> a b c d e
(display-liste '(a (b ()) c (d) e))
--> a (b ()) c (d) e
```

Ta có thể sử dụng lambda để xây dựng các hàm tổ hợp một ngôi như sau :

```
; compose: (T2 → T3) × (T1 → T2) → (T1 → T3)
(define (compose f g)
   (lambda (x) (f (g x))))
```

hoặc định nghĩa cách khác như sau :

```
(define compose
   (lambda (f g)
      (lambda args
         (f (apply g args)))))
((compose sqrt *) 12 75)
--> 30
```

Dùng lambda để xây dựng hàm `incr` nhận một tham đối *x*, để trả về một hàm cho phép gia thêm một lượng vào *x* (xem ví dụ mục trước) :

```
; incr: (Number →Number) →(Number →Number)
(define (x)
   (lambda (y) (+ x y)))
((incr 12) 5)
--> 17
```

Ta định nghĩa hàm `incr2` cho phép tăng lên *2* một tham số :

```
(define incr2 (incr 2))
(define x 3)
(incr2 5)
--> 7   ; Kết quả không phải là 8.
```

IV.2.4. Kỹ thuật sử dụng phối hợp lambda

Dạng `let` là tương đương với dạng `lambda`. Chẳng hạn :

```
(let ((x 1) (y 2)) (+ x y))
--> 3
((lambda (x y) (+ x y)) 1 2)
--> 3
```

Như vậy :

```
(let ((x1 e1)          ⇔   ((lambda (x1 ... xk)
     ...                        body)
     (xk ek))             e1 ... ek)
  body)
```

Ta có thể định nghĩa một hàm sử dụng `let` phối hợp với lambda. Chẳng hạn, để tính biểu thức $\sqrt{a^2+b^2}$, ta có thể định nghĩa hàm bổ trợ x^2 nhờ lambda :

```
(define (pitagore a b)
  (let ((sqr (lambda (x) (* x x))))
     (sqrt (t(sqr a) (sqr b)))))
(pitagore 3 4)
--> 5.
```

Với lambda, ta cũng có thể sử dụng kỹ thuật «lời hứa tính toán». Chẳng hạn để mô phỏng dạng `if` tuỳ theo điều kiện để thực hiện các việc khác nhau, ta có thể định nghĩa hàm chờ như sau :

```
(define (my-if x y z)
  (if x (y) (z)))
(my-if (= 1 2) (lambda () #t) (lambda () #f))
--> #f
```

Ở đây cần sử dụng các cặp dấu ngoặc để gọi các lời hứa `y` và `z`.

```
(define (fact n)
  (my-if (= n 0)
     (lambda () 1)
     (lambda () (* n (fact (- n 1))))))
(fact 5)
--> 120
```

Định nghĩa hàm nhân :

```
(define (mult-n n)
   (lambda (x) (* n x)))
((mult-n 9) 7)
--> 63
```

Giả sử ta cần viết một thủ tục tạo sinh các số tự nhiên :

```
gennerator: 0 → Integer
```

cho phép liệt kê các số nguyên tại mỗi lời gọi :

```
(gennerator)
--> 0
(gennerator)
--> 1
(gennerator)
--> 2
```

v.v..., ta sử dụng đột biến trên các phần tử bộ đôi của danh sách như sau :

```
(define gennerator
```

```
   (let ((curr (list -1)))
      (lambda ()
        (set-car! curr (+ (car curr) 1))
        (car curr))))
(gennerator)
--> 0
(gennerator)
--> 1
(gennerator)
--> 2
(gennerator)
--> 3
```

Danh sách `(-1)` là giá trị gán cho `curr`, chỉ được tính duy nhất một lần trong thủ tục. Tuy nhiên nếu `curr` luôn luôn được tính lại mỗi lần gọi đến thủ tục theo cách định nghĩa như dưới đây thì sẽ gặp lỗi sai :

```
(define (gennerator)
  (define curr '(-1))
     (set-car! curr (+ (car curr) 1))
     (car curr))
(gennerator)
--> Error: exception (set-car! '(-1) 0)
```

Do danh sách `(-1)` được tạo ra một lần cho mọi trường hợp, khi định nghĩa thủ tục, lời gọi `set-car!` làm thay đổi các lệnh của thủ tục. Về mặt lý thuyết, điều này không đúng với cú pháp của ngôn ngữ Scheme.

```
gennerator
--> (lambda () (define curr '(-1)) ... )
(gennerator)
--> 0
gennerator
--> (lambda () (define curr '(0)) ... )
```

Những sai sót kiểu này thường xảy ra do vô ý khi lập trình với đột biến. Giả sử ta áp dụng hàm `set!` của Scheme để làm thay đổi của một biến đã được định nghĩa trước đó hiện đang tồn tại (hoặc được tạo ra bởi `define`, hoặc là tham đối của một hàm khác). Hàm `set!` có cú pháp như sau :

(***set! v s***)

Khi thực hiện `set!`, Scheme thay thế giá trị cũ của biến `v` bởi `s`. Ví dụ :

```
(define truc 1)
truc
--> 1
(set! truc 9999)
truc
--> 9999
```

Chú ý rằng hàm `set!` không trả về kết quả. Định nghĩa sau đây không đúng vì biến `toto` chưa có.

```
(set! toto 3)
--> ERROR
```

Sử dụng hàm `set!` trên đây để định nghĩa thủ tục tạo sinh các số tự nhiên, ta gặp lỗi sai, luôn luôn cho kết quả 0 :

```
(define (gennerator_e1)
        (define curr -1)
        (set! curr (+ curr 1))
     curr)
(gennerator_e1)
--> 0
(gennerator_e1)
--> 0
```

Định nghĩa sau đây cũng sai, luôn luôn cho kết quả 0 :

```
(define (gennerator_e2)
   (define curr (list -1))
   (set-car! curr (+ (car curr) 1))
        (car curr))
(gennerator_e2)
--> 0
(gennerator_e2)
--> 0
```

IV.2.5. Định nghĩa hàm nhờ tích luỹ kết quả

Người ta thường gặp nhiều cấu trúc lập trình giống nhau nhưng áp dụng cho nhiều hàm khác nhau. Sau đây, ta sẽ xét một ví dụ áp dụng kỹ thuật tích luỹ kết quả nhờ hàm `list-it`. Ta định nghĩa lần lượt 4 hàm thực hiện các việc sau :

1. Tính tổng giá trị của một hàm áp dụng cho các phần tử danh sách

```
(define (sum h L)
    (if (null? L )
        0
        (+ (h (car L )) (sum h (cdr L)))))
```

2. Tính tích giá trị của một hàm áp dụng cho các phần tử danh sách

```
(define (product h L)
    (if (null? L)
        1
        (* (h (car L )) (product h (cdr L)))))
```

3. Định nghĩa lại hàm `append` ghép hai danh sách

```
(define (myappend L1 L2)
    (if (null? L1)
       L2
       (cons (car L1) (myappend (cdr L1) L2))))
```

4. Định nghĩa lại hàm `map` cho hàm một biến h

```
(define (mymap h L)
```

```
    (if (null? L )
        '()
        (cons (h (car L)) (mymap h (cdr L )))))
```

Giả sử danh sách $L = (x_0, x_1, \ldots, x_n)$, ta có cấu trúc chung của các hàm trên như sau :

```
(sum h L)
     = (+ (h x0) (+ (h x1) (+ ... (+ h xn) 0) ... )))
(product h L)
     = (* (h x0) (* (h x1) (* ... (* h xn) 0) ... )))
(myappend L M)
     = (cons x0 (cons x1 (... (cons xn L2) ... )))
(mymap h L)
     = (cons (h x0 )(cons (h x1) (...(cons (h xn) '()) ...)))
```

Bằng cách sử dụng một hàm *f* có đối số thứ nhất là danh sách *L* và đối số thứ hai là kết quả tích luỹ liên tiếp, giá trị cuối cùng của hàm sẽ là :

```
(f x0 (f x1 (... (f xn R) ...)))
```

với *R* là giá trị đầu.

Từ đó ta đây dựng hàm `list-it` có tính tổng quát như sau :

```
(define (list-it f L R)
    (if (null? L )
        R
        (f (car L) (list-it f (cdr L) R))))
```

Hàm `list-it` thực hiện hàm *f* lần lượt cho các phần tử của danh sách *L*, kết quả được tích luỹ bắt đầu từ *R*. Sử dụng hàm `list-it` này, ta có thể gọi để thực hiện tính toán *L* hệt công việc của bốn hàm trên nhưng được viết lại như sau (dòng tiếp theo là ví dụ áp dụng) :

```
; Hàm (sum h L )
(list-it (lambda (x y)
     (+ (h x) y)) L 0)
(list-it (lambda (x y) (+ (sqrt x) y)) '(1 2 3 4 5) 0)
--> 8.38233
; Hàm (product h L)
(list-it (lambda (x y)
     (* (h x) y)) L 1)
(list-it (lambda (x y) (* (sqrt x) y)) '(1 2 3 4 5) 1)
--> 10.9545
; Hàm (myappend L1 L2)
(list-it cons L1 L2)
(list-it cons '(a b c) '(1 2))
--> '(a b c 1 2)
; Hàm (mymap h L)
(list-it (lambda (x y)
     (cons (h x ) y)) L '())
(list-it (lambda (x y)
     (cons (sqrt x ) y)) '(1 2 3 4 5) '())
```

```
--> '(1 . 1.41421 1.73205 2 . 2.23607)
(map sqrt '(1 2 3 4 5))
--> '(1 . 1.41421 1.73205 2 . 2.23607)
```

Chú ý khi áp dụng, cần cung cấp tham đối thực sự cho các tham đối là hàm *h*.

5. Định nghĩa các hàm `fold`

Trong chương 1, ta đã định nghĩa hàm `foldr` «bao qua phải» dùng để tính toán tích luỹ kết quả trên các phần tử của một danh sách. Sau đây ta định nghĩa trong Scheme hai hàm `fold` là `foldl` (left) và `foldr` (right) cùng nhận vào một hàm *f* hai tham đối : một phần tử xuất phát *a* và một danh sách *L*, để trả về kết quả là áp dụng liên tiếp (luỹ kế) hàm *f* cho *a* và với mọi phần tử của *L*. Hai hàm khác nhau ở chỗ hàm `foldl` lấy lần lượt các phần tử của *L* từ trái qua phải, còn hàm `foldl` lấy lần lượt các phần tử của *L* từ phải qua trái :

```
(define (foldl f a L)
   (if (null? L)
        a
       (foldl f (f a (car L)) (cdr L))))

(define (foldr f a L)
   (if (null? L)
        a
       (f (car L) (foldr f a (cdr L)))))

(foldl cons 0 '(1 2 3 4 5))
--> '(((((0 . 1) . 2) . 3) . 4) . 5)

(foldr cons 0 '(1 2 3 4 5))
--> '(1 2 3 4 5 . 0)
```

IV.2.6. Tham đối hoá từng phần

Như đã trình bày ở chương 1 về nguyên lý lập trình hàm, kỹ thuật tham đối hóa từng phần (currying) cho phép dùng biến để truy cập đến các hàm có số tham biến bất kỳ $f(x_1, \ldots, x_n)$.

Chẳng hạn, hàm hai biến $f(x, y)$ được xem là hàm một biến x trả về giá trị là hàm y :

$$x \rightarrow (y \rightarrow f(x, y))$$

Giả sử xét hàm `max` tìm số lớn nhất trong thư viện Scheme, với n=2 :

```
(max (* 2 5) 10)
--> 10
```

Sử dụng kỹ thuật Currying, ta viết :

```
(define (curry2 f)
    (lambda (x) (lambda (y) (f x y))))

(((curry2 max) (* 2 5)) 10)
--> 10
```

Với n=3, ta cũng xây dựng tương tự :

```
(define (curry3 f)
    (lambda (x)
        (lambda (y)
            (lambda (z) (f x y z)))))
```

```
((((curry3 max) (* 2 5)) 10) (+ 2 6))
--> 10
```

Từ đó ta có thể xây dựng cho hàm *n* đối bất kỳ. Ưu điểm của kỹ thuật Currying là có thể đặc tả một hàm ngay khi mới biết giá trị của tham số thứ nhất, hoặc các tham số đầu tiên cho các hàm có nhiều hơn hai tham đối.

Đối với hàm một biến, ưu điểm của kỹ thuật Currying là người ta có thể tổ hợp tuỳ ý các hàm mà không quan tâm đến các tham đối của chúng.

IV.2.7. Định nghĩa đệ quy cục bộ

Trong chương trước, ta đã làm quen với khái niệm hàm bổ trợ để định nghĩa các hàm Scheme. Hàm bổ trợ có thể nằm ngay trong hàm cần định nghĩa, được gọi là hàm cục bộ. Người ta có thể sử dụng dạng hàm `letrec` trong thư viện Scheme để định nghĩa đệ quy một hàm cục bộ. Chẳng hạn ta cần xây dựng một danh sách các số tự nhiên *0..n* với *n* cho trước như sau :

```
; iota : number -> list(number)
; hoặc iota : n -> (0 1 2 ... n)
(define (iota n)
    (if (zero? n )
        '(0)
    (append (iota (- n 1)) (list n))))
(iota 10)
--> '(0 1 2 3 4 5 6 7 8 9 10)
```

Định nghĩa trên đây đúng đắn, tuy nhiên việc sử dụng hàm ghép danh sách `append` làm tốn bộ nhớ, `do` luôn luôn phải thêm các phần tử mới vào cuối một danh sách. Vì vậy ý tưởng cải biên là làm ngược lại vấn đề : xây dựng hàm cho phép nhận một số *m* để trả về danh sách các số tự nhiên từ *m* đến *n* là (*m m+1 ....n*). Thay vì sử dụng hàm ghép danh sách, ta sử dụng `cons` để xây dựng hàm bổ trợ như sau :

```
(define (m-to-n m n)
    (if (< n m)
        '()
        (cons m (m-to-n (+ m 1) n))))
(m-to-n 3 10)
--> '(3 4 5 6 7 8 10)
```

Do hàm `m-to-n` không dùng ở đâu khác, nên cần đặt bên trong hàm `iota`. Sau khi định nghĩa, cần gọi với tham đối *m*=0 :

```
(define (iota n)
   (define (m-to-n m n)
     (if (< n m)
        '()
        (cons m (m-to-n (+ m 1) n))))
   (m-to-n 0 n))
(iota 10)
--> '(0 1 2 3 4 5 6 7 8 9 10)
```

Giả sử thay vì định nghĩa hàm cục bộ `m-to-n`, ta sử dụng dạng `let` để tạo ra lần lượt các số tự nhiên kể từ *m.* Tuy nhiên không cần dùng đến tham biến *n* vì *n* đã được cấp bởi hàm `iota`:

```
(define (iota n)
    (let ((m-to-n (lambda (m)
            (if (< m n)
                '()
                (cons m (m-to-n (+ m 1)))))))
        (m-to-n 0)))
(iota 10)
--> '()
```

Ta thấy kết quả sai vì lời gọi `m-to-n` trong hàm chỉ dẫn đến thực hiện `let` một lần mà không thực hiện gọi đệ quy. Để khắc phục, ta sử dụng dạng `letrec`, là dạng `let` đặc biệt để tạo ra lời gọi đệ quy. Cú pháp của `letrec` giống hệt `let` cũng gồm phần liên kết và phần thân :

(***letrec*** ((***v*₁ *e*₁**) ... (***vₖ eₙ***)) ***s***)

Các biến ***vᵢ***, ***i=1..N***, được gán giá trị ***eᵢ*** để sau đó thực hiện phần thân ***s*** là một biểu thức nào đó. Tuy nhiên mỗi phép gán biến có thể «nhìn thấy» lẫn nhau, nghĩa là khi tính biểu thức ***eᵢ*** thì có thể sử dụng các biến ***vᵢ*** với ***i***, ***j*** tuỳ ý. Nghĩa là không giống hoàn toàn `let`, các biến cục bộ ***v*₁**, ... , ***vₙ*** đều được nhìn thấy trong tất cả các biểu thức ***e*₁**, ... , ***eₖ***. Tuy nhiên, cần chú ý rằng mỗi biểu thức ***eⱼ*** được tính mà không cần tính giá trị của biến ***vⱼ***, khi ***eⱼ*** là một biểu thức lambda.

Nhờ ngữ nghĩa này, dạng `letrec` thường được sử dụng để định nghĩa các thủ tục *đệ quy tương hỗ* (mutually recursive).. Chẳng hạn, các vị từ `odd?` và `even?` là trường hợp điển hình cho các hàm thừa nhận định nghĩa đệ quy tương hỗ. Sau đây là ví dụ sử dụng `letrec` để định nghĩa tương hỗ hai thủ tục cục bộ kiểm tra một số nguyên là chẵn (even) hay lẻ (odd) mà không sử dụng hai hàm thư viện của Scheme là `even?` và `odd?` :

```
(letrec
      ((local-even? (lambda (n)
          (if (= n 0)
             #t
             (local-odd? (- n 1)))))
      (local-odd? (lambda (n)
          (if (= n 0)
             #f
             (local-even? (- n 1))))))
(list (local-even? 27) (local-odd? 27)))
--> '(#f #t)
```

Bây giờ hàm `iota` được định nghĩa lại như sau :

```
(define (iota n)
     (letrec
          ((m-to-n (lambda (m)
              (if (< n m)
                  '()
```

```
                    (cons m (m-to-n (+ m 1 )))))))
          (m-to-n 0)))
(iota 10)
--> '(0 1 2 3 4 5 6 7 8 9 10)
```

Sử dụng phối hợp dạng `letrec` và lambda như sau :

(*lambda* (*x*$_1$ ... *x*$_N$)
(*define* *f*$_1$ *e*$_1$)
...
(*define* *f*$_N$ *e*$_N$)
***s*)**

⇔

(*lambda* (*x*$_1$ ... *x*$_N$)
(*letrec* (*f*$_1$ *e*$_1$)
...
(*f*$_N$ *e*$_N$))
***s*))**

IV.3 Xử lý trên các hàm

IV.3.1. Xây dựng các phép lặp

Trong mục trước, ta đã định nghĩa hàm `list-it` sử dụng kỹ thuật tích luỹ kết quả để tạo ra một cấu trúc lập trình giống nhau áp dụng cho nhiều hàm khác nhau. Sau đây, ta sẽ xây dựng các hàm lặp mới là `append-map`, `map-select`, `every` và `some` để mở rộng thư viện các hàm của Scheme (vốn chỉ có hai thủ tục lặp có sẵn là `map` và `for-each`).

1. Hàm append-map

Khi một hàm *f* nhận tham đối là các phần tử của một danh sách để trả về các giá trị là danh sách, người ta cần nối ghép (concatenation) các danh sách này. Ta xây dựng hàm `append-map` như sau :

```
(define (append-map f L)
     (apply append (map f L)))
```

Áp dụng hàm `append-map`, ta có thể xây dựng hàm `flatting` để làm phẳng («cào bằng») một danh sách phù hợp khác rỗng theo nguyên tắc : ghép kết quả làm phẳng các phần tử của danh sách ở mức thứ nhất, kết quả làm phẳng của nguyên tử là danh sách (thu gọn về nguyên tử) :

```
(define (flatting s)
    (if (list? s)
        (append-map flatting s)
        (list s)))
(flatting '(a (b c) ((d (e))) "yes" ()))
--> '(a b c d e "yes")
(flatting 'a)
--> '(a)
(flatting 10)
--> '(10)
```

```
(flatting '())
--> '()
```

2. Hàm map-select

Nhiều khi, người ta chỉ muốn áp dụng hàm `map` cho một số phần tử của một danh sách thoả mãn một vị từ `p?` nào đó mà thôi, nghĩa là sử dụng `map` có lựa chọn. Ta xây dựng hàm `map-select` với ý tưởng như sau : sử dụng `append-map` và gán giá trị `'()` cho các phần tử không thoả mãn vị từ, và do vậy các giá trị rỗng này không đưa vào danh sách kết quả cuối cùng khi hàm trả về.

```
(define (map-select f L p?)
     (append-map
        (lambda (x)
             (if (p? x)
                  (list (f x))
                  '()))
        L))
(map-select (lambda (x)(/ 1 x))
        '(a 3 0 5 7 9)
        (lambda (x)
            (and (number? x) (not (zero? x)))))
--> '(1/3 1/5 1/7 1/9)
(map-select sqrt '(1 2 3 4 5) odd?)
--> '(1. 1.73205 2.23607)
```

3. Các hàm every và some

Khi các đối số của phép hội lôgic `and` là các giá trị của hàm f nào đó, ta có thể định nghĩa hàm `every` để mở rộng `and` như sau :

(***every f '***(e_1 ... e_N)) **=** (***and*** (***f*** e_1) ... (***f*** e_N))

Tuy nhiên ta không thể định nghĩa `every` một cách trực giác là áp dụng phép `and` cho danh sách các giá trị của f :

```
(define (every f L)
     (apply and (map f L)))
```

Bởi vì hàm `apply` không nhận `and` làm tham đối. Trong Scheme, `and` không phải là hàm mà là một dạng đặc biệt. Ta định nghĩa `every` theo cách đệ quy truyền thống như sau :

```
(define (every f L)
    (if (null? L)
        #t
        (and (f (car L))
             (every f (cdr L)))))
(every even? '(0 2 4 6 8))
--> #t
(every number? '(1 3 a 5))
--> #f
```

Một cách tương tự, ta xây dựng hàm `some` để mở rộng phép tuyển lôgic `or` bằng cách thay thế `and` bởi `or` có dạng:

(***some f '***(***e*** $_1$ ... ***e*** $_N$)) **=** (***or*** (***f e*** $_1$) ... (***f e*** $_N$))

Hàm `some` như sau :

```
(define (some f L)
    (if (null? L)
        #f
        (or (f (car L))
             (some f (cdr L)))))
(some number? '(1 3 a 5))
--> #t
(some odd? '(0 2 4 6 8))
--> #f
```

IV.3.2. Trao đổi thông điệp giữa các hàm

Sau khi định nghĩa các hàm và cài đặt các kiểu dữ liệu, người sử dụng có thể thao tác trực tiếp trên dữ liệu bằng cách sử dụng kỹ thuật lập trình *truyền thông điệp* (message transmission). Thay vì xem dữ liệu như là một giá trị kiểu đơn vị (một số nguyên, một bộ đôi, một danh sách, *v.v* ...), ta có thể xem dữ liệu như là một hàm nhận các thông điệp và thực hiện tính toán tuỳ thuộc vào thông điệp đã nhận. Người ta gọi đây là *hàm bẻ ghi* (switch function). Những hàm thấy được từ bên ngoài chỉ còn là một lời gọi với một thông điệp đặc biệt. Ta xét lại ví dụ về xử lý các số hữu tỷ ở chương trước. Ta đã khai báo các hàm tạo số hữu tỷ, xác định tử số và mẫu số như sau :

functions

```
create-rat : integer × integer → rational
numer :      rational     → integer
denom :      rational     → integer
```

Bây giờ ta định nghĩa lại hàm `create-rat` có dạng như sau :

```
create-rat : (integer × integer) → (symbol → Integer)
```

Hàm `create-rat` sẽ tạo ra một hàm nhận vào một thông điệp (kiểu ký hiệu) để trả về một số nguyên, tuỳ theo nội dung thông điệp cho biết đó là tử số hay mẫu số :

```
(define (create-rat x y) ; precondition: y ≠ 0
   (define g (gcd x y))      ; gcd là ước số chung lớn nhất của x, y
   (define N (quotient x g))
   (define D (quotient y g))
     (lambda (message)
        (cond
           ((eq? message 'numerator) N)
           ((eq? message 'denominator) D)
           (else (error "unknown message" message)))))
((create-rat 9 15) 'numerator)
--> 3
```

```
((create-rat 9 15) 'denominator)
--> 5
((create-rat 9 15) 'denom)
--> *** Error: "unknown message" (denom)
(create-rat 9 15)
--> *** Error: '#{Procedure 6700 (unnamed in create-rat)}
```

Ta tiếp tục định nghĩa hai hàm `numer` và `denom` sẽ gọi đến hàm `create-rat` :

```
; numer : (integer × integer) → integer
(define (numer R)
   (R 'numerator))
; denom : (integer × integer) → integer
(define (denom R)
   (R 'denominator))
(numer (create-rat 9 15))
--> 3
(denom (create-rat 9 15))
--> 5
```

Với cách xây dựng các hàm xử lý các số hữu tỷ trên đây, người sử dụng chỉ có thể gọi đến một hàm nhờ một thông điệp `message` : lỗi sai `do` một lời gọi hàm nào đó, vô tình hay cố ý, đều không thể xảy ra. Trong trường hợp một thông điệp cần các tham đối bổ sung, hàm bẻ ghi sẽ dẫn về một hàm cần phải thực hiện với những tham đối này.

Nếu cần thêm một bộ kiểm tra `=rat`, chỉ cần thêm mệnh đề sau đây vào điều kiện kiểm tra `message` của hàm `create-rat` (trước `else`) trên đây :

```
((eq? message '=rational)
   (lambda(R) (= (* N (denom R)) (* D (numer R)))))
```

Lúc này, ta có thể viết hàm `=rat` trả về một hàm một tham đối là số hữu tỷ cần so sánh như sau :

```
(define (=rat R1 R2)
   ((R1 '=rational) R2)) ; áp dụng hàm trả về với R2
(=rat (create-rat 1 3) (create-rat 1 3))
--> #t
(=rat (create-rat 2 3) (create-rat 3 2))
--> #f
```

Kỹ thuật lập trình xem dữ liệu như là một hàm nhận các thông điệp để thực hiện tính toán thường khó sử dụng, đòi hỏi người lập trình phải quản lý tốt ý đồ xử lý theo nội dung thông điệp. Tất cả những gì là dữ liệu đều có thể tạo ra hàm nhờ lambda nhưng rất khó nhận biết được những thông điệp nào sẽ thoả mãn hàm. Chẳng hạn sau đây là một ví dụ sử dụng bộ đôi nhưng thay đổi hai hàm `car` là `cdr` :

```
(define (cons x y)
   (lambda (m)
      (cond
         ((eq? m 'car) x)
         ((eq? m 'cdr) y)
         (else (error "unknown message")))))
(define (car D) (D 'car))
```

```
(define (cdr D) (D 'cdr))
(cdr (cons 'tom 'jerry))
--> 'jerry
(car (cons ''tom ''jerry))
--> 'tom
(car '(1 2 3))
--> ERROR: attempt to call a non-procedure
```

IV.3.3. Tổ hợp các hàm

Từ khái niệm hàm bậc cao, ta có thể định nghĩa các *hàm tổ hợp* (function composition) như sau :

; $g \circ f : (T_1 \rightarrow T_2) \rightarrow T_3$

Hoặc tổ hợp trực tiếp ngay trong tên hàm :

```
(define (compofg x ) (g (f x))
```

chẳng hạn :

```
(define (cacddddr L)
     (car (cddddr L)))
(cacddddr '(a b c d e f))
-->'e
```

Hoặc sử dụng lambda để tạo ra dạng một «lời hứa» :

```
(define (compos g f)
     (lambda (x) (g (f x))))
```

chẳng hạn :

```
((compos car cddddr) '(a b c d e f g ))
--> 'e
(define fifth
     (compos car cddddr))
(fifth '(a b c d e f g ))
--> 'e
```

Nhờ khái niệm bậc của hàm, nhiều sai sót có thể được phát hiện. Giả sử ta xét các hàm `map` và `cdr` làm việc với dữ liệu kiểu danh sách :

$map : (T_1 \rightarrow T_2) \times List(T_1) \rightarrow List(T_2)$

$cdr : List(T) \rightarrow List(T)$

Nếu áp dụng hàm `map` là hàm bậc 2 có `cdr` làm tham đối sẽ gây ra lỗi :

```
(map cdr (list 1 2 3))
--> ERROR: cdr: Wrong type in arg1 1
```

Trong định nghĩa hàm `cdr`, danh sách có kiểu `List(T)`, nên danh sách `(1 2 3)` phải có kiểu `List(Integer)`. Như vậy, để áp dụng được hàm `map`, cần phải có :

`T1 = List(T) = Integer`

nhưng điều này là không thể xảy ra. Tương tự, từ định nghĩa hàm sau đây :

```
f : (Number → T1) → T2
```

ta định nghĩa hàm :

```
(define (f g) (g 2))
```

thì lời gọi sau đây là sai :

```
(f f)
--> ERROR: Wrong type to apply: 2
```

Tuy nhiên những lời gọi sau đây lại cho kết quả đúng :

```
(f list)
--> '(2)      ; tạo danh sách một phần tử
(f sqrt)
--> 1.41421   ; tính căn bậc hai một số nguyên
```

Ví dụ sau là một định nghĩa hàm sử dụng một hàm khác làm tham đối nhưng thực tế, tham đối này chẳng đóng vai trò gì trong việc tính toán ở thân hàm, chỉ có mặt cho «phải phép» xét về mặt cú pháp :

```
(define (f g n)
   (if (= n 0)
       1
       (* n (g g (- n 1)))))
(define (mystery n) (f f n))
(mystery 5)
--> 120
(f f 5)
--> 120
```

Người đọc có thể nhìn nhận ra ngay `mystery` là hàm tính giai thừa ! Xét về phép toán, định nghĩa trên tỏ ra hợp lý (tính đúng giai thừa). Tuy nhiên, mọi ngôn ngữ có định kiểu mạnh sẽ từ chối định nghĩa này, vì không thể xác định được bậc của hàm *f* và vai trò của nó trong thân hàm. Một cách tương tự, người đọc có nhận xét gì khi thay đổi lại định nghĩa hàm tính giai thừa như sau :

```
(define (fac g h n)
   (if (= n 0)
       1
       (* n (h h h (- n 1)))))
```

Có thể thay đổi dòng cuối cùng thành `(g g g ... )` ?

IV.3.4. Các hàm có số lượng tham đối bất kỳ

Khi gọi thực hiện một hàm, bộ diễn dịch Scheme đặt tương ứng các tham đối hình thức với các tham đối thực sự. Sau đây là các dạng định nghĩa hàm có nhiều tham đối tuỳ ý và dạng sử dụng phép tính lambda tương đương.

Dạng định nghĩa 1 :

```
(define (f x y z) ... )
```

tương đương với :

```
(define f (lambda (x y z) ... )
```

Ví dụ với lời gọi :

```
(f 1 (+ 1 2) 4)
```

các tham đối hình thức lần lượt được nhận giá trị : *x=1, y=3, z=4*.

Dạng định nghĩa 2 :

```
(define (g . L) ... )
```

(chú ý *g* là tên hàm, có dấu cách trước và sau dấu chấm) tương đương với :

```
(define g (lambda L ... ))
```

Ví dụ với lời gọi :

```
(g 1 2 3 4 5)
```

tham đối hình thức là danh sách *L* được lấy giá trị *L = '(1 2 3 4 5)*.

Dạng định nghĩa 3 :

```
(define (h x y . z) ... )
```

tương đương với :

```
(define h (lambda (x y . z) ... )
```

Ví dụ với lời gọi :

```
(h 1 2 3 4)
```

các tham đối hình thức được lấy giá trị như sau : `((x y . z)` được so sánh với `(1 2 3 4)`, từ đó, *x=1, y=2, z=(3 4)*

Trong 3 dạng định nghĩa trên đây, hàm *f* thừa nhận đúng 3 tham đối, hàm *g* có số lượng tham đối tuỳ ý, hàm *h* phải có tối thiểu 2 tham đối. Ta có thể dễ dàng định nghĩa hàm `list` theo dạng 2 :

```
(define (list . L) L)
(list 1 2 3 4 5 6 7 8 9 0)
--> '(1 2 3 4 5 7 8 9 0)
```

Ta có thể mở rộng một hàm hai tham đối để định nghĩa thành hàm có nhiều tham đối. Ví dụ, từ hàm cộng + là phép toán hai ngôi, ta định nghĩa hàm `add` để cộng dồn các phần tử của tham đối là một dãy số nguyên tuỳ ý :

```
(define (add . L)       ; L : List(Number)
   (define (add-bis L R)
      (if (null? L)
           R
           (add-bis (cdr L) (+ (car L) R))))
   (add-bis L 0))
(add 2 3 4 5)
--> 14
```

Áp dụng hàm `apply`, hàm `add` có thể định nghĩa theo cách khác như sau :

```
(define (add . L)
   (if (null? L)
        0
        (+ (car L) (apply add (cdr L )))))
(add 1 2 3 4 5)
--> 15
(define L '(1 2 3 4 5))
```

```
(apply add L)
--> 15
```

Một cách tổng quát, các dạng định nghĩa 2 và 3 có thể viết dưới dạng :

```
(define (f v0 v1 . . vn- 1 .vN) s)
```

tương đương với dạng lambda :

```
(define (f)
    (lambda (L) s))
```

trong đó thân hàm **s** là một biểu thức, **L** là một danh sách các đối tuỳ ý.

IV.4 Một số ví dụ

IV.4.1. Phương pháp xấp xỉ liên tiếp

Nhiều khi, ta cần giải phương trình có dạng :

$$x = f(x) \qquad (1)$$

trong đó f là một hàm có biến số x. Chẳng hạn để tìm giá trị x từ phương trình :

$$x^2 = 2$$

ta có thể viết lại dưới dạng phương trình (1) :

$$x = (x^2 + 2)/2x \qquad (2)$$

Để giải phương trình (2), ta sử dụng phương pháp xấp xỉ liên tiếp như sau :

$$x_0 = c$$
$$x_{n+1} = (x_n^2 + 2)/2x_n$$

trong đó c là một giá trị nào đó để khởi động quá trình lặp. Nếu viết lại các phương trình trên theo dạng (1) và đặt hằng $c = f(x_o)$, ta nhận được :

$$x_0 = f(x_o) \qquad (3)$$
$$x_{n+1} = f(x_n)$$

Như vậy, lời giải tìm nghiệm x từ phương trình (1) là tìm giới hạn của dãy $\{ x_n \}$ theo (3). Trong toán học, người ta phải chứng minh tính hội tụ của dãy để nhận được $x = f(x)$. Áp dụng phương pháp xấp xỉ liên tiếp, ta có thể «giải» hay giải thích cách dùng phép đệ quy tính $n!$:

```
fac(n) = 1, với n ≤ 0
fac(n) = n * fac(n-1)
```

Từ cách tính lặp các giá trị $\{ x_n \}$, ta xây dựng hàm xấp xỉ liên tiếp như sau :

```
(define (n-iter f x x0)
   (if (zero? n)
       x0
       (f (n-iter f (- n 1) x0))))
```

Ví dụ 1 : tính $\sqrt{2}$ nhờ phương pháp xấp xỉ liên tiếp $x = \frac{x}{2} + \frac{1}{x}$ với $x_0 = 1$:

```
(n-iter (lambda (x) (+ (/ x 2)
              (/ 1. 0 x))) 10 1)
--> 1. 41421
```

Ví dụ 2 : Tính gần đúng $x = sin\ (2x)$:

```
(n-inter (lambda (x) (sin (* 2 x))) 10 1)
--> 0.948362
```

IV.4.2. Tạo thủ tục định dạng

Sau đây là một ví dụ tự tạo một thủ tục định dạng `format` các kết quả đưa ra. Thủ tục hoạt động tương tự Common Lisp có dạng như sau :

```
(define (format format-str . L-args)
```

trong đó, `format-str` là chuỗi định dạng, còn `L-args` là các biểu thức tham đối (số lượng tuỳ ý) được đưa ra theo cách quy ước trong chuỗi định dạng tương ứng.

Chuỗi định dạng đưa ra mọi ký tự có mặt trong đó, trừ ra các ký tự có đặt trước một ký tự đặc biệt **^** được quy ước như sau :

`^s` hoặc `^S`	chỉ vị trí để `write` đưa ra giá trị tham đối tương ứng
`^a` hoặc `^A`	chỉ vị trí để `display` đưa ra giá trị tham đối tương ứng
`^%`	nhảy qua dòng mới
`^^`	in ký tự ^

Ví dụ :

```
(format "Display ^^ with the format:^% x = ^s" (+ 1 2))
--> Display ^ with the format:
    x = 3
(format "sin ^s = ^s^%" (/ 3.141592 2) (sin (/ 3.141592 4)))
--> sin 1.5708 = 0.707107
```

Thủ tục `format-str` không trả về giá trị mà tuỳ theo nội dung chuỗi định dạng để đưa ra kết quả. Cách hoạt động đơn giản như sau : thủ tục duyệt chuỗi định dạng và đưa ra mọi ký tự không đặt trước một ký tự đặc biệt **^**. Khi gặp **^**, thủ tục sẽ xử lý lần lượt từng trường hợp bởi `case`. Tham số `i` trong hàm bổ trợ `format-to` chỉ định vị trí của ký tự hiện hành.

```
(define (format str . L)
  (let ((len (string-length str)))
    (letrec ((format-to (lambda (i L)
      (if (= i len)
          (newline) ; 'indefinite
          (let ((c (string-ref str i)))
            (if (eq? c #\^)
              (case (string-ref str (+ 1 i))
                ((#\a #\A)
                  (display (car L))
                  (format-to (+ i 2) (cdr L)))
                ((#\s #\S)
                  (write (car L))
                  (format-to (+ i 2) (cdr L)))
                ((#\%)
                  (newline) (format-to (+ i 2) L))
                ((#\^)
                  (display #\^)
                  (format-to (+ i 2) L))
```

```
                (else (display "unknown character")))
              (begin (display c)
                  (format-to (+ i 1) L))))))))
        (format-to 0 L))))
```

IV.4.3. Xử lý đa thức

IV.4.3.1. Định nghĩa đa thức

Ta xét các đa thức (polynomial) biến x hệ số thực. Mỗi đa thức là tổng hữu hạn các đơn thức (monomial). Mỗi đơn thức có thể là 0 hoặc một biểu thức dạng :

ax^n

với a là hệ số thực và n là bậc của x, n nguyên dương và $a \neq 0$.

Khi $n = 0$, đơn thức ax^0 trở thành hằng a. Đơn thức 0 không có bậc, đôi khi người ta nói bậc của nó là -∞.

Ví dụ :

$9x^4 + 7x^5 + -10 + -7x^5 + 27x$

là một đa thức của x. Lúc này dấu cộng có vai trò phân cách các đơn thức. Ta chưa định nghĩa phép cộng trên đa thức, tuy nhiên một cách trực giác, ta có thể rút gọn đa thức thành :

$9x^4 + -10 + 27x$

Để xử lý các đa thức trong Scheme, trước hết ta cần đưa ra một cách biểu diễn đa thức thống nhất. Ta đã biết rằng phép cộng các đơn thức có cùng bậc theo nguyên tắc như sau :

$ax^n + bx^n = o$ *nếu* $a+b = 0$
$ax^n + bx^n = (a+b)x^n$ *nếu không*

Bằng cách cộng tất cả các đơn thức cùng bậc và sắp xếp một đa thức theo bậc giảm dần, ta nhận được cách biểu diễn chính tắc (canonical) cho các đa thức như sau :

$a_p x^p + ... + a_i x^i$

Đối với một đa thức khác 0, đơn thức có bậc cao nhất được gọi là đơn thức *trội* (dominant), và bậc của nó là bậc của đa thức, hệ số của nó là hệ số định hướng (director) hay hệ số trội. Ví dụ đa thức trên đây có bậc là 4 và được viết lại là :

$9x^4 + 27x + -10$

Người ta định nghĩa phép cộng hai đa thức :

$(a_p x^p + ... + a_i x^i) + (a_q x^q + ... + a_j x^j)$

bằng cách cộng các đơn thức cùng bậc và đặt các đơn thức dưới dạng chính tắc, ta nhận được đa thức tổng.

IV.4.3.2. Biểu diễn đa thức

Có hai phương pháp biểu diễn các đa thức trong Scheme :

1. **Biểu diễn đầy đủ** (full representation)

Theo phương pháp này, người ta tạo một danh sách gồm tất cả các hệ số (bằng 0 hoặc khác 0), bắt đầu từ hệ số trội. Ví dụ đa thức $9x^4 + 27x + -10$ được biểu diễn bởi danh sách :

```
'(9 0 0 27 -10)
```

2. **Biểu diễn hổng** (hollow representation)

Sử dụng một danh sách các đơn thức khác 0 theo bậc giảm dần, mỗi đơn thức được xác định bởi bậc và hệ số tương ứng. Nghĩa là mọi đa thức P, khác 0, đều có dạng :

$P = cx^d + Q$

với cx^d là đơn thức trội có bậc d và hệ số d, Q là đa thức còn lại có bậc thấp hơn.

Sau đây ta chọn xét phương pháp *biểu diễn hổng* các đa thức. Việc sử dụng nguyên tắc *phân tách một cách chính tắc một đa thức thành một đơn thức trội và một đa thức còn lại có bậc thấp hơn* cho phép định nghĩa đa thức như là một cấu trúc dữ liệu. Giả thiết từ lúc này trở đi, mọi đa thức đều có cùng biến x và các hàm xử lý lấy tên với tiếp đầu ngữ `poly`.

Trước tiên, ta tiến hành các thao tác xử lý đa thức mà chưa nêu ra cách biểu diễn hổng trong Scheme.Ta gọi `zero-poly` là đa thức 0, còn `cons-poly` là hàm tạo đa thức khác 0, có ba tham biến. Hai tham biến thứ nhất và thứ hai định nghĩa đơn thức trội gồm hệ số và bậc của nó, tham biến thứ ba chỉ định đa thức còn lại có bậc thấp hơn (là đa thức đã cho nhưng đã bỏ đi đơn thức có bậc cao nhất) :

```
poly = zero-poly
```

hoặc :

`poly = (cons-poly coeff degree poly)` với điều kiện `coeff` ≠ 0

Các hàm tiếp cận đến các thành phần đa thức là `degree-poly`, `coeff-domin`, `remain-poly` thõa mãn các quan hệ sau :

```
(coeff-domin (cons-poly coeff degree poly)) = coeff
(degree-poly (cons-poly coeff degree poly)) = degree
(remain-poly (cons-poly coeff degree poly)) = poly
```

Để phân biệt các đa thức 0, xây dựng vị từ `zero-poly?` thõa mãn quan hệ :

```
(zero-poly? zero-poly) = #t
(zero-poly? (cons-poly coeff degree poly)) = #f
```

Để duy trì quan điểm trừu tượng hoá vấn đề, tạm thời ta chưa nêu cụ thể cách xây dựng các hàm tạo mới đa thức và các hàm tiếp cận đến các thành phần của đa thức vừa trình bày trên đây mà tiếp tục sử dụng chúng trong các phép xử lý dưới đây.

IV.4.3.3. Xử lý đa thức

Cho trước đa thức $P = cx^d + Q$, các phép xử lý trên P bao gồm :

- Nhân đa thức với một hằng số $a*P$
- So sánh hai đa thức $P_1 ? P_2$
- Cộng hai đa thức $P_1 + P_2$
- Nhân hai đa thức $P_1 * P_2$

1. Nhân đa thức với một hằng số

Để nhân hằng số a với đa thức P, lấy a nhân với hệ số trội rồi sau đó lấy a nhân với đa thức còn lại :

```
(define (mult-scalar-poly a P)
    (cond ((zero? a) zero-poly) ; xử lý hệ số =0
          ((zero-poly? P) zero-poly) ; xử lý đa thức 0
          (else
```

```
            (let ((d (degree-poly))
                  (c (coeff-domin P))
                  (Q (remain-poly)))
              (cons-poly d (* c a)
                   (mult-scalar-poly a Q))))))
```

2. So sánh hai đa thức

So sánh hai đa thức bằng cách so sánh hai đơn thức trội, sau đó tiếp tục so sánh hai đa thức còn lại :

```
(define (equal-poly? P1 P2)
    (cond ; xử lý đa thức 0
          ((zero-poly? P1) (zero-poly? P2))
          ((zero-poly? P2) (zero-poly? P1))
          (else; xử lý đa thức ≠ 0
              (let ((d1 (degree-poly P1))
                    (d2 (degree-poly P2))
                    (c1 (coeff-domin P1))
                    (c2 (coeff-domin P2))
                    (Q1 (remain-poly P1))
                    (Q2 (remain-poly P2)))
                ; hoặc so sánh hai đa thức còn lại trước tiên theo bậc và hệ số
                (and (= d1 d2) (= c1 c2)
                     ; và so sánh hai đa thức còn lại của hai đa thức còn lại
                     (equal-poly? Q1 Q2))))))
```

3. Phép cộng đa thức

Khi cộng hai đa thức P_1 và P_2 cần phân biệt hai trường hợp :

- Nếu các đa thức P_1 và P_2 có cùng bậc, chỉ việc cộng các đơn thức trội và xem xét trường hợp chúng có triệt tiêu lẫn nhau không.
- Nếu các đa thức P_1 và P_2 không có cùng bậc, tìm đa thức có đơn thức bậc cao hơn, sau đó thực hiện phép cộng các đa thức còn lại.

```
(define (add-poly P1 P2)
    (cond
       ; xử lý các đa thức 0
       ((zero-poly? P1) P2)
       ((zero-poly? P2) P1)
       ; xử lý các đa thức ≠ 0
       (else
         (let ((d1 (degree-poly P1))
               (d2 (degree-poly P2))
               (c1 (coeff-domin P1))
               (c2 (coeff-domin P2))
               (Q1 (remain-poly P1))
               (Q2 (remain-poly P2)))
           (cond
              ((= d1 d2) ; hai đa thức có cùng bậc
                (let ((c (+ c1 c2)))
```

```
                (if (zero? c)
                   (add-poly Q1 Q2)
                   (cons-poly
                        d1
                        c
                        (add-poly Q1 Q2)))))
          ((< d1 d2) ; hai đa thức không có cùng bậc
            (cons-poly d2 c2 (add-poly P1 Q2)))
          (else
            (cons-poly d1 c1 (add-poly Q1 P2))))))))
```

4. Phép nhân hai đa thức

Giả sử :

$P_1 = (c_1x^{d_1} + Q_1)$ và $P_2 = (c_2x^{d_2} + Q_2)$

khi đó, kết quả phép nhân hai đa thức P_1 và P_2 là :

$$(c_1x^{d_1} + Q_1) * (c_2x^{d_2} + Q_2) = c_1 * c_2\, x^{(d_1 + d_2)} + (c_1x^{d_1} + Q_1) * Q_2 + Q_1 * (c_2x^{d_2} + Q_2)$$

Hàm `mul-poly` được xây dựng như sau :

```
(define (mul-poly P1 P2)
  (cond ((zero-poly? P1) zero-poly)
        ((zero-poly? P2) zero-poly)
        (else
          (let
              ((d1 (degree-poly P1))
               (d2 (degree-poly P2))
               (c1 (coeff-domin P1))
               (c2 (coeff-domin P2))
               (Q1 (remain-poly P1))
               (Q2 (remain-poly P2)))
            (if (zero-poly? Q1)
              ; Q1 = 0
              (cons-poly (+ d1 d2) (* c1 c2)
                   (mul-poly P1 Q2))
              ; Q1 ≠ 0
              (add-poly
                 (add-poly
                    (mul-poly
                      (cons-poly d1 c1 zero-poly) P2)
                    (mul-poly Q1 Q2))
                 (mul-poly Q1 P2)))))))
```

IV.4.3.4. Biểu diễn trong một đa thức

Bây giờ ta tìm cách biểu diễn hổng các đa thức trong Scheme : mỗi đa thức được biểu diễn bởi một danh sách các đơn thức khác 0 theo thứ tự bậc giảm dần. Ta chọn cách biểu diễn mỗi đơn thức bởi một bộ đôi như sau :

```
(degree . coefficient)
```

Như vậy mỗi đa thức được biểu diễn bởi một danh sách kết hợp alist. Chẳng hạn đa thức :

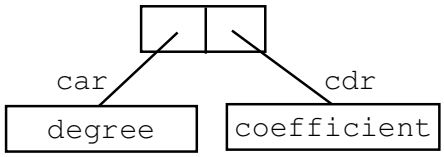
$9x^4 + 27x + -10$

được biểu diễn bởi alist :

```
((4. 3) (1. 20) (0. -10))
```

Sau đây ta xây dựng các hàm tạo mới đa thức và các hàm tiếp cận đến các thành phần của đa thức như sau :

```
(define (cons-poly degree coeff Q)
   (cond
     ((zero? coeff)
       (display
        "Error: the coefficient can't be zero !"))
     ((zero-poly? Q)
       (list (cons degree coeff)))
     (else
       (cons (cons degree coeff) Q))))
(define degree-poly caar)
(define coeff-domin cdar)
(define (remain-poly Q)
    (if (null? (cdr Q))
      zero-poly
      (cdr Q)))
```



Hình IV.2. Biểu diễn đơn thức bởi bộ đôi.

Đối với đa thức 0 ta có thể biểu diễn tùy ý, Chẳng hạn :

```
(define zero-poly 0)
(define (zero-poly? P)
    (and (number? P) (zero? P)))
```

IV.4.3.5. Đưa ra đa thức

Để nhìn thấy các đa thức trên màn hình, hay trên giấy in, ta có thể biểu diễn chúng nhờ các kí tự ASCII thông dụng. Chẳng hạn đơn thức ax^n sẽ được biểu diễn dạng `ax^n`.

Sau đây ta xây dựng hàm `print-poly` để đưa ra một đa thức. Hàm này đưa ra liên tiếp các đơn thức nhờ gọi đến hàm `print-mono`. Trong trường hợp bậc một đơn thức là 1 thì không in ra dạng `ax^1` mà in ra `x` (đảm bảo tính thẩm mỹ) :

```
(define (print-poly P)
   (if (zero-poly? P)
     zero-poly
     (let ((c (coeff-domin P))
           (d (degree-poly P))
           (Q (remain-poly P)))
       (print-mono d c)
   (if (not (zero-poly? Q))
    (begin (display "+")
```

```
        (print-poly Q))))))
(define (print-mono degree coeff)
(cond ((zero? degree) (display coeff))
    ((=1 degree) (display coeff) (display "x"))
    (else (display coeff)
        (display "x")
        (display "^")
        (display degree))))
(define P1
  (cons-poly 2 9 (cons-poly 1 27 (cons-poly 0 -10 0))))
P1
--> '((2 . 9) (1 . 27) (0 . -10))
(print-poly P1)
--> 9x^2+27x+-10
(add-poly P1 P1)
--> '((2 . 18) (1 . 54) (0 . -20))
(print-poly (add-poly P1 P1))
--> 18x^2+54x+-20
(print-poly (mul-poly P1 P1))
--> 81x^4+486x^3+1278x^2+-1350x+400
```

Những xử lý ký hiệu trên các đa thức mà ta vừa trình bày trên đây thường được ứng dụng trong các hệ thống *tính toán hình thức* (formal calculus).

IV.4.4. Thuật toán quay lui

Khi giải những bài toán tổ hợp (combinatorial problems) hay bài toán trò chơi đố chữ (puzzle) người ta thường không có thuật toán trực tiếp để tìm ra lời giải, mà thường áp dụng thuật toán «thử sai» (try and error). Ý tưởng của thuật toán là người ta phải thử liên tiếp các phương án, hoặc dẫn đến thành công và kết thúc, hoặc khi thất bại thì phải quay lui (backtracking) trở lại phương án đã lựa chọn trước đó để tiếp tục quá trình.

IV.4.4.1. Bài toán tám quân hậu

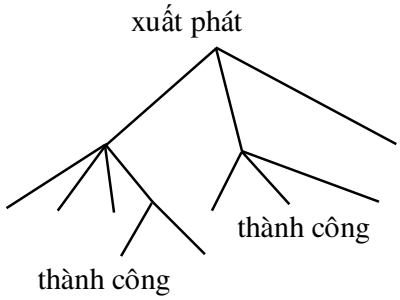
Một ví dụ cổ điển là bài toán tám quân hậu (đã xét ở chương 1). Yêu cầu đặt 8 quân hậu lên bàn cờ vua 8×8 ô lúc đầu không chứa quân nào sao cho các quân hậu không ăn được lẫn nhau. Lời giải được tiến hành dần dần như sau :

Giả sử gọi quân hậu là Q, đầu tiên đặt Q tại cột thứ nhất, sau đó, tìm cách đặt Q ở cột thứ hai sao cho không bị Q ở cột trước ăn và cứ thế tiếp tục cho các cột tiếp theo. Có thể tại một sự lựa chọn nào đó không cho phép đặt được Q tiếp theo (tại cột j). Khi đó, người ta phải xem lại sự lựa chọn cuối cùng trước sự thất bại đó (ở cột kế trước j-1) để bắt đầu lại. Nếu tất cả khả năng cho sự lựa chọn cuối cùng đều thất bại, người ta lại phải quay lui đến sự lựa chọn trước lựa chọn cuối cùng (ở cột j-2), v.v...

Kỹ thuật quay lui thường được minh hoạ bởi một cây mà mỗi nút trong là một trạng thái tìm kiếm lời giải và một đường đi từ gốc đến một lá (nút ngoài) nào đó có thể là một lời giải của bài toán đã cho.

Xuất phát từ nút gốc của cây, là trạng thái đầu, có thể tiếp cận đến các con của nó là các lựa chọn có thể để đạt đến trạng thái tiếp theo. Khi đi đến một nút cho biết thất bại, người ta

phải quay lên nút cha và bắt đầu với nút con chưa được tiếp cận. Nếu như đã được tiếp cận hết các con mà mà vẫn thất bại, người ta lại quay lên nút tổ tiên (cha của nút cha)v.v... Quá trình quay lui đã được giải quyết trong phương pháp *tìm kiếm theo chiều sâu trước* (depth-first-search algorithm).



Hình IV.3. Tìm lời giải trên cây trạng thái.

Ý tưởng của thuật toán như sau :

Gọi thuật toán depth-first-search với trạng thái đầu (nút gốc) đã biết :

- Nếu trạng thái đầu là đích (goal state), kết thúc thành công
- Ngược lại, tiếp tục các bước sau cho đến khi thành công hoặc thất bại :
  a. Từ trạng thái đầu chưa phải là nút thành công, tiếp cận một trạng thái kế tiếp, giả sử gọi là S. Nếu không tiếp cận được trạng thái kế tiếp nào, ghi nhận thất bại.
  b. Gọi lại thuật toán depth-first-search với S là trạng thái đầu.
  c. Nếu thành công, kết thúc. Ngược lại, tiếp tục vòng lặp.

IV.4.4.2. Tìm kiếm các lời giải

Ta định nghĩa hàm `a-solution` cho phép trừ một trạng thái đã cho, trả về một lời giải kế tiếp trạng thái này hoặc `#f` nếu thất bại. Nguyên lý hoạt động rất đơn giản như sau :

- Nếu là trạng thái cuối cùng (không còn trạng thái kế tiếp), thì đó là một lời giải để trả về, nếu không phải thì trả về `#f` và xem như thất bại.
- Nếu đang ở trạng thái trung gian, liệt kê tất cả các trạng thái kế tiếp và bắt đầu sự lựa chọn một trạng thái kế tiếp chừng nào chưa tìm ra lời giải.

Giả thiết ta đã xây dựng được các hàm : hàm `followingstates` cho phép liệt kê tất cả các trạng thái có thể tiếp cận đến xuất phát từ một trạng thái đã cho, vị từ `finalstate?` kiểm tra trạng thái cuối cùng, vị từ `solution?` kiểm tra tính hợp thức của lời giải. Trong hàm `a-solution` có sử dụng `some` là dạng `or` mở rộng (xem IV.3.1) :

```
(define (a-solution state)
    (if (finalstate? state)
        (if (solution? state) state #f)
         (some a-solution (followingstates state))))
```

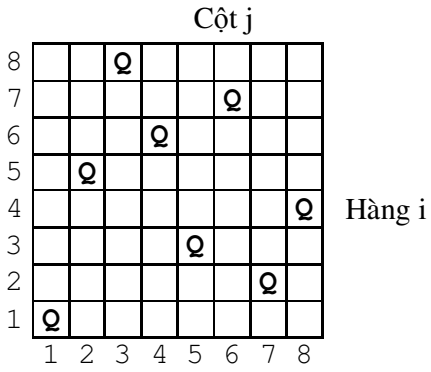
Trong một số trường hợp, người ta mong muốn nhận được danh sách tất cả các lời giải. Ta xây dựng hàm `list-of-solutions` bằng cách gộp vào danh sách lần lượt các lời giải tiếp theo có thể tìm được như sau :

```
(define (list-of-solutions state)
    if (finalstate? state)
```

```
        (if (solution? state)
            (list state) '())
            (append-map
                list-of-solutions
                (followingstates state))))
```

Cuối cùng, người ta mong muốn nhận được một số lời giải tùy theo yêu cầu của người sử dụng. Ta xây dựng hàm `some-solutions` dựa theo cách xây dựng hàm `a-solution` nhưng khi tìm thấy một lời giải, máy yêu cầu người sử dụng trả lời có tiếp tục tìm lời giải khác không. Vị từ `other-solution?` dùng để duy trì quá trình tìm kiếm. Sau khi đã đưa ra hết các lời giải tìm thấy, hàm trả về `#f`.

```
(define (some-solutions state)
    (if (finalstate? state)
         (if (solution? state)
             (other-solution? state)
             #f)
         (some-solutions (followingstates state))))
(define (other-solution? state)
    (display state)
    (newline)
    (display "Other solution (Y/N)?:")
    (eq? (read) 'n))
```



Hình IV.4. Một lời giải của bài toán 8 quân hậu.

Sau khi đã định nghĩa các hàm chính, ta cần định nghĩa các hàm bổ trợ `finalstate?`, `solution?`, `followingstates` tương ứng với một cách tổ chức dữ liệu cho bài toán 8 quân hậu. Trước tiên ta cần tìm cách biểu diễn các trạng thái tìm kiếm lời giải.

Bàn cờ vua có 8×8 ô được đánh số theo hàng 1..8 và theo cột 1..8. Do mỗi cột chỉ đăt được một con Q, ta cần biết vị trí là toạ độ hàng tương ứng với mỗi cột đang xét. Như vậy, một trạng thái sẽ là một danh sách các số nguyên $(x_1,..., x_k)$ với x_i là số thứ tự hàng của con Q thứ i đặt ở cột thứ i, i=1..k, k=1..8.

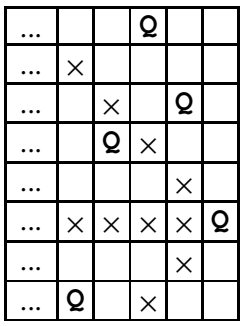
Ví dụ trạng thái của một lời giải được cho trong ***Error! Reference source not found.*** là danh sách đầy đủ :

(1 5 8 6 3 7 2 4)

Nghĩa là lời giải được tìm thấy khi trạng thái đã có đủ 8 con Q và mỗi con Q không thể bị ăn bởi một con Q nào khác trên bàn cờ. Từ đó, ta định nghĩa vị từ `solution?` cũng là `finalstate?`.

```
(define (finalstate? state)
    (= (length state) 8))
(define solution? finalstate?)
```

Để liệt kê các trạng thái tiếp theo từ một trạng thái đã cho, ta cần xét 8 vị trí là 8 hàng có thể trên cột tiếp theo để đặt con Q vào. Vị trí cho Q mới này (`newqueen`) phải được chọn sao cho không bị ăn bởi các Q khác trong trạng thái đang xét. Ta cần xác dịnh hàm `admissible?` để kiểm tra nếu một vị trí cho một con Q mới là tương thích với những con Q đã dặt trước đó.



con Q mới tại một vị trí chấp nhận được

Hình IV.5. Vị trí chấp nhận được của một quân hậu.

Nói cách khác, không có con Q nào nằm trên hàng, trên cột, trên đường chéo thuận và trên đường chéo nghịch đi qua vị trí này. Để xây dựng hàm `admissible?`, ta cần xây dựng một hàm bổ trợ kiểm tra khả năng chấp nhận của con Q mới với một con Q đã đặt trước đó tại một khoảng cách (distance) đã cho. Tham biến `distance` là khoảng cách giữa con Q mới và con Q trước đó. Còn `existing-queens` là danh sách trạng thái đang xét.

Từ ý tưởng trên, vị từ `admissible?` được xây dựng như sau :

```
(define (admissible? newqueen existing-queens)
   (letrec ((admisible-to?
     (lambda (existing-queens distance)
       (if (null? existing-queens)
           #t
           (let ((aqueen (car existing-queens)))
              ; kiểm tra lần lượt từng con hậu đã có mặt trong danh sách
              (and ; kiểm tra không cùng đường chéo thuận
                   (not (= aqueen (+ newqueen distance)))
                   ; kiểm tra không cùng hàng
                   (not (= aqueen newqueen))
                   ; kiểm tra không cùng đường chéo nghịch
                   (not (= aqueen (- newqueen distance)))
                   ; tiếp tục kiểm tra các quân hậu tiếp theo
                   (admisible-to? (cdr existing-queens)
                                  (+ 1 distance)))))))))
     (admisible-to? existing-queens 1)))
```

Vị từ `admissible?` không kiểm tra hai quân hậu nằm trên cùng cột. Vị trí chấp nhận được của một con Q được minh hoạ trên ***Error! Reference source not found.*** :

Để xây dựng danh sách các trạng thái tiếp theo một trạng thái đã đạt được ở bước trước, hàm `followingstates` dưới đây tìm kiếm vị trí chấp nhận được cho một con Q mới. Nếu tìm được vị trí thoả mãn, thêm toạ độ hàng vào cuối danh sách trạng thái để trả về :

```
(define (followingstates state)
  (append-map
    (lambda (position)
       (if (admissible? position state)
            (list (cons position state))
            '()))
     (list-1-to-n 8)))
```

Hàm `list-1-to-n` có mặt trong hàm `followingstates` dùng để liệt kê danh sách các số nguyên từ 1.. *n* được xây dựng như sau :

```
(define (list-1-to-n n)
  (letrec ((loop
       (lambda (k L)
          (if (zero? k)
               L
               (loop (- k 1) (cons k L))))))
    (loop n '())))
(list-1-to-n 8)
--> '(1 2 3 4 5 6 7 8)
```

IV.4.4.3. Tổ chức các lời giải

Như vậy, ta đã xây dựng xong các hàm chính và các hàm bổ trợ để tìm lời giải cho bài toán 8 quân hậu. Các hàm chính là :

`(a-solution state)`
--> Tìm một lời giải.

`(list-of-solutions state)`
--> Tìm tất cả các lời giải.

`(some-solutions state)`
--> Tìm lần lượt các lời giải theo yêu cầu.

Lúc đầu, trạng thái `state` là một danh sách rỗng, sau khi tìm ra lời giải, danh sách được làm đầy. Các hàm bổ trợ được sử dụng trong các hàm trên đây là :

`(finalstate? state)`

--> Vị từ kiểm tra nếu một trạng thái tương ứng với một vị trí không thể tiếp tục.

`(solution? state)`

--> Vị từ kiểm tra một trạng thái là một lời giải (tương tự hàm `finalstate?`).

`(followingstate state)`

--> Danh sách tất cả các trạng thái tiếp theo chấp nhận được và khả năng xuất phát từ một trạng thái đã cho.

`(admissible? newqueen existing-queens)`

--> Vị từ kiểm tra nếu vị trí của con Q mới không bị ăn bởi các con Q khác đặt tại các cột trước đó trong danh sách trạng thái.

Bây giờ ta chạy demo bài toán 8 quân hậu và nhận được kết quả như sau :

```
(a-solution '())
--> '(4 2 7 3 6 8 5 1)
```

```
(some-solutions '())
--> (4 2 7 3 6 8 5 1)
    Other solution (Y/N)?: y
    (5 2 4 7 3 8 6 1)
    Other solution (Y/N)?: y
    (3 5 2 8 6 4 7 1)
    Other solution (Y/N)?: y
    (3 6 4 2 8 5 7 1)
    Other solution (Y/N)?: y
    (5 7 1 3 8 6 4 2)
    Other solution (Y/N)?: y
    (4 6 8 3 1 7 5 2)
    Other solution (Y/N)?: n
    #t
```

Số lượng tất cả các lời giải cho bài toán tám quân hậu được tính như sau :

```
(length (list-of-solutions '()))
--> 92
```

Tóm tắt chương 4

- Một hàm trong Scheme có kiểu dữ liệu `procedure`. Để kiểm tra tính hợp thức về kiểu của hàm, sử dụng vị từ `procedure?`.
- Một hàm Scheme cũng có kiểu s-biểu thức, hàm có thể được dùng làm tham đối và cũng có thể làm giá trị trả về của một hàm khác.
- Một hàm Scheme có thể được định nghĩa theo cách thông thường hoặc sử dụng phép tính lambda.
- Phép tính lambda sử dụng các quy tắc biểu diễn hàm đệ quy và cho phép tính toán trên hàm để trả về kết quả.
- Giá trị một biểu thức lambda là một hàm nặc danh, khi sử dụng tính toán cần phải cung cấp các tham đối thực sự để trả về kết quả.
- Dạng `let` là tương đương với dạng lambda và có thể phối hợp để định nghĩa các hàm.
- Có nhiều kỹ thuật định nghĩa hàm : nhờ tích luỹ kết quả, tham đối hoá từng phần (currying), các bộ lặp và sơ đồ lặp, sơ đồ đệ quy và đệ quy cục bộ, trao đổi thông điệp giữa các hàm, tổ hợp các hàm, hàm có số lượng tham đối bất kỳ, v.v...
- Kỹ thuật lập trình hàm giải quyết hiệu quả thuật toán quay lui áp dụng cho bài toán 8 quân hậu.

Bài tập chương 4

1. Cho biết giá trị của :

```
((lambda (x y z) (+ x y z)) 1 (+ 2 5) 3)
((lambda (L) ((lambda (x) (append L (list x))) 0)) '(a))
```

2. Cho các định nghĩa sau :

```
(define (f xl x2 x3 x4)
  (lambda (m) (m xl x2 x3 x4)))
(define (g i z)
  (z (lambda (u v w x)
       (cond ((= i 1) u)
             ((= i 2) v)
             ((= i v) w)
             (else '())))))
(define x (f -2 3 4 20))
(define y (list 3 5))
(define z (cons y (list 3 5)))
```

Cho biết kết quả trả về của các lời gọi sau đây :

```
(g 0 x)
(g 4 1)
(g 3 x)
(eq? (cadr z) (car y))
(eq? (car z) (cdr z))
```

3. Cho :

$U_0 = V_0 = 1$

$U_n = U_{n-1} + V_{n-1}$

$V_n = U_{n-1} * V_{n-1}$

Dùng `letrec` tính giá trị của $U_3 * V_4$?

4. Các hàm sau đây làm gì ? Tìm độ phức tạp của chúng ?

```
(define (mystery1 L)
   (if (null? L)
      ()
      (append (mystery1 (cdr L))
              (list (car L)))))
(define (mystery2 L)
   (if (null? (cdr L))
      (car L)
      (if (> (car L) (mystery2 (cdr Ll)))
             (car L)
             (mystery2 (cdr L)))))
```

5. Cho đa thức bậc n hệ số thực (hoặc nguyên) một biến như sau :

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

Để biểu diễn $P(x)$ trong Scheme, người ta thường sử dụng một danh sách các hệ số theo một chiều :

$(a_0, a_1, a_2, \ldots a_n)$ hoặc $(a_n, a_{n-1}, \ldots a_1, a_0)$

Hãy viết trong Scheme hàm `(eval-pol p x)` để tính giá trị của đa thức $P(x)$ với một giá trị x sử dụng cả hai phương pháp đệ quy và phương pháp lặp, mỗi phương pháp xử lý theo hai cách biểu diễn hệ số trên đây.

6. Viết hàm tính độ sâu của danh sách :

```
(profondeur '(a (b ( c d)) e))
--> 3
```

7. Viết hàm đếm số lượng các phần tử có giá trị bằng phần tử đã cho :

```
(nb-occurrence* 'a '(e a c a (b c a) (a a))
--> 5
```

8. Viết hàm tạo danh sách nối vòng cho một danh sách đã cho.
9. Viết hàm kiểm tra một danh sách có là tiền tố của một danh sách đã cho.
10. Viết hàm đếm các phần tử của một danh sách nối vòng đã cho.
11. Viết đầy đủ thủ tục xác định danh sách con *L[B..A]*, nghĩa là tìm hai cận *B* (below), *A* (above), $1 \leq A, B \leq N$ sao cho tổng các phần tử của nó là tổng con lớn nhất của *L* (xem ví dụ ở mục III.4.3.1, *3*).
12. Cho một xâu ký tự có độ dài *N*, *N* được xem rất lớn. Hãy phân loại mỗi ký tự theo 4 kiểu như sau : kiểu chữ thường, kiểu chữ hoa, kiểu chữ số và kiểu khác (ký tự không thuộc ba kiểu trên) ?
13. Cho một danh sách có *N* từ (word) 32 bit, *N* được xem rất lớn. Hãy đếm số bit bằng 1 trong mỗi từ của danh sách đã cho ?
14. Cho một danh sách có *N* số nguyên. Hãy viết các thủ tục sắp xếp mô phỏng các thuật toán sắp xếp chèn và chọn.
15. Khi sắp xếp một dãy, người ta thường sử dụng hàm bổ trợ `swap(a, b)` để hoán đổi giá trị của hai biến. Hãy cho biết vì sao hàm sau đây không sử dụng được ?

```
(define (swap a b)
   (let ((temp a))
     (begin (set! a b)
        (set! b temp))))
```

16. Áp dụng thuật toán quay lui giải bài toán «mã đi tuần» trên bàn cờ vua 8×8 lúc đầu chưa có quân nào : xuất phát từ một ô, một quân mã có thể đi qua hết tất cả các ô của bàn cờ, mỗi ô đi qua đúng một lần, tuân theo luật cờ vua.
17. Áp dụng thuật toán quay lui giải bài toán «quân tịnh» : tìm cách đặt tối đa số quân tịnh (đen hoặc trắng) lên bàn cờ vua 8×8 lúc đầu chưa có quân nào sao cho không quân tịnh nào ăn được quân nào.

CHƯƠNG V. CẤU TRÚC DỮ LIỆU

Ta biết rằng việc lập trình phụ thuộc phần lớn vào cách mô hình hoá dữ liệu của bài toán cần giải. Mối quan hệ giữa thuật toán và cấu trúc dữ liệu trong lập trình đã được Niclaus Wirth, tác giả ngôn ngữ lập trình Pascal, đưa ra một công thức rất nổi tiếng :

Cấu trúc dữ liệu + Thuật toán = Chương trình

(Data structure + Algorithms = Programs)

Như đã thấy, thuật toán mới chỉ phản ánh các thao tác cần xử lý, còn đối tượng để xử lý trong máy tính lại là dữ liệu. Nói đến thuật toán là nói đến thuật toán đó tác động lên dữ liệu nào. Còn nói đến dữ liệu là nói đến dữ liệu ấy cần được tác động bởi thuật toán nào để đưa đến kết quả mong muốn. Dữ liệu biểu diễn thông tin cần thiết để giải bài toán, gồm dữ liệu đưa vào, dữ liệu đưa ra và dữ liệu tính toán trung gian. Một cấu trúc dữ liệu liên quan đến ba yếu tố : kiểu dữ liệu, các phép toán tác động lên dữ liệu và cách biểu diễn dữ liệu trong bộ nhớ của máy tính tuỳ theo công cụ lập trình.

Trong chương này, chúng ta sẽ trình bày một số cấu trúc dữ liệu tiêu biểu như tập hợp, ngăn xếp, danh sách móc nối, cây...

V.1 Tập hợp

Trong toán học, *tập hợp* (set) là một nhóm hay một bộ sưu tập các *đối tượng*[1] phân biệt $\{x_1, x_2, ..., x_n\}$, được gọi là các *phần tử* (elements) của tập hợp. Do mỗi phần tử của một tập hợp chỉ được liệt kê một lần và không được sắp xếp thứ tự nên người ta không nói đến phần tử thứ nhất, phần tử thứ hai, v.v... Ví dụ hai tập hợp sau đây là đồng nhất :

$$\{ a, b, d, a, d, e, c, d \} = \{ a, b, c, d \}$$

Do tập hợp cũng là một danh sách, người ta có thể sử dụng cấu trúc danh sách để biểu diễn tập hợp trong Scheme. Như vậy, một tập hợp rỗng là một danh sách rỗng. Để so sánh hai tập hợp có bằng nhau không, ta có thể sử dụng vị từ `equal?` như sau :

```
(define (setequal? E1 E2)
    (cond ; hai tập hợp rỗng thì bằng nhau
          ((and (null? E1) (null? E2)) #t)
          ; hai tập hợp có hai phần tử đầu tiên bằng nhau
```

[1] Khái niệm đối tượng của tập hợp có tính trực giác, do nhà toán học người Đức *G.* Cantor đưa ra từ năm 1985. Đến năm 1902, nhà triết học người Anh *B.* Russell đã chỉ ra những nghịch lý toán học (paradox) hay những mâu thuẫn lôgic trong lý thuyết tập hợp.

```
            ((equal? (car E1) (car E2))
                  (setequal? (cdr E1) (cdr E2)))
            ; hai tập hợp có hai phần tử đầu tiên khác nhau thì khác nhau !!!
            (else #f)))
(setequal? '(1 3 4 5 6) '(1 3 4 5 6))
--> #t
```

hoặc :

```
(define E1 '(a b c d e))
(define E2 E1)
(setequal? E1 E2)
--> #t
```

Để ý rằng trong vị từ `setequal?` trên đây, ta chưa xử lý hai tập hợp có các phần tử giống *y* nhau và có cùng số phần tử, nhưng không được sắp xếp thứ tự như nhau :

```
(setequal? '(1 5 4 3 6) '(1 3 4 5 6))
--> #f
```

Sau đây ta sẽ sử dụng thường xuyên hàm `member` để xử lý tập hợp. Hàm này kiểm tra một phần tử có thuộc một danh sách đã cho hay không :

```
; Trường hợp phần tử kiểm tra có kiểu đơn giản
(member  'c '(a b c d e))
--> '(c d e)
; Trường hợp phần tử kiểm tra có kiểu phức hợp
(member (list 'a) '(b (a) c))
--> '((a) c)
```

Vị từ `in?` kiểm tra một phần tử có thuộc một tập hợp đã cho hay không ?

```
(define (in? x E)
    (cond ((null? E) #f) ; danh sách rỗng
          ((member x E) #t) ; x là phần tử kiểu đơn giản
          (else (in? x (cdr E))))) ; x là phần tử kiểu phức hợp
(in? 'c E1)
--> #t
```

Để xây dựng một tập hợp từ một danh sách, người ta phải loại bỏ các phần tử trùng lặp. Hàm `list->set` sau đây sử dụng hàm `member` lần lượt kiểm tra các phần tử của danh sách đã cho. Kết quả trả về chỉ giữ lại phần tử cuối cùng đối với những phần tử trùng nhau và chưa sắp xếp lại các phần tử theo thứ tự (xem phương pháp sắp xếp nhanh ở cuối chương).

```
(define (list->set L)
    (cond ((null? L) '())
          ((member (car L) (cdr L))
                (list->set (cdr L)))
          (else (cons (car L)
                (list->set (cdr L))))))
(list->set '(a b d a d e c d))
--> '(b a e c d)
(define (union2 E1 E2)
    (cond ((null? E1) E2)
```

```
            ((member (car E1) E2) (union2 (cdr E1) E2))
            (else (cons (car E1) (union2 (cdr E1) E2)))))
```

1. Phép hợp trên các tập hợp

Giả sử cho hai tập hợp E_1 và E_2, ta cần tìm kết quả của phép hợp của hai tập hợp $E_1 \cup E_2$ là một tập hợp như sau :

```
(define (union2 E1 E2)
    (cond ; tập hợp thứ nhất là rỗng thì kết quả là tập hợp thứ hai
          ((null? E1) E2)
          ; nếu tập hợp thứ nhất có phần tử thuộc tập hợp thứ hai thì bỏ qua
          ((member (car E1) E2) (union2 (cdr E1) E2))
          ; tiếp tục sau khi giảm kích thước tập hợp thứ nhất
          (else (cons (car E1) (union2 (cdr E1) E2)))))
(union2 '(1 2 3 7) '(2 3 4 5 6))
--> '(1 7 2 3 4 5 6)
```

Mở rộng phép hợp của hai tập hợp, ta xây dựng phép hợp trên các tập hợp bất kỳ bằng cách sử dụng hàm `list-it` đã được định nghĩa ở chương trước :

```
(define (union . Lset)
    (list-it union2 Lset '()))
(union '(1 2 3 4) '(2 3 4 5 6) '(4 5 6 7) '(6 7 8))
--> '(1 2 3 4 5 6 7 8)
```

2. Phép giao trên các tập hợp

Tương tự cách xây dựng phép hợp trên các tập hợp bất kỳ, trước tiên ta xây dựng phép giao của hai tập hợp $E_1 \cap E_2$ như sau :

```
(define (intersection2 E1 E2)
    (cond ; nếu một tập hợp là rỗng thì kết quả cũng rỗng
          ((null? E1) E1)
          ; chọn ra phần tử nằm ở cả hai tập hợp
          ((member (car E1) E2) (cons (car E1)
              (intersection2 (cdr E1) E2)))
          ; tiếp tục sau khi giảm kích thước tập hợp thứ nhất
          (else (intersection2 (cdr E1) E2))))
(intersection2 '(1 2 3 4)'(2 3 4 5 6))
--> '(2 3 4)
```

Mở rộng phép giao của hai tập hợp, ta xây dựng phép giao trên các tập hợp bất kỳ bằng cách sử dụng hàm `apply` đã được định nghĩa ở mục **Error! Reference source not found.** :

```
(define (intersection . Lset)
    (cond ; giao của các tập hợp rỗng cũng là rỗng
          ((null? Lset) '())
          ; giao của một tập hợp là chính nó
          ((null? (cdr Lset)) (car Lset))
          ; đưa về thực hiện phép giao của hai tập hợp
          (else (intersection2
              (car Lset)
              (apply intersection (cdr Lset))))))
```

```
(intersection '(1 2 3 4) '(2 3 4 5 6) '(4 5 6 7))
--> '(4)
```

3. Phép hiệu của hai tập hợp

Cho hai tập hợp E_1 và E_2, ta âënh nghéa phép hiệu của hai tập hợp $E_1 \backslash E_2$ như sau :

```
(define (difference E1 E2)
    (cond ; nếu E2 rỗng thì kết quả là E1
          ((null? E2) E1)
          ; nếu E1 rỗng thì kết quả là rỗng
          ((null? E1) '())
          ; nếu E1 có phần tử thuộc E2 thì bỏ qua
          ((member (car E1) E2)
              (difference (cdr E1) E2))
          ; tiếp tục sau khi giảm kích thước tập hợp E1
          (else (cons (car E1)
                      (difference (cdr E1) E2)))))
(difference '(1 2 3 4 5) '(1 2))
--> '(3 4 5)
```

4. Tìm các tập hợp con của một tập hợp

Cho trước tập hợp E, ta cần tìm tập hợp tất cả các tập hợp con (sub-set) của E, ký hiệu 2^E. Chẳng hạn cho $E=\{ a, b, c \}$ thì $2^E = \{ \varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$ có tất cả 8 phần tử bao gồm tập hợp rỗng và bản thân tập hợp E.

Nếu tập hợp E rỗng, thì 2^E cũng rỗng. Nếu E khác rỗng, xét một phần tử $a \in E$, khi đó có các tập hợp con chứa a và các tập hợp con không chứa a. Đầu tiên xây dựng tập hợp $E\backslash\{a\}$, sau đó, chèn a vào tất cả các tập hợp con của $E\backslash\{a\}$. Ta có hàm `subset` như sau :

```
(define (subset E)
    (if (null? E)
        (list '())
        (let ((Lremain (subset (cdr E))))
          (append Lremain
             (map (lambda (L)
                    (cons (car E) L))
                  Lremain)))))
(subset '(a b c))
--> '(() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

V.2 Ngăn xếp

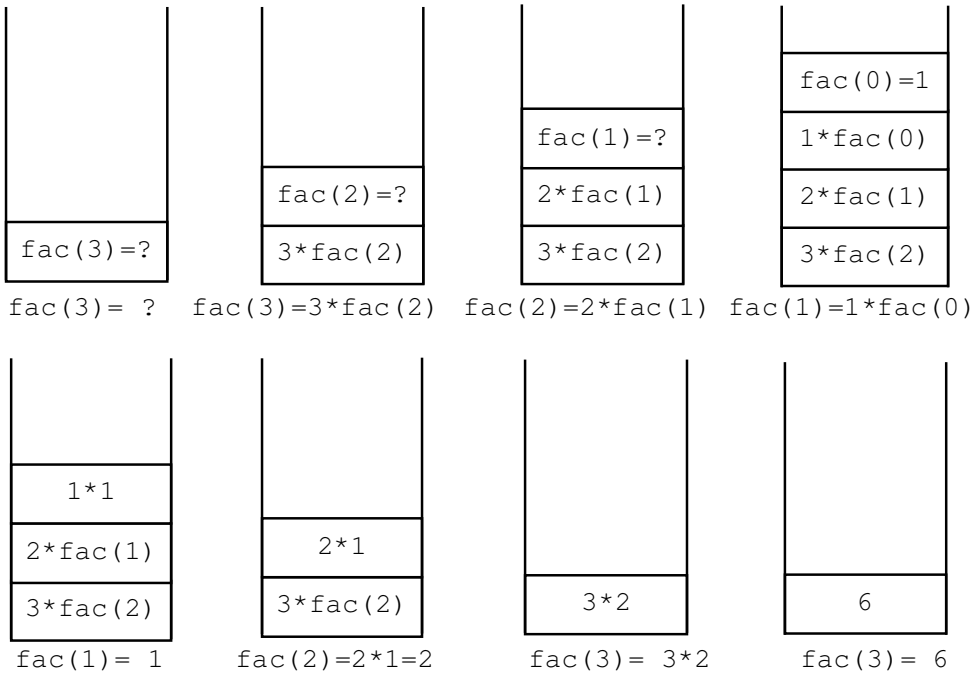
Danh sách kiểu *«ngăn xếp»* (stack), còn được gọi là «*chồng*» (tương tự một chồng đĩa, một băng đạn...), là một cấu trúc dữ liệu mà phép bổ sung hay loại bỏ một phần tử luôn luôn được thực hiện ở một đầu gọi là *đỉnh* (top). Nguyên tắc hoạt động *«vào sau ra trước»* của ngăn xếp đã dẫn đến một tên gọi khác là danh sách *kiểu LIFO* (Last In First Out).

Ngăn xếp được sử dụng rất phổ biến trong tin học. Hình V.1. minh hoạ hoạt động của một ngăn xếp khi thực hiện thủ tục đệ quy tính *n!*.

V.2.1. Kiểu dữ liệu trừu tượng ngăn xếp

Sau đây là đặc tả cấu trúc dữ liệu trừu tượng kiểu ngăn xếp :

```
Types Stack(T)
functions
   empty-stack:              → Stack(T)
   empty-stack?              : Stack(T)→ boolean
   push-stack  : T × Stack(T)     → Stack(T)
   pop-stack   : Stack(T)    -/→ Stack(T)
   top-stack   : Stack(T)    -/→ T
preconditions
   pop-stack(s: Stack(T))
          chỉ xác định khi và chỉ khi (not empty-stack?(s))
   top-stack(s: Stack(T))
          chỉ xác định khi và chỉ khi (not empty-stack?(s))
axioms
   var x: T, s : Stack(T)
   empty-stack?(empty-stack) = true
   empty-stack?(push-stack (x, S)) = false
   pop-stack(push-stack (x, S)) = S
   top-stack(push-stack (x, S)) = x
```



Hình V.1. Hoạt động của ngăn xếp thực hiện thủ tục đệ quy tính n!.

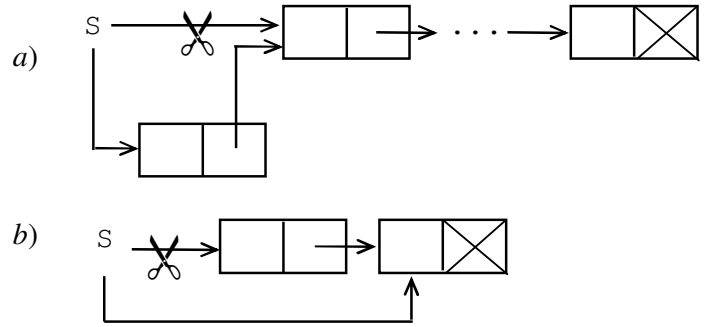
Ta cần xây dựng các hàm Scheme thao tác trên ngăn xếp như sau :

```
(empty-stack? (empty-stack))
-- #t
```

```
(empty-stack? (push-stack x S))
-- #f
(pop-stack (push-stack x S))
--> S
(top-stack (push-stack x S)
--> x
```

V.2.2. Xây dựng ngăn xếp

Có nhiều cách để biểu diễn cấu trúc dữ liệu kiểu ngăn xếp trong Scheme. Phương pháp tự nhiên hơn cả là biểu diễn ngăn xếp dưới dạng một danh sách mà các thao tác bổ sung và loại bỏ một phần tử được thực hiện ở một đầu danh sách. Trong Scheme, mỗi lần bổ sung một phần tử vào danh sách kéo theo việc tạo ra một bộ đôi (dotted pair) mới.



Hình V.2. Hoạt động bổ sung *(*a*)* và loại bỏ *(*b*)* một phần tử của ngăn xếp.

Ta có các hàm Scheme như sau :

```
(define (empty-stack) '())
(define empty-stack? null?)
(define (push-stack x S)
     (cons x S))
(define (pop-stack S)
     (if (empty-stack? S)
         (display "ERROR: stack is empty!")
         (cdr S)))
(define (top-stack S)
     (if (empty-stack? S)
         (display "ERROR: stack is empty!")
         (car S)))
```

Ta thấy các thao tác trên ngăn xếp tương tự đối với danh sách. Sau đây là một số ví dụ minh hoạ thao tác trên ngăn xếp sử dụng các hàm đã có phù hợp với ca1c tiên đề trong đặc tả trên đây :

```
(empty-stack? (push-stack 'a (empty-stack)))
--> #f
(pop-stack (push-stack 'a '(1 2 3)))
--> '(1 2 3)
```

```
(pop-stack (push-stack 'a '(1 2 3)))
--> '(1 2 3)
(top-stack (push-stack 'a '(1 2 3)))
--> 'a
(top-stack (pop-stack
    (push-stack 1 (pop-stack
       (push-stack 2 (push-stack 3 (pop-stack
           (push-stack 4 (push-stack 5
(empty-stack)))))))))))
--> 3
```

V.2.3. Xây dựng trình soạn thảo văn bản

Sau đây là một ví dụ đơn giản minh hoạ ứng dụng ngăn xếp để xây dựng một trình soạn thảo đơn giản cho phép thực hiện vào ra từng dòng văn bản một. Hoạt động như sau : hàm nhận dòng vào là một tham biến để lưu giữ trong một vùng nhớ trung gian (buffer). Dòng vào chứa các ký tự hỗn hợp mà một ký tự nào đó có thể là một lệnh soạn thảo (edit command). Có bốn loại ký tự như sau :

- Các ký tự khác ba ký tự `#`, `$` và `newline` đều là ký tự văn bản để được lưu giữ trong vùng nhớ trung gian.
- Ký tự `#` dùng để xoá ký tự đứng trước trong vùng nhớ trung gian.
- Ký tự `$` dùng để xoá tất cả ký tự trong vùng nhớ trung gian.
- Ký tự qua dòng `newline` để kết thúc một dòng vào và đưa nội dung vùng nhớ trung gian lên màn hình.

Ta sẽ sử dụng một ngăn xếp để biểu diễn vùng nhớ trung gian. Trong chương trình có sử dụng một hàm cục bộ `loop` để đọc các ký tự liên tiếp từ dòng vào. Chỉ số *i* là ký tự đang đọc thứ *i*.

```
(define (lineeditor str)
   (let ((L (string-length in-str)))
     (letrec ((loop
        (lambda (i buffer)
              (if (< i L)
                 (let ((readchar (string-ref in-str i)))
                    (case readchar
                    ((#\#) (loop (+ i 1) (pop-stack buffer)))
                    ((#\$) (loop (+ i 1) (empty-stack)))
                    ((#\.) (map (display (reverse buffer)))
                    (else (loop (+ i 1)
                 (begin
                    (display "ERROR: dot is the end of
                    command!")
                    (newline))))))
        (loop 0 (empty-stack)))))

(lineeditor "XY$abce#def.")
--> abcdef
```

Người đọc có thể phát triển trình soạn thảo trên đây để xử lý các tình huống soạn thảo văn bản phức tạp hơn.

V.2.4. Ngăn xếp đột biến

Ta muốn rằng hoạt động của ngăn xếp gần gũi với các chồng (pile) hơn, nghĩa là việc bổ sung loại bỏ phần tử chỉ liên quan đến một chồng để ta có thể gán cho biến. Ta xây dựng kiểu dữ liệu trừu tượng *ngăn xếp đột biến* (mutable stack) *Stack!*(*T*) (sau tên có một dấu chấm than) như sau :

```
Types Stack!(T)
functions
   empty-stack!   :                → Stack!(T)
   empty-stack!?  : Stack!(T)      → boolean
   push-stack!    : T × Stack!(T)  → Stack!(T)
   pop-stack!     : Stack!(T)      ↛ T
   top-stack!     : Stack!(T)      ↛ T
```

Để phân biệt với các ngăn xếp đã xét, ta thêm vào sau các tên kiểu và hàm một dấu chấm than **!** tương tự các đột biến. Điểm hoạt động khác biệt của ngăn xếp đột biếnlà sau khi loại bỏ một phần tử thì hàm trả về phần tử trên đỉnh.

Trong chương trước, khi xét môi trường làm việc của Scheme, ta thấy rằng không thể sử dụng lệnh `set!` để thay đổi nội dung của ngăn xếp chẳng hạn như :

```
(define (push-stack x s)
    (set! s (cons x s)) s)
```

Ta phải áp dụng tính chất đột biến của các danh sách, bằng cách biểu diễn ngăn xếp rỗng (empty stack) bởi một danh sách khác rỗng để sau đó thay đổi bởi `cdr`.

```
(define (empty-stack!) (list 'stack!))

(define (empty-stack!? S)
    (and (pair? S)
         (null? (cdr S))
         (eq? 'stack! (car S))))

(define S1 (empty-stack!))
S1            ; xem nội dung của ngăn xếp
--> '(stack!)
(empty-stack!? S1)
--> #t
```

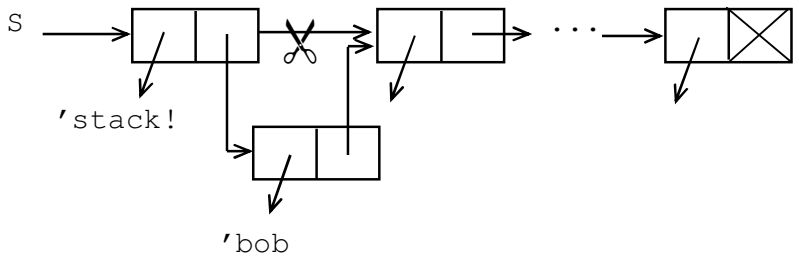
Để bổ sung (push) và loại bỏ (pop) trên ngăn xếp *S,* ta giả thiết rằng biến *S* luôn luôn trỏ đến cùng một bộ đôi. Tất cả các thao tác vật lý được thực hiện nhờ lệnh `cdr` như minh hoạ trong hình dưới đây.

Lúc ngăn xếp rỗng, danh sách *S* chỉ chứa phần tử `'stack!`. Sau khi bổ sung phần tử `'bob` thì phần tử này đứng thứ hai (ngay sau `'stack!`) trong danh sách *S*.

Ta xây dựng hàm như sau :

```
(define (push-stack! x S)
     (set-cdr! S (cons x (cdr S))))
```

```
(push-stack! 'ann S1)
(push-stack! 'bob S1)
(push-stack! 'jim S1)
```



Hình V.3. Hoạt động bổ sung một phần tử vào ngăn xếp.

Ta tiếp tục xây dựng hàm xem và loại bỏ phần tử đỉnh của ngăn xếp :

```
(define (top-stack! S)
     (if (empty-stack!? S)
         (begin
           (display "ERROR: stack! is empty!")
           (newline))
         (cadr S)))

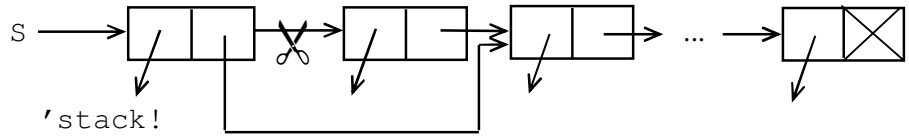
(top-stack! S1)
--> 'jim

(define (pop-stack! S)
     (if (empty-stack!? S)
         (begin
           (display "ERROR: stack! is empty!") (newline))
         (let ((top (cadr S)))
           (set-cdr! S (cddr S))
           top)))

(pop-stack! S1)
--> 'jim

(top-stack! S1)
--> 'bob
```

Thao tác vật lý phép loại bỏ được minh hoạ như sau :



Hình V.4. Hoạt động loại bỏ một phần tử khỏi ngăn xếp.

Sau đây là một cách khác sử dụng kỹ thuật truyền thông điệp để xây dựng các hàm xử lý cấu trúc dữ liệu kiểu ngăn xếp trong Scheme.

```
(define (make-empty-stack!)
    (let ((content '()))
       (lambda (message . Largt)
          (case message
```

```
            ((empty-stack!?) (null? content))
            ((push-stack!)
              (set! content
                    (cons (car Largt) content)) content)
            ((pop-stack!)
              (if (null? content)
                 (begin
                    (display "ERROR: stack is empty!")
                    (newline))
                 (let ((top (car content)))
                    (set! content (cdr content))
                    top)))
            ((top-stack!)
              (if (null? content)
                 (begin
                    (display "ERROR: stack is empty!")
                    (newline))
                  (car content))))))
```

; Kiểm tra ngăn xếp rỗng nhờ thông điệp 'empty-stack!?

```
(define (empty-stack!? S) (S 'empty-stack!?))
```

; Bổ sung một phần tử vào ngăn xếp nhờ thông điệp 'push-stack!

```
(define (push-stack! x S) (S 'push-stack! x))
```

; Loại bỏ một phần tử khỏi ngăn xếp nhờ thông điệp 'pop-stack!

```
(define (pop-stack! S) (S 'pop-stack!))
```

; Xử lý phần tử đỉnh của ngăn xếp nhờ thông điệp 'top-stack!

```
(define (top-stack! S) (S 'top-stack!))
```

; Định nghĩa một ngăn xếp mới

```
(define S2 (make-empty-stack!))
(empty-stack!? S2)
--> #t
(push-stack! 'ann S2)
--> '(ann)
(push-stack! 'bob S2)
--> '(bob ann)
(push-stack! 'jim S2)
--> '(jim bob ann)
(top-stack! S2)
--> 'jim
(pop-stack! S2)
--> 'jim
(top-stack! S2)
--> 'bob
```

V.2.5. Tính biểu thức số học dạng hậu tố

Một biểu thức số học thường được viết dưới dạng trung tố, chẳng hạn :

$(9 + 2) * \sqrt{52}$

Tuy nhiên, trong bộ nhớ, các biểu thức được biểu diễn dạng cây (ta sẽ xét cấu trúc cây trong mục tiếp theo). Khi tính toán, biểu thức số học dạng cây nhị phân (cho các phép toán hay hàm, gọi chung là *toán tư*, có tối đa hai *toán hạng*) được đưa ra thành một chuỗi ký tự dạng hậu tố (dạng ký pháp Ba Lan) như sau :

$9\ 2\ +\ 52\ \sqrt{}\ *$

Lúc này các dấu ngoặc không còn nữa do các biểu thức dạng hậu tố thường không nhập nhằng (un-ambiguous). Khi đọc biểu thức từ trái qua phải, ta gặp các toán hạng trước khi gặp một toán tử. Nếu áp dụng toán tử này cho các toán hạng trước đó, ta nhận được một kết quả là một toán hạng mới và quá trình tiếp tục theo đúng chế độ «vào trước ra sau». Do vậy, người ta sử dụng một ngăn xếp để mô phỏng quá trình tính toán biểu thức : các toán hạng (được đọc từ biểu thức hậu tố, lần lượt từ trái qua phải) được «đẩy vào» ngăn xếp cho đến khi gặp toán tử. Thực hiện phép toán với các toán hạng «lấy ra» từ ngăn xếp rồi lại đẩy kết quả vào ngăn xếp cho đến khi hết biểu thức và lấy kết quả nằm ở đỉnh ngăn xếp.

Sau đây ta xây dựng hàm tính một biểu thức số học dạng hậu tố (người đọc có thể tìm đọc các giáo trình «Cấu trúc dữ liệu và thuật toán» để hiểu chi tiết hơn). Chương trình sử dụng ngăn xếp đột biến xử lý các phép toán số học `+`, `-`, `*`, `/` và hàm `sqrt` :

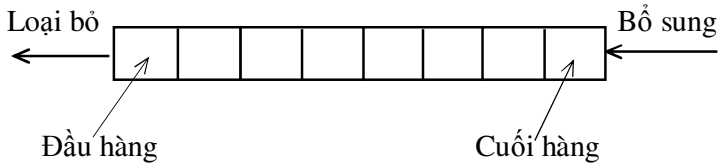
```
(define (evaluation postfix-expr)
   (let ((pile (make-empty-stack!)))
      (letrec ((compute
         (lambda (exp)
            (if (null? exp)
               (top-stack! pile)
               (let ((s (car exp)))
                  (if (number? s)
                     (push-stack! s pile)
                     (begin
                      (case s
                       ((sqrt)
                          (let ((v (pop-stack! pile)))
                          (push-stack! (sqrt v) pile)))
                        ((+)
                          (let ((v2 (pop-stack! pile))
                          (v1 (pop-stack! pile)))
                            (push-stack! (+ v1 v2) pile)))
                        ((*)
                          (let ((v2 (pop-stack! pile))
                          (v1 (pop-stack! pile)))
                             (push-stack! (* v1 v2) pile)))
                        ((-)
                          (let ((v2 (pop-stack! pile))
                          (v1 (pop-stack! piler)))
                             (push-stack! (- v1 v2) pile)))
                        ((/)
                          (let ((v2 (pop-stack! pile))
                          (v1 (pop-stack! pile)))
                            (push-stack! (/ v1 v2) pile))))
                  (compute (cdr exp)))))))))
      (compute postfix-expr))))
```

```
(evaluation '(9 2 + 52 sqrt *))
--> 79.3221
```

V.3 Tệp

V.3.1. Cấu trúc dữ liệu trừu tượng kiểu tệp

Cấu trúc dữ liệu kiểu *tệp* (file) mô phỏng một *hàng đợi* (queue) có chế độ hoạt động theo kiểu «vào trước ra trước» FIFO (First In First Out). Lúc này phép bổ sung được thực hiện ở cuối hàng (đuôi), và phép loại bỏ được thực hiện ở đầu hàng :



Hình V.5. Mô phỏng cấu trúc dữ liệu kiểu tệp tương tự một hàng đợi.

Ta xây dựng *File*(*T*) cấu trúc dữ liệu trừu tượng kiểu tệp có các phần tử kiểu *T* bất kỳ của Scheme như sau :

```
Types File(T)
functions
   empty-file :              → File(T)
   empty-file?: File(T)      → boolean
   push-file  : T × File(T) → File(T)
   pop-file   : File(T)      → File(T)
   top-file   : File(T)      → T
axioms
   var s: T, F: File(T)
   empty-file?(empty-file) = true
   empty-file?(push-file (s, F)) = false
   top-file(push-file (s, F)) =
        if empty-file?(F) then s else top-file(F))
   pop-file(push-file(s, F) =
        if empty-file?(F)
        then empty-file
        else push-file(s, pop-file(F))
```

Sau đây ta xây dựng các hàm xử lý tệp sử dụng kiểu danh sách của Scheme. Việc bổ sung phần tử được thực hiện ở cuối danh sách và việc loại bỏ thực hiện ở đầu danh sách.

```
(define (empty-file) '())
(define empty-file? null?)
(define (push-file s F)
   (append F (list s)))
(define (pop-file F)
   (if (empty-file? F)
```

```
        (begin
           (display "ERROR: file is empty!")
           (newline))
        (cdr F)))
(define (top-file F)
    (if (empty-file? F)
        (begin
           (display "ERROR: file is empty!")
           (newline))
        (car F)))
```

Ta nhận thấy rằng việc sử dụng danh sách là không hợp lý khi kích thước tệp tăng lên và lệnh `appenf` trong hàm `push-file` sẽ gây ra chi phí tính toán lớn.

V.3.2. Ví dụ áp dụng tệp

Sau đây ta xét một ví dụ áp dụng cấu trúc dữ liệu kiểu tệp vừa xây dựng. Giả sử cho trước một danh sách L gồm các số nguyên tăng ngặt và một số nguyên N, ta cần tính số lượng các cặp số (x, $x + N$) với x và N có mặt trong danh sách.

Chẳng hạn nếu L = (0 2 3 4 6 7 8 9 10 12 13 14 16 20) và N = 8, thì ta tìm đếm được bốn cặp số thoả mãn như sau : (2, 10), (4, 12), (6, 14) và (12, 20).

Phương pháp đơn giản nhất có tính trực quan là với mỗi số x lấy từ L, tiến hành tìm kiếm các số $x + N$ trong L . Tuy nhiên phương pháp này có chi phí lớn vì ta phải duyệt qua duyệt lại nhiều lần danh sách L. Sau đây là phương pháp chỉ cần duyệt một lần danh sách L.

Ta sử dụng tệp F có độ dài < N và kiểm tra mỗi phần tử x của F :

- Nếu tệp F rỗng thì ta chỉ việc thêm vào phần tử x.
- Nếu *top* là phần tử đầu tệp và $x > top + N$, ta chắc chắn rằng ta không thể còn tìm được phần tử y nào của F mà $y = top + N$ và do vậy ta phải loại bỏ F.
- Nếu $x = top + N$, ta loại bỏ x khỏi L để thêm x vào F và đếm cặp thoả mãn (x, *top*).
- Nếu $x < top + N$, ta chỉ loại bỏ x khỏi L để thêm x vào F.

Chương trình Scheme như sau :

```
(define (pair-count L N)
 (letrec ((count
     (lambda (L result F)
      (cond ((null? L) result)
       ((empty-file? F)
        (count
           (cdr L)
           result
           (push-file (car L) F)))
        (else
           (let
              ((difference (- (car L) (top-file F))))
               (cond
                ((< N difference)
                   (count L result (pop-file F)))
                ((= N difference)
```

```
                    (count
                       (cdr L)
                       (+ result 1)
                       (pop-file (push-file (car L) F))))
                 (else
                    (count
                       (cdr L)
                       result
                       (push-file (car L) F))))))))))
    (count L 0 (empty-file))))
(pair-count '(0 2 3 4 6 7 9 10 12 13 14 16 20) 8)
--> 4
```

V.3.3. Tệp đột biến

Giả sử gọi *File !*(*T*) cấu trúc dữ liệu kiểu tệp đột biến (mutable file), ta đặc tả kiểu trừu tượng như sau :

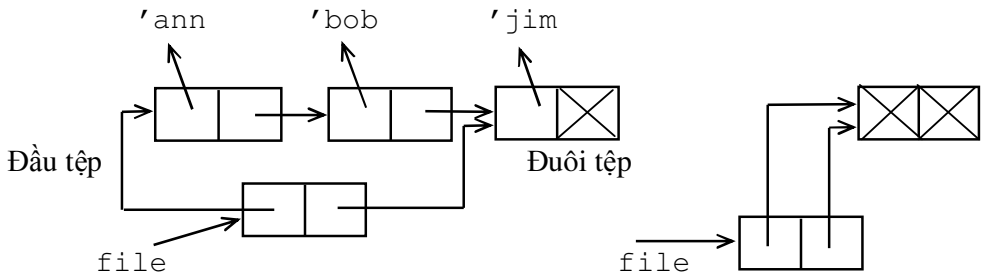
```
Types File!(T)
functions
   empty-file :              → File!(T)
   empty-file!?              : File!(T)→ boolean
   push-file! : T × File!(T)     → File!(T)
   pop-file!  : File!(T)     → File!(T)
   top-file!  : File!(T)     ⟶ T
axioms
   var s: T, F: File!(T)
   empty-file!?(empty-file!) = true
   empty-file!?(push-file! (s, F)) = false
   top-file!(push-file! (s, F)) =
        if empty-file!?(F) then s else top-file!(F))
   if empty-file!?(F)
   then pop-file!(push-file!(s, F) = s
   if not (empty-file!?(F))
   then pop-file!(push-file!(s, F) =
                        push-file!(s, pop-file!(F))
```

Ta có thể áp dụng kỹ thuật xây dựng các ngăn xếp đột biến cho cấu trúc dữ liệu kiểu tệp đột biến. Ta sẽ không sử dụng lệnh `append` để bổ sung phần tử mới vào tệp mà sử dụng một con trỏ trỏ đến đuôi của tệp. Với cách xây dựng này, độ dài tệp sẽ không biết giới hạn.

Như vậy, cấu trúc tệp có dạng một bộ đôi (dotted pair) mà thành phần `car` trỏ đến danh sách các phần tử là nội dung tệp, còn thành phần `cdr` trỏ đến phần tử bộ đôi cuối cùng của danh sách này.

```
(define (empty-file!)
    (cons '() '()))
(define (empty-file!? file)
    (null? (car file)))
```

```
(define (push-file! s file)
    (let ((doublet (cons s '())))
      (if (null? (car file))
          (set-car! file doublet)
          (set-cdr! (cdr file) doublet))
      (set-cdr! file doublet)))
```



a) Nội dung tệp `'(ann bob jim)` *b*) Tệp rỗng

Hình V.6. Hoạt động loại bỏ một phần tử khỏi ngăn xếp.

```
(define (top-file! file)
   (if (empty-file!? file)
       (begin
          (display "ERROR: file is empty!")
          (newline))
       (caar file)))
(define (pop-file! file)
   (if (empty-file!? file)
       (begin
          (display "ERROR: file is empty!")
          (newline))
       (begin (set-car! file (cdar file)))))
```

Sau đây là vận dụng xây dựng tệp có nội dung `'(ann bob jim)` :

```
(define F (empty-file!))
(empty-file!? F)
#t
(push-file! 'jim F)
(push-file! 'bob F)
(push-file! 'ann F)
(top-file! F)
--> 'jim
(pop-file! F)
(top-file! F)
--> 'bob
```

V.4 Cây

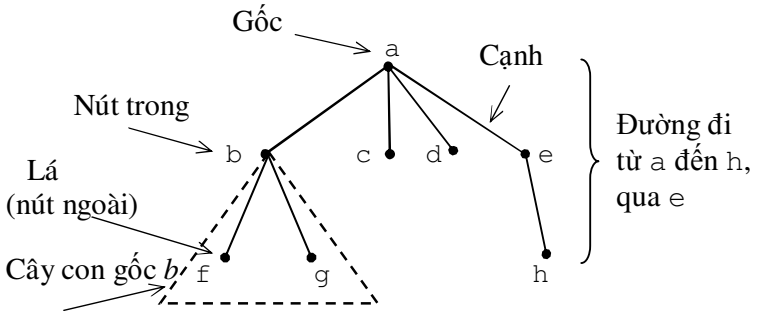
Cây (tree) gồm một tập hợp *nút* (node) được tổ chức theo kiểu phân cấp. Nếu cây không có nút nào thì được gọi là *cây rỗng* (empty tree).

Khi cây khác rỗng, nút trên cùng, cao nhất, là *nút gốc* (root node). Phía dưới nút gốc có thể có nhiều *nút con* (child node) nối với nó bởi các *cạnh* (edge). Lúc này, nút gốc là *nút cha* (father node). Mỗi nút con lại có thể là một nút cha có nhiều nút con nối với nó, và cứ thế tiếp tục. Phần cây tạo ra từ một nút con bất kỳ là *cây con* (subtree).

Những nút ở mức thấp nhất không có nút con nào nối với nó là *lá* (leaf) hay *nút ngoài* (exterior node). Một nút không phải là lá là *nút trong* (interior node). Các nút con có cùng cha là các *nút anh em* (siblings). Mỗi nút của cây có một cha duy nhất (trừ nút gốc), là *hậu duệ* hay *con cháu* (descendant) của nút gốc và là *tổ tiên* (ancestor) của các nút lá. Một *nhánh* (branch) hay một *đường đi* (path) là tập hợp các nút và các nhánh xuất phát từ một nút đến một nút khác.

Trong toán học, cây là một đồ thị (graph) liên thông không có chu trình. Trong một cây, luôn luôn tồn tại duy nhất một đường đi từ nút gốc đến một nút bất kỳ, hoặc giữa hai nút nào đó đã cho. Như vậy, một cây có thể rỗng, hoặc chỉ có một nút gốc, hoặc gồm một nút gốc và một hoặc nhiều cây con.

Cấu trúc cây được ứng dụng nhiều trong tin học : tổ chức thư mục của các hệ điều hành Unix, MS-DOS, ..., một chương trình Scheme, cây gia phả, ...



Hình V.7. Hình ảnh cây.

Sau đây ta sẽ xét cách Scheme xử lý các cấu trúc dữ liệu dạng cây, trước tiên là *cây nhị phân* (binary tree), loại cây mà mỗi nút cha chỉ có thể có tối đa hai nút con.

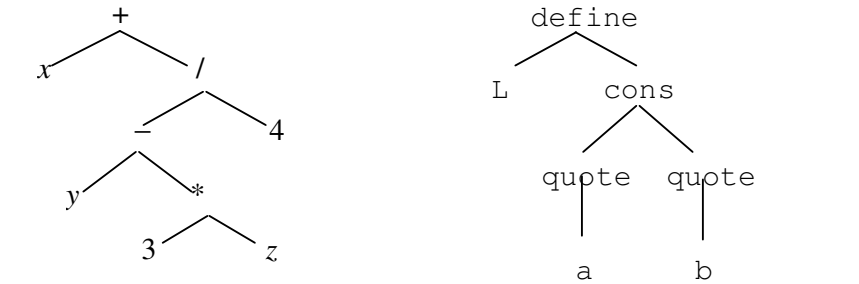
V.4.1. Cây nhị phân

V.4.1.1. Kiểu trừu tượng cây nhị phân

Cho trước một kiểu dữ liệu cơ sở *T* là một kiểu dữ liệu bất kỳ. Cây nhị phân kiểu T được định nghĩa như sau :

1. 0 (số không) là một *cây nhị phân* kiểu *T*, được gọi là cây rỗng.
2. Nếu *E* là một phần tử kiểu *T*, *A1* và *A2* là những cây nhị phân kiểu *T*, thì bộ ba (*E, A1, A2*) cũng là những cây nhị phân kiểu *T* .
3. Chỉ những đối tượng được định nghĩa bởi hai quy tắc trên đây mới là các cây nhị phân kiểu *T*.

Nếu *A* = (*E, A1, A2*) là cây nhị phân, thì *E* là nút *gốc*, *A1* là *cây con* bên trái, *A2* là cây con bên phải của *A*. Nếu *A* là cây suy biến chỉ gồm một nút, thì nút đó luôn luôn là nút gốc của một cây con. Người ta còn gọi *A* là cây *được gắn nhãn* (labelled tree, hay tag tree) bởi các giá trị trong *T*. Hình dưới đây minh hoạ hai cây nhị phân, một cây biểu diễn biểu thức số học *x+(y-3*z)/4* và một cây biểu diễn một s-biểu thức như sau :



a) *x* + (*y* −*3*z*)*/4* *b*) `(define L (cons 'a 'b))`

Hình V.8. Hai cây nhị phân biểu diễn :
a) *biểu thức số học ; b*) *biểu thức Scheme.*

Chú ý sự lệch đối xứng của định nghĩa : nếu *A* không phải là cây rỗng, thì cây (*E, A, 0*) khác với cây (*E, 0, A*). Cần phân biệt một nút với một giá trị gắn cho nút đó. Các nút luôn luôn rời (phân biệt) nhau, còn các giá trị thì có thể giống nhau. Ví dụ, cây biểu diễn s-biểu thức của Scheme trong hình V.9.(b) ở trên có 7 nút, trong khi đó chỉ có 6 giá trị phân biệt nhau gán cho chúng. Người ta đưa ra năm loại cây nhị phân đặc biệt như sau :

1. *Cây suy thoái* (degenerated hay filiform tree) mỗi nút chỉ có đúng một con không rỗng.
2. *Cây hình lược trái* (left comb) có mọi con bên phải là một lá.
   Cây hình lược phải (right comb) có mọi con bên trái là một lá.
3. *Cây đầy đủ* (complete tree) có mọi mức của cây đều được làm đầy
4. *Cây hoàn thiện* (perfect tree) mọi mức của cây đều được làm đầy, trừ mức cuối cùng, nhưng các lá là bên trái nhất có thể.
5. *Cây đầy đủ địa phương* mọi nút có từ 0 đến 2 con.

Giả sử *BinTree* (*T*) cấu trúc dữ liệu kiểu cây nhị phân, ta đặc tả kiểu trừu tượng như sau :

Types *BinTree* (*T*)

functions

```
empty-tree  :  BinTree(T)
empty-tree? :  BinTree(T) → boolean
create-tree :  T × BinTree(T) × BinTree(T) → BinTree(T)
root        :  BinTree(T) → T
left        :  BinTree(T) → BinTree(T)
right       :  BinTree(T) → BinTree(T)
leaf?       :  BinTree(T) → boolean (có thể không cần)
```

preconditions

```
root(A)   chỉ được xác định nếu và chỉ nếu A ≠ empty-tree
left(A)   chỉ được xác định nếu và chỉ nếu A ≠ empty-tree
right(A)  chỉ được xác định nếu và chỉ nếu A ≠ empty-tree
leaf?(A)  chỉ được xác định nếu và chỉ nếu A ≠ empty-tree
```

axioms

```
empty-tree?(empty-tree)  = true
```

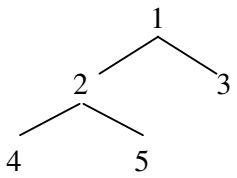
```
empty-tree?(create-tree(E, A1, A2))     = false
root(create-tree(E, A1, A2))  = E
left(create-tree(E, A1, A2))  = A1
right(create-tree(E, A1, A2)) = A2
leaf?(A) <=> empty-tree?(left(A)) và empty-tree?(right(A))
```

V.4.1.2. Biểu diễn cây nhị phân

1. Biểu diễn tiết kiệm sử dụng hai phép cons

```
; Định nghĩa cây rỗng :
(define empty-tree ())      ; hoặc
(define empty-tree (list))
; Định nghĩa vị từ kiểm tra cây có rỗng không :
(define empty-tree? null?)
; Định nghĩa cây khác rỗng (E, A1, A2) :
(define (create-tree E A1 A2)
    (cons E (cons A1 A2)))
; Định hàm trả về nút gốc :
(define root car)
; Định hàm trả về cây con trái :
(define left cadr)
; Định hàm trả về cây con phải :
(define right cddr)
; Định nghĩa vị từ kiểm tra có phải nút lá không :
(define (leaf? A)
    (and (null? (cadr A)) (null? (cddr A))))
; Ví dụ tạo một cây nhị phân có 3 nút :
(create-tree 1
   (create-tree 2 empty-tree empty-tree)
   (create-tree 3 empty-tree empty-tree))
--> '(1 (2 ()) 3 ())
```

Ví dụ : Cây *A* gồm 5 nút được cho như hình sau đây :



Hình V.10. Cây nhị phân có 5 nút.

được định nghĩa như sau :

```
(create-tree 1
   (create-tree 2
   (create-tree 4 empty-tree empty-tree)
   (create-tree 5 empty-tree empty-tree))
   (create-tree 3 empty-tree empty-tree))
--> '(1 (2 (4 ()) 5 ()) 3 ())
```

2. Biểu diễn dạng đầy đủ

Để dễ dàng xử lý khi lập trình, thông thường người ta biểu diễn cây dạng danh sách, đầy đủ các nút, sử dụng 3 phép `cons` :

```
(define empty-tree ())
(define empty-tree? null?)
(define (create-tree E A1 A2)
   (list E A1 A2))
(define root car)
(define left cadr)
(define right caddr)
(define (leaf? A)
   (and (null? (cadr A)) (null? (caddr A))))
```

; Ví dụ tạo một cây nhị phân có 3 nút :

```
(create-tree 1   (create-tree 2 empty-tree empty-tree)
(create-tree 3 empty-tree empty-tree))
--> '(1 (2 () ()) (3 () ()))
```

Cây *A* trong *Hình V.10.* trên đây được định nghĩa như sau :

```
(create-tree                                                    1
   (create-tree                                                 2
   (create-tree       4        empty-tree        empty-tree)
   (create-tree       5       empty-tree       empty-tree))
   (create-tree       3       empty-tree       empty-tree))
   --> (1 (2) (4 () ()) (5 () ())) (3 () ()))
```

Trong dạng đầy đủ, mỗi nút lá *A* đều được biểu diễn dạng `(A () ())`.

3. Biểu diễn đơn giản

Khi kiểu các phần tử không chứa các bộ đôi, kể cả không chứa danh sách, một lá có thể được biểu diễn đơn giản bởi giá trị của nút, mà không cần `cons`. Ta định nghĩa lại như sau :

```
(define empty-tree ())
(define empty-tree? null?)
(define (create-tree E A1 A2)
  (if (and (null? A1) (null? A2))
      E
      (list E A1 A2)))
      ; hay (cons E (cons A1 A2)) tiết kiệm hơn
(define (root A)
  (if (pair? A)
      (car A )
      A))
(define (left A)
  (if (pair? A)
      (cadr A)
      ()))
(define (right A)
```

```
   (if (pair? A)
       (caddr A)
       ;hay (cadr A) trong trường hợp biểu diễn tiết kiệm
   ()))
(define (leaf? A)
    (not (pair? A)))
; Ví dụ tạo một cây nhị phân có 3 nút :
(create-tree 1
                (create-tree 2 empty-tree empty-tree)
                (create-tree 3 empty-tree empty-tree))
                --> '(1 2 3)
```

Cây *A* trong *Hình V.10.* trên đây được định nghĩa như sau :

```
(create-tree 1
   (create-tree 2
   (create-tree 4 empty-tree empty-tree)
   (create-tree 5 empty-tree empty-tree))
   (create-tree 3 empty-tree empty-tree))
--> '(1 (2 4 5) 3)
```

Trong dạng đơn giản này, các nút của cây `A` được biểu diễn bởi giá trị của nút lần lượt theo thứ tự duyệt cây *gốc–trái–phải*.

V.4.1.3. Một số ví dụ lập trình đơn giản

1. Đếm số lượng các nút có trong một cây

Xây dựng hàm *size* có dạng hàm như sau :

```
; size : BinTree(T) → Integer
(define (size A)
   (if (empty-tree? A)
       0
       (+ 1 (size (left A)) (size (right A)))))
; Đếm số lượng các nút có trong cây A ở Hình V.10
   ````:
(size A)
--> 5
```

### 2. Tính độ cao của một cây

Độ cao của một cây khác rỗng là số *cạnh* tối đa nối gốc của cây với các lá của nó. Theo quy ước, độ cao của một cây rỗng có thể là –1. Xây dựng hàm *height* có dạng như sau :

```
; height : BinTree(T) → Integer
(define (height A)
   (if (empty-tree? A)
        -1
        (+ 1 (max (height (left A)) (height (right A))))))
; Độ cao của cây A ở Hình V.10.:
```
```

```
(height A)
--> 2
```

Kích thước n và độ cao h của cây nhị phân thường được dùng để đo độ phức tạp tính toán. Ta có :

$[log_2 n] \le h \le n-1$ (ký hiệu $[x]$ để chỉ phần nguyên của x)

Cây đầy đủ : $n = 2^{h+1} - 1$

Cây suy thoái : $n = h + 1$

Có tất cả $\frac{C_n^{2n}}{(n+1)}$ cây nhị phân kích thước n,

với C_p^n là tổ hợp chặp p (cách chọn p phần tử) của n (trong số n phần tử).

Cả hai hàm `size` và `height` trên đều có chi phí tuyến tính theo n.

V.4.1.4. Duyệt cây nhị phân

Cây được sử dụng chủ yếu để tổ chức lưu trữ dữ liệu nên khi cần khai thác người ta phải *duyệt* (traverse) hay *thăm* (visit) nội dung của cây. Có rất nhiều chiến lược duyệt cây. Do cây được xây dựng đệ quy, một cách tự nhiên, chương trình duyệt cây có thể sử dụng đệ quy. Mỗi chiến lược duyệt cây đều trả về kết quả là danh sách các nút đã được duyệt qua.

Ta sẽ xét sau đây chiến lược duyệt cây nhị phân theo chiều sâu. Trong trường hợp tổng quát, thuật toán có dạng như sau :

```
(define (run-around A)
  (if (empty-tree? A)
      <result>
      (f (root A)                          ; duyệt nút gốc
           (run-around (left A ))          ; duyệt cây con trái
           (run-around (right A))))))      ; duyệt cây con phải
```

Trong đó, `<result>` là kết quả trả về (thường là một danh sách rỗng) và `f` là hàm xử lý một phần tử của cây A (liên quan đến nội dung hay nhãn của nút vừa được duyệt). Thứ tự duyệt cây trên đây (gốc-trái-phải) là *thứ tự trước* hay còn được gọi là *thứ tự tiền tố* (prefixe order). Nếu đổi lại thứ tự duyệt trên đây thì ta nhận được :

- *Thứ tự giữa* hay *trung tố* (infixe order) nếu nút gốc được xử lý *giữa* các lời gọi đệ quy (trái-gốc-phải).
- *Thứ tự sau* hay *hậu tố* (postfixe order) nếu nút gốc được xử lý *sau* các lời gọi đệ quy (trái-phải-gốc).

Tuy nhiên, ta cũng có thể có thêm ba thứ tự duyệt cây theo hướng ngược lại là : *gốc-phải-trái, phải-gốc-trái* và *phải-trái-gốc*.

Ví dụ hàm `nodes` sau đây duyệt cây nhị phân để trả về danh sách tất cả các nút của cây đó. Do hàm sử dụng lệnh `append` nên có chi phí bậc hai đối với kích thước của cây.

```
; nodes: BinTree(T) → List(T)
; Duyệt cây nhị phân theo thứ tự trước :
(define (pre-nodes A)
```

```
   (if (empty-tree? A)
       '()
       (append (list (root A))
               (pre-nodes (left A))
               (pre-nodes (right A)))))
```

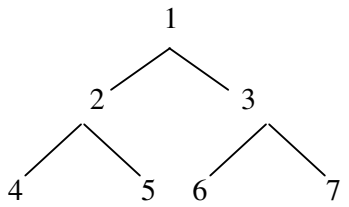
; Duyệt cây nhị phân theo thứ tự giữa :

```
(define (inf-nodes A)
   (if (empty-tree? A)
       '()
       (append (inf-nodes (left A))
               (list (root A))
               (inf-nodes (right A)))))
```

; Duyệt cây nhị phân theo thứ tự sau :

```
(define (post-nodes A)
   (if (empty-tree? A)
       '()
       (append (post-nodes (left A))
               (post-nodes (right A))
               (list (root A)))))
```

Để minh hoạ các thuật toán duyệt cây trên đây, giả sử *A* là cây được cho như hình sau :



Hình V.11. Cây nhị phân A có 7 nút

Cây *A* được định nghĩa theo kiểu đơn giản :

```
   (define A
      (create-tree 1
          (create-tree 2
              (create-tree 4 empty-tree empty-tree)
              (create-tree 5 empty-tree empty-tree))
   (create-tree 3
(create-tree 6 empty-tree empty-tree)
(create-tree 7 empty-tree empty-tree)))
--> '(1 (2 4 5) (3 6 7))
   (pre-nodes A)
   --> '(1 2 4 5 3 6 7); thứ tự trước
   (inf-nodes A)
   --> '(4 2 5 1 6 3 7); thứ tự giữa
   (post-nodes A)
   --> '(4 5 3 6 7 3 1); thứ tự sau
```

V.4.2. Cấu trúc cây tổng quát

Trong trường hợp cây tổng quát, mỗi nút đều được gắn nhãn và có số lượng cây con tuỳ ý không hạn chế. Một cách trực giác, cây có gắn nhãn là một cấu trúc dữ liệu gồm một giá trị nhãn và một danh sách có thể rỗng các cây con gắn nhãn. Như vậy cây không bao giờ rỗng, nhưng danh sách các cây con có thể rỗng. Trong trường hợp cây chỉ có một con, thì không có khái niệm cây con trái hay cây con phải. Ta cũng cần chú ý rằng cấu trúc cây nhị phân không phải là trường hợp riêng của cây tổng quát.

Ta có định nghĩa đệ quy như sau :

1. Một danh sách các cây kiểu *T* là một *rừng* (forest) kiểu *T*.
2. Nếu *E* là một phần tử kiểu *T* và *F* là một rừng kiểu *T*, thì cặp (*E*, *F*) là một cây kiểu *T*.
3. Chỉ những đối tượng được định nghĩa ở quy tắc 1 là rừng kiểu *T*.
   Chỉ những đối tượng được định nghĩa ở quy tắc 2 là cây kiểu *T*.

Cho trước tập hợp các cây kiểu *T*. Với mỗi cây *T,* các cây T_1, ..., T_n được gọi là con của *T*. Một nhãn *e nào đó là gốc của* T. Số con ở mỗi nút là *bậc* (degree) của nút đó. Một nút có có bậc 0 là một lá

V.4.2.1. Kiểu trừu tượng cây tổng quát

Ta xây dựng kiểu trừu tượng của cây tổng quát gồm *Tree*(*T*) và *Forest*(*T*)=*List*(*Tree*(*T*)) như sau :

types *Tree*(*T*), *Forest*(*T*)

functions

```
   ()          : Forest(T)
   null?       : Forest(T)     → boolean
   cons        : Tree(T)×Forest(T)      →Forest(T)
   car         : Forest(T)     → Tree(T)
   cdr         : Forest(T)     → Forest(T)
   create-tree: T × Forest(T)→ Tree(T)
   root        : Tree(T)       → T
   forest      : Tree(T)       → Forest(T)
   leaf?       : Tree(T)       → boolean
```

var `F: Forest(T); A, E: Tree(T)`

preconditions

`car(F)` và `cdr(F)` chỉ xác định khi và chỉ khi `F ≠ ()`

axioms (các tiên đề liên quan đến `list` không nhắc lại ở đây)

```
   root(create-tree(E, F))  = E
   forest(create-tree(E, F)) = F
   leaf?(A) and forest(A)= ()
```

V.4.2.2. Biểu diễn cây tổng quát

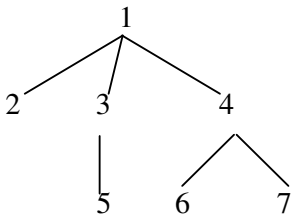
Ta đưa ra hai cách biểu diễn cây tổng quát như sau :

1. Biểu diễn cây nhờ một bộ đôi

```
(define create-tree cons)
(define root car)
```

```
(define forest cdr)
(define (leaf? A)
    (null? (cdr A)))
```

Ví dụ *A* là cây tổng quát như sau :



Hình V.12. Cây A tổng quát có 7 nút

A được biểu diễn bởi :

```
'(1 (2) (3 (5)) (4 (6) (7)))
```

2. Biểu diễn cây đơn giản qua các lá

Khi kiểu các phần tử không thể chứa các bộ đôi (kể cả các danh sách), một lá có thể được biểu diễn đơn giản bởi giá trị của nút.

```
(define (createe-tree E F)
   (if (null? F)
        E
        (cons E F)))

(define (root A)
   (if (pair? A)
        (car A)
        A))

(define (forest A)
   (if (pair? A)
        (cdr A)
        '()))

(define (leaf? A)
   (not (pair? A)))
```

Chẳng hạn cây *A* trên đây được biểu diễn bởi :

```
'(1 2 (3 5) (4 6 7))
```

V.4.2.3. Một số ví dụ về cây tổng quát

Nguyên tắc xử lý cây tổng quát *A* như sau : trước tiên kiểm tra danh sách các cây con là rừng của *A* có rỗng không ? Nếu rỗng thì dừng và trả về kết quả. Nếu khác rỗng thì lần lượt xử lý các cây con của *A*.

1. Đếm số lượng các nút trong cây

Hàm `tree-size` đếm số lượng các nút trong cây :

```
; tree-size : Tree(T) → Integer
(define (tree-size A)
   (define (size-forest F)
```

```
      (if (null? F)
        1
        (+ 1 (tree-size (car F)) (size-forest (cdr F)))))
   (size-forest (forest A)))
(define A '(1 2 (3 5) (4 6 7)))
(tree-size A)
--> 13
```

2. Tính độ cao của cây

Hàm `tree-height` tính độ cao của cây :

```
; tree-height : Tree(T) → Integer
(define (tree-height A)
   (define (height-forest F hmax)
      (if (null? F)
          hmax
          (height-forest (cdr F)
              (+ 1 (max (tree-height (car F)) hmax)))))
   (height-forest (forest A) 0))
(define A '(1 2 (3 5) (4 6 7)))
(tree-height A)
--> 3
```

V.4.2.4. Duyệt cây tổng quát không có xử lý trung tố

Chiến lược duyệt cây tổng quát tìm danh sách các nút gồm phép duyệt theo thứ tự trước và phép duyệt theo thứ tự sau, không có thứ tự trung tố.

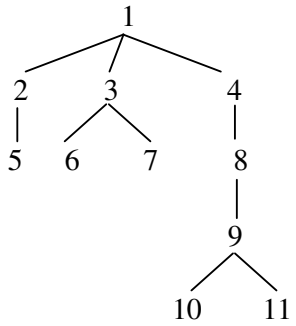
Thuật toán duyệt cây theo thứ tự trước : tại mỗi bước đệ quy, sử dụng lệnh `append` để tạo danh sách là nút gốc và kết quả duyệt các cây con của rừng :

```
; pre-nodes: Tree(T) → List(T)
(define (pre-nodes A)
   (define (nodes-forest F)
      (if (null? F)
          '()
          (append (pre-nodes (car F))
                  (nodes-forest (cdr F)))))
   (append (list (root A)) (nodes-forest (forest A))))
```

Thuật toán duyệt cây theo thứ tự sau : tương tự phép duyệt theo thứ tự trước nhưng phép ghép theo thứ tự ngược lại :

```
; post-nodes: Tree(T) → List(T)
(define (post-nodes A)
   (define (nodes-forest F)
       (if (null? F)
           '()
           (append (post-nodes (car F))
                   (nodes-forest (cdr F)))))
   (append (nodes-forest (forest A)) (list (root A))))
```

Ví dụ cho cây *A* gồm 11 nút sử dụng phương pháp biểu diễn cây đơn giản qua các lá :



Hình V.13. Cây A tổng quát có 11 nút

ta thực hiện hai phép duyệt cây như sau :

```
(define A
    '(1 (2 5) (3 6 7) (4 (8 (9 10 11)))))
(pre-nodes A)
--> '(1 2 5 3 6 7 4 8 9 10 11)
(post-nodes A)
--> ' (5 2 6 7 3 10 11 9 8 4 1)
```

V.4.3. Ứng dụng cây tổng quát

V.4.3.1. Xây dựng cây cú pháp

Cây cú pháp là cây tổng quát sao cho giá trị gắn cho mỗi nút xác định được số lượng các con của nút đó. Số nút này có thể cố định hoặc thay đổi tuy theo trường hợp vận dụng. Chẳng hạn một chương trình Scheme đúng đắn chính là một cây cú pháp.

Trong quá trình phân tích cú pháp (để biên dịch mã nguồn thành mã đích), người ta sử dụng một bảng dữ liệu lưu giữ thông tin liên quan đến phép toán và tham số tương ứng. Bảng này cung cấp các cặp giá trị gồm nút và số lượng các cây con tương ứng để kiểm tra tính đúng đắn của biểu thức.

Thông thường người ta sử dụng các giá trị nút ở dạng *nguyên tử* (atom). Người ta thường ưu tiên sử dụng cách biểu diễn đơn giản cây tổng quát qua các lá. Khi biết được số lượng các con của một nút, người ta dùng các phép toán tiếp cận đến cây con thứ *n*, tiếp cận đến các phép toán cơ sở, cũng như tiếp cận đến các cấu trúc dữ liệu đã được định nghĩa trong mã nguồn.

Sau đây là một số định nghĩa hàm tạo các cây cú pháp :

```
; Định nghĩa cây chỉ gồm một nút
(define (create0 E) E)
; Cây gồm một nút và một cây con
(define (create1 E A) (list E A))
; Cây gồm một nút và hai cây con
(define (create2 E A1 A2) (list E A1 A2))
```

v.v ...

```
; Hàm trả về cây con thứ nhất với điều kiện cây đã cho có tối thiểu 1 con
```

```
(define son1 cadr)
; Hàm trả về cây con thứ hai với điều kiện cây đã cho có tối thiểu 2 con
(define son2 caddr)
; Hàm trả về cây con thứ ba với điều kiện cây đã cho có tối thiểu 3 con
(define son3 cadddr)
```

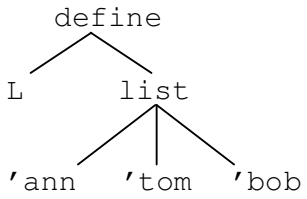
v.v ...

Chẳng hạn xây dựng cây cú pháp cho biểu thức Scheme

```
(define L (list 'ann 'tom 'bob))
```

như sau :

```
(define T
    (create2 'define 'L (create3 'list 'ann 'tom 'bob)))
T
--> '(define L (list ann tom bob))
(son2 (son2 T))
--> 'tom
```



Hình V.14. Cây nhị phân biểu diễn biểu thức Scheme.

V.4.3.2. Ví dụ : đạo hàm hình thức

Dưới đây là thủ tục tính đạo hàm của một biểu thức. Cây tổng quát biểu diễn biểu thức có quy ước như sau : các lá gồm số nguyên và ký hiệu biểu diễn các biến hình thức, các nút trong là các phép toán. Trong ví dụ này, để đơn giản, chỉ xét phép cộng và phép nhân.

```
; type Varable = Symbol
; type Operator = {+, *}
; type Exp = Tree(Integer ∪ Variable ∪ Operator)
; Chú ý : kiểu Exp có thể mô tả cụ thể hơn như sau :
; type Exp = Integer ∪ Variable ∪ Operator × Exp × Exp
; derivative : Exp × Variable → Exp
; Thủ tục trả về đạo hàm của E đối với biến V
(define (derivative E V)
  (cond
    ((leaf? E)
      (if (eq? (root E) V) 1 0))
    ((eq? (root E) '+)
      (create2 '+ (derivative (son1 E) V)
                  (derivative (son2 E) V)))
    ((eq? (root E) '*)
```

```
        (create2 '+ (create2 '* (derivative (son1 E) V)
                                            (son2 E))
                    (create2 '* (son1 E)
                                (derivative (son2 E) V))))
      (else
        (error "unknown operator" (root E)))))

(derivative '(+ x (* 2 y)) 'y)
--> (+ 0 (+ (* 0 y) (* 2 1)))

(derivative '(+ (* -3 x x) (* 2 x) 1) 'x)
--> (+ (+ (* 0 x) (* -3 1)) (+ (* 0 x) (* 2 1)))

(derivative '(* 2 x) 'x)
--> (+ (* 0 x) (* 2 1))
```

Bài tập chương 5

1. Cho trước một định nghĩa hàm như sau :

   ```
   (define (less k x)
     (cond ((null? x) '())
           ((< (car x) k) (cons (car x) (less k (cdr x))))
           (else (less k (cdr x)))))
   ```

   Hãy cho biết giá trị trả về của các lời gọi sau đây, giải thích tại sao :

   ```
   (less 5 '())
   (less 5 '(4 5 6))
   (less 5 '(7 3 6 2))
   ```

   Viết lại hàm trên nhưng chỉ sử dụng `if` mà không sử dụng `cons`, đồng thời loại bỏ biểu thức thừa `(less k (cdr x))` ?

2. Viết hàm chuyển đổi các số nhị phân (hệ cơ số 2) ra thập phân (hệ cơ số 10) và ngược lại (chú ý các tham biến là các số nguyên) theo gợi ý như sau :

   ```
   (binary->decimal 10001)
   --> 17
   (decimal->binary 17)
   --> 10001
   ```

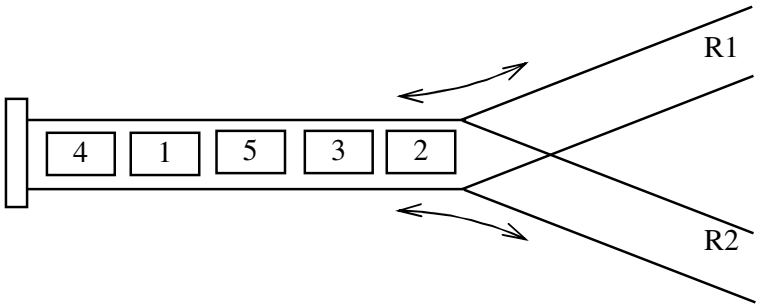
3. Viết hàm chuyển đổi cơ số để trả về kết quả là hàm chuyển đổi các số nhị phân ra thập phân và ngược lại theo gợi ý như sau (có thể định nghĩa các hàm bổ trợ) :

   ```
   ((base-converse 2 10) 10001)
   --> 17
   ((base-converse 10 2) 17)
   --> 10001
   ```

4. Viết hàm `(converse P N)` chuyển đổi số thập phân *N* bất kỳ sang số hệ cơ số *P*, với $P \le 36$ (số trong hệ cơ số *P* được viết bởi *P* chữ số 0..9A..*Z*) theo gợi ý như sau :

```
(converse 16 17)
--> 11
(converse 8 17)
--> 21
(converse 2 17)
--> 10001
```

5. Xây dựng cấu trúc dữ liệu kiểu ngăn xếp đột biến nhưng có kích thước giới hạn, nghĩa là số phần tử của ngăn xếp không vượt quá *MaxNum*. Có thể biểu diễn ngăn xếp bởi một vector, trong đó sử dụng một số nguyên làm chỉ số đỉnh của ngăn xếp.
6. Xây dựng hàm tính biểu thức số học dạng hậu tố sử dụng ngăn xếp thông thường.
7. Từ cách sử dụng kỹ thuật truyền thông điệp để xây dựng các hàm xử lý ngăn xếp trong lý thuyết, hãy viết thêm các hàm cho phép đưa ra xem nội dung của ngăn xếp với thông điệp `view-content`.
8. Viết lại hàm `nodes` không sử dụng lệnh `append` để trả về danh sách tất cả các nút của cây nhị phân.
9. Viết các thuật toán duyệt cây nhị phân theo chiều rộng.
10. Viết các thuật toán duyệt cây nhị phân theo chiều sâu.
11. Trong một nhà ga, người ta cần sắp xếp các toa tàu trên các đường ray như hình vẽ dưới đây. Sử dụng một ngăn xếp để biểu diễn hoạt động sắp xếp các toa tàu trên các đường ray, hãy cho biết các thao tác cần phải thực hiện sao cho các toa tàu được sắp xếp theo thứ tự 5 4 3 2 1 ?



Hình V.15. Sắp xếp theo thứ tự các toa tàu trên các đường ray.

12. Xây dựng hàm `(pairlis L1 L1 alist)` để trả về một danh sách kết hợp bằng cách thêm vào đầu của `alist` các bộ đôi nhận được từ các cặp phần tử lấy từ *L1* và *L2* lần lượt tương ứng (giả thiết *L1* và *L2* có cùng độ dài).

```
(pairlis '(1 2) (one two) '((3 . three) (4 . four)))
--> ((1. One) (2. two) (3. three) (4. four))
```

13. Xây dựng hàm `modival` để làm thay đổi giá trị kết hợp với một khóa thành giá trị mới, hay thêm mới một bộ đôi nếu khóa đã cho không tìm thấy trong một danh sách kết hợp đã cho.
14. Từ danh sách các số thực X_1, Y_1, X_2, Y_2, X_3 tương ứng lần lượt là toạ độ của ba điểm *A, B, C* trong một mặt phẳng toạ độ. Hãy cho biết ba điểm *A, B, C* có tạo thành một tam giác cân, đều, vuông, thường hay suy biến (diểm đường thẳng) ?

15. Từ danh sách 8 số thực $X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4, Y_4$ *lần lượt là toạ độ của bốn điểm A, B, C, D* tương ứng trong một mặt phẳng toạ độ. Hãy cho biết ba điểm *A, B, C* có lập thành một tam giác không ? Nếu *ABC* là một tam giác, hãy xét xem :
    1. Điểm *D* có nằm bên trong tam giác này không ?
    2. Tính diện tích và chu vi của tam giác
    3. Tính khoảng cách từ *D* đến ba điểm *A, B, C*.
16. Trong mặt phẳng toạ độ, phương trình đường dốc của một đường thẳng có độ dốc *m* và đi qua điểm *P* có toạ độ (x_1, y_1) là : $y - y_1 = m(x - x_1)$. Từ danh sách gồm độ dốc và toạ độ điểm của hai đường thẳng, hãy kiểm tra xem chúng có giao nhau hay song song với nhau. Nếu chúng có giao nhau, kiểm tra chúng có vuông góc với nhau hay không ?
17. Từ một số thực dương *R* và một số nguyên dương *MAX*, hãy tìm một phân số gần số *R* nhất trong số các phân số có dạng P/Q với $Q \leq MAX$?
18. Cho một ma trận các số nguyên $N \times N$ phần tử giả sử được biểu diễn dưới dạng một danh sách gồm *N* phần tử là các danh sách con, mỗi danh sách con biểu diễn một hàng *N* phần tử, liên tiếp hàng nọ nối tiếp hàng kia. Hãy xét xem ma trận đã cho có là ma trận gì :
    a) Ma trận tam giác trên-phải (có các phần tử nắm phía dưới đường chéo chính bằng 0) ?
    b) Ma trận tam giác dưới-phải (có các phần tử nắm phía trên đường chéo chính bằng 0) ?
    c) Ma trận tam giác trên-trái (có các phần tử nắm phía dưới đường chéo phụ bằng 0) ?
    d) Ma trận tam giác dưới-trái (có các phần tử nắm phía trên đường chéo phụ bằng 0) ?
    e) Ma trận đầy đủ (không rơi vào một trong 4 trường hợp tam giác ở trên) ?
    f) Tìm một cách biểu diễn tối ưu cho các ma trận tam giác ?

CHƯƠNG VI. MÔI TRƯỜNG VÀ CẤP PHÁT BỘ NHỚ

Chương này tập trung nghiên cứu các khái niệm *phạm vi* (scope) và *tầm nhìn* (visibility) của các biến trong khi thực hiện chương trình. Ta sẽ sử dụng bộ nhớ để mô hình hóa mối *liên kết* (link) giữa biến và giá trị của biến qua lệnh gán biến bởi `set!`. Ta sẽ nghiên cứu khái niệm cơ chế *đóng* (closure) trong mối liên hệ với dạng `set!` và trình bày cách vận dụng dạng lệnh liên kết biến `letrec`.

Từ khái niệm biến đột biến (biến thay đổi giá trị một cách bất thường) đã xét, ta đi đến khái niệm *nguyên mẫu* (protype). Đây là khái niệm về tổ hợp các cơ chế đóng và *gán* (assignment) để *đóng gói* (encapsulation) các trạng thái của một cấu trúc phức hợp và các xử lý trên đó, tương tự khái niệm lớp đối tượng trong lập trình hướng đối tượng.

VI.1 Môi trường

VI.1.1. Một số khái niệm

Ta xem xét các quy tắc thể hiện mối liên kết giữa một biến và giá trị của nó. Biến là tên gọi để chỉ định đến một giá trị, hay liên kết với một vị trí chứa giá trị trong bộ nhớ (bound occurrence). Liên kết một biến với một giá trị được hình thành khi tính một biểu thức kiểu liên kết, nghĩa là một biểu thức sử dụng một trong các dạng `let`, `let*`, `letrec`, `do`, `define`, hoặc khi gọi hàm, hoặc khi sử dụng phép tính lambda.

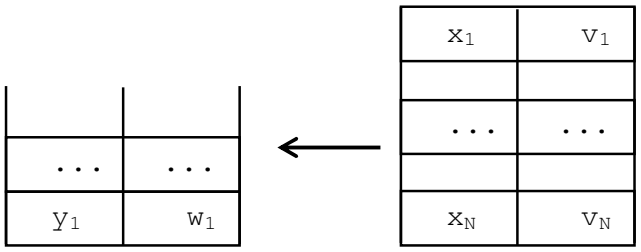
Khi gọi hàm, hoặc thực hiện phép tính lambda, các tham biến hình thức được liên kết với (được truyền) các giá trị tham đối thực sự tương ứng.

Khi một biểu thức được tính, câu hỏi đặt ra là : giá trị nào sẽ gán cho biến ? Giá trị này có thể tùy thuộc vào phạm vi của biến trong đoạn chương trình và vào *thời gian sống* (lifetime hay extend) của biểu thức đang tính. Để mô tả tập hợp các liên kết giữa biến đang hoạt động có thể tiếp cận và giá trị của chúng, người ta đưa vào khái niệm *môi trường* (environment). Môi trường được xem là một dãy các khối liên kết. Liên kết của một biến là giá trị gán cho biến trong môi trường.

Chẳng hạn, khi tính biểu thức với dạng `let` trong một ngữ cảnh nào đó, một môi trường hiện hành được hình thành. Dạng `let` có cú pháp như sau (xem chương 2) :

```
(let ((x1 e1)
      ...
      (xn eN))
   body)
```

Đầu tiên, các biểu thức ***e1***, ..., ***eN*** được tính đồng thời trong môi trường hiện hành để nhận được các giá trị ***v1***,..., ***vN***. tương ứng. Sau đó, thân của `let` được tính trong môi trường mới nhận được bằng cách thêm các khối liên kết ***x1*** → ***v1***, ..., ***xN*** → ***vN*** vào môi trường hiện hành như hình dưới đây :



Môi trường hiện hành của `let` Môi trường của thân của `let`

Hình VI.1. Môi trường hiện hành của dạng `let`.

Ví dụ trong lệnh gán :

```
(let ((x (+ 1 2)))
     (* x x))
```

thì biến *x* được liên kết với 3. Trên đây, ta thấy rằng quan hệ giữa một biến và giá trị của nó không thể giải thích mà không nói đến *bộ nhớ* (memory). Việc đưa vào lệnh gán đã làm thay đổi khái niệm ban đầu về môi trường.

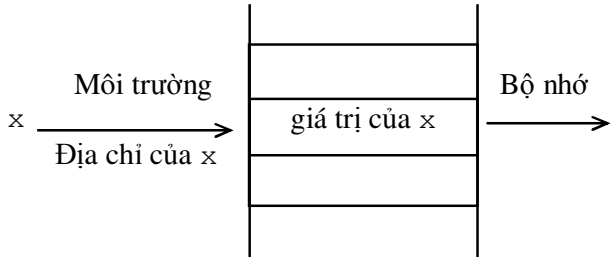
Một môi trường không chỉ ra giá trị của một biến nữa mà chỉ ra địa chỉ của nó trong bộ nhớ. Nghĩa là, giá trị của một biến phụ thuộc vào hai hàm :

- Hàm kết hợp một biến với một vị trí nhớ được gọi là *môi trường*.
- Hàm kết hợp một vị trí nhớ của biến với một giá trị chứa trong đó, gọi là *bộ nhớ*.

Giá trị của biến *x* được cho bởi việc tổ hợp các hàm này :

```
(memory (environmet x))
```

Ta có sơ đồ như sau :



Hình VI.2. Phụ thuộc của biến vào môi trường và bộ nhớ.

VI.1.2. Phạm vi của một liên kết

Trong Scheme, phạm vi của một liên kết còn được gọi là *vùng* (region). Phạm vi là *tĩnh* (static scope) nếu nó không phụ thuộc vào việc thực hiện chương trình. Trong trường hợp ngược lại, người ta gọi phạm vi là *động* (dynamic scope). Một môi trường mà không xảy ra các phép liên kết biến được gọi là *môi trường mức đỉnh* (top level environment). Sau đây ta sẽ xét kỹ hơn về hai khái niệm phạm vi tĩnh và phạm vi động.

VI.1.2.1. Phạm vi tĩnh

Giả sử cho biểu thức sử dụng hai lần dạng `let` :

```
(let ((a 5))
    (let ((f ((lambda (x) (+ a x)))
        (a 0))
      (f 10)) ; thân let trong
)
--> 15
```

Khi tính, biến *a* là nhìn thấy được trong thân của `let` ngoài, còn hàm *f* là nhìn thấy được trong thân của `let` trong. Thân của `let` trong cũng là môi trường của hàm `f`. Vấn đề đặt ra là vì sao giá trị của biểu thức tính được bằng 15 mà không bằng 10 ?

Trong định nghĩa hàm `f`, ta thấy `(f 10)` thực hiện việc thêm 10 vào giá trị của `a`, nhưng là giá trị nào của *a* ?

- Tại thời điểm định nghĩa `f`, thì *a* = `5` ?
- Tại thời điểm gọi `f`, thì lúc này *a* = `0` ?

Trong Scheme, người ta sử dụng giá trị của *a nhìn thấy được tại thời điểm định nghĩa* `f`. Việc lựa chọn này được gọi là phạm vi tĩnh, còn nếu lựa chọn giá trị tại thời điểm gọi hàm thì được gọi là phạm vi động. Vậy câu trả lời là 15.

Đối với các định nghĩa ở mức đỉnh, người ta cũng sử dụng nguyên tắc như vậy. Ta xét chầu làm việc sau :

```
(define b 10) ; b là biến toàn cục
(define (f x)
    (* b x))        ;b=10 trong môi trường của định nghĩa f
(f 5)
--> 50              ;f nhận được giá trị của b
(let ((b 0)) ;b không được nhìn thấy khi gọi f
(f 5))  ;f được gọi và chỉ nhìn thấy b toàn cục
--> 50
```

Nghĩa là giá trị của `b` tại thờiđiểm định nghĩa *f* đã được sử dụng để tính trong thân hàm.

VI.1.2.2. Phép đóng = biểu thức lambda + môi trường

Khi xét phạm vi tĩnh của biến, ta đưa vào khái niệm *biến tự do* (free occrence) là biến xuất hiện trong một biểu thức nhưng không được liên kết giá trị. Ví dụ biểu thức :

```
(lambda (x) (lambda (y) (+ x y a)))
```

chứa các biến tự do là *a* và +, các biến liên kết là `x` và `y`. Cùng một biến có thể vừa là biến liên kết vừa là biến tự do. Ví dụ, trong biểu thức sau, *x* vừa là một liên kết, vừa là một biến tự do :

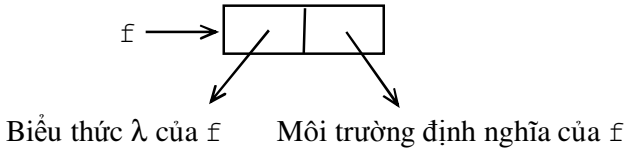
```
((lambda (x) (+ x 1)) x)
```

Vấn đề đặt ra là khi tính hàm sử dụng biểu thức lambda, làm sao nhớ được các tham biến có phạm vi tĩnh tại thời điểm gọi mà giá trị của chúng đã được xác định tại thời điểm định nghĩa lúc còn là biến tự do ?

Cách giải quyết đơn giản là sử dụng một cặp con trỏ kết hợp. Con trỏ thứ nhất được tạo ra tự nhiên nhờ biểu thức lambda định nghĩa hàm. Con trỏ thứ hai dùng để trỏ đến môi trường ở thời điểm định nghĩa hàm này. Cặp con trỏ, một trỏ đến biểu thức lambda và một trỏ đến môi trường, được gọi là một *phép đóng* (closure) :

(*closure* = *lambda eapression* + *environment*)

Như vậy, tại thời điểm gọi hàm, các tham biến hình thức sẽ được kết với các giá trị của các tham đối thực sự và các giá trị của các biến tự do sẽ được tìm kiếm trong môi trường định nghĩa biến nhờ con trỏ thứ hai.



Hình VI.3. Cặp con trỏ biểu diễn một phép đóng.

Hình vẽ mô tả cách biểu diễn bên trong của định nghĩa hàm nhờ phép tính lambda. Vùng nhớ gắn với một giá trị của biến chứa hai con trỏ : con trỏ địa chỉ của biểu thức lambda định nghĩa hàm và địa chỉ của môi trường định nghĩa hàm.

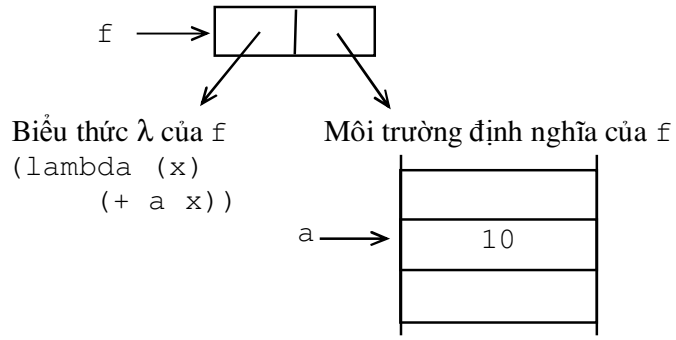
VI.1.2.3. Thay đổi bộ nhớ và phép đóng

Bộ nhớ biến bị thay đổi khi thực hiện các hàm có tên kết thúc bởi một dấu chấm than (**!**) và bởi lệnh `define`. Môi trường bị thay đổi khi gọi hàm, hay sử dụng `let`, *v*, *v*... Ví dụ sau đây làm thay đổi cả bộ nhớ và môi trường :

```
(let ((a 10))
     (set! a 0)
a)
--> 0
```

Bây giờ ta tổ hợp lệnh gán với việc tạo ra một phép đóng. Ta định nghĩa một biến cục bộ *a* và một hàm *f* sử dụng *a* này :

```
(let ((a 10))
     (let ((f (lambda (x) (+ a x))))
(f 0)))
--> 10
```



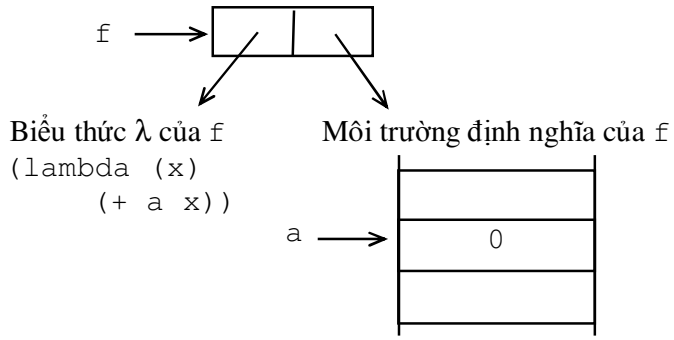
Hình VI.4. Phép đóng minh hoạ sử dụng biến.

Bây giờ nếu ta thay đổi giá trị của *a* bởi một lệnh `set!` thì giá trị của *a* đặt trong phép đóng của *f* đã bị thay đổi :

```
(let ((a 10))
     (let ((f (lambda (x) (+ a x))))
    (set! a 0)
(f 0)))
--> 0
```

Có thể giải thích hiện tượng này như sau : phép đóng kết hợp với *f* chứa một con trỏ đến môi trường này. Do liên kết này của *a* là nhìn thấy tại thời điểm của `set!` nên có sự đột biến nội dung vị trí nhớ của *a*, từ đó dẫn đến sự thay đổi giá trị của phép đóng kết hợp với *f*.

Sau đây là sơ đồ giải thích sau khi thực hiện `set!` :



Hình VI.5. Phép đóng sau khi thực hiện `set!`.

VI.1.2.4. Nhận biết hàm

Để nhận biết hàm, Scheme có vị tự `procedure?`

```
(procedure? car)
--> #t
(procedure? (labda (x) (* a x)))
--> #t
(procedure? (cons 'a 'b))
--> #f
```

Chú ý rằng `cons` không phải là một hàm. Mặt khác, việc so sánh các hàm bởi `eq?` chỉ kiểm tra tính đồng nhất vật lý của các đối tượng Scheme. Ví dụ :

```
(eq? cadr cadr)
--> #t
(eq? '(lambda (x) x) '(lambda (x) x))
--> #t
((lambda (L) (car L)) '(1 2 3))
--> 1
```

Tuy nhiên :

```
(eq? car (lambda (L) (car L)))
--> #f
```

kể cả khi hai đối tượng cần so sánh tỏ ra *y* hệt nhau :

```
(eq? (lambda (x) x) (lambda (x) x))
--> #f
```

Hai hàm *f* và *g* được gọi bằng nhau theo nghĩa rộng (extent equal) nếu chúng có cùng miền xác định và cùng nhận những giá trị như nhau trong miền này. Kiểu bằng nhau toán học này không kiểm định được khi miền xác định là vô hạn. Vì vậy, vị từ so sánh `equal?` cũng không dùng được để so sánh trên các hàm :

```
(equal? car (lambda (L) (car L)))
--> #f
```

```
(equal? (lambda (x) x) (lambda (x) x))
--> #f

(equal? cadr cadr)
--> #t

(eq? '(lambda (x) x) '(lambda (x) x))
--> #t
```

Ta nhận thấy hai vị từ so sánh `equal?` và `eq?` có cách dùng giống nhau.

VI.1.2.5. Phạm vi động

Giả sử ta cần định nghĩa các hàm như sau :

```
(define (f a)
     (+ a (g 5)))

(define (g x)
     (+ a x))
```

Khi gọi đến *f* hoặc *g*, sai số đều xuất hiện do biến *a* không phải là biến toàn cục :

```
(f 1)
--> ERROR: undefined variable: a

(g 1)
--> ERROR: undefined variable: a
```

Tại thời điểm gọi `(f 1)`, biến *a* được liên kết giá trị 1 (tham biến hình thức nhận giá trị của tham đối thực sự), nhưng liên kết này không được nhận biết bởi *g*. Khi đó lời gọi `(g 5)` gây ra lỗi. Tại thời điểm định nghĩa hàm *g*, biến *a* là tự do và phép đóng của *g* chứa con trỏ trỏ đến môi trường toàn cục nhưng *a* chưa được nhận biết trong môi trường này.

Như vậy, liên kết tĩnh không cho phép định nghĩa các hàm *f* và *g* như trên, mà phải có một định nghĩa biến toàn cục cho *a :*

```
(define a 10)
```

Khi đó, gọi đến *f* hoặc *g* đều có kết quả :

```
(f 1)
--> 16

(g 1)
--> 11
```

Quy tắc để tìm liên kết khi định nghĩa một biến phụ thuộc vào dạng thức liên kết và ngữ nghĩa của chúng. Trong dạng thức sử dụng phép tính lambda dưới đây :

```
((lambda (x1 ... xN) body)
      e1 ... eN)
```

thì phạm vi liên kết giữa một tham biến hình thức x_i với giá trị e_i là thân ***body*** của biểu thức lambda, tương tự dạng `let` :

```
(let ((x1 e1)
      ...
      (xN eN))
   body)
```

Trong dạng thức sử dụng `letrec` :

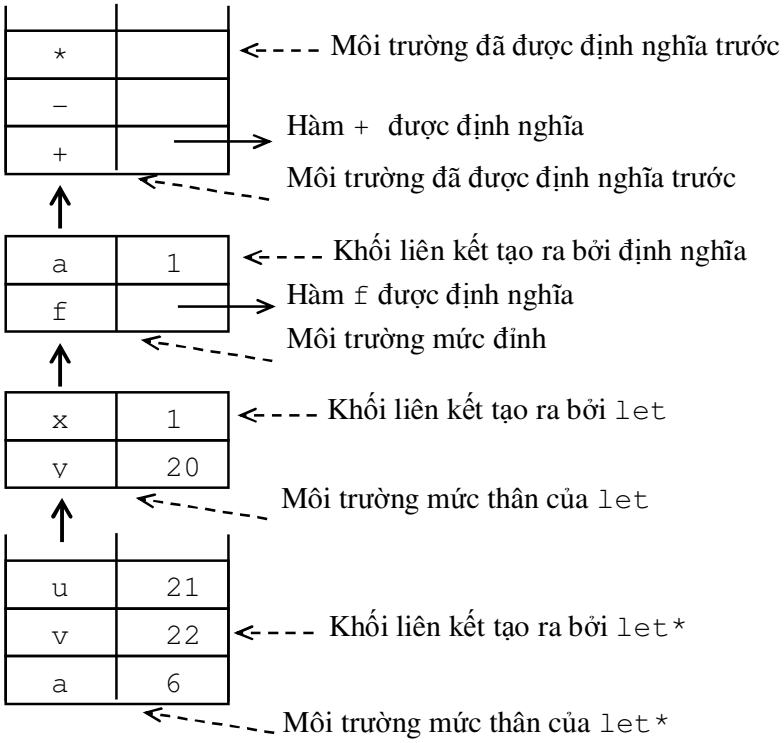
```
(letrec ((f1 e1)
         ...
         (fN eN))
   body)
```

thì phạm vi liên kết giữa f_i và giá trị e_i là thân `letrec` đã được thêm tất cả biểu thức e_j với mọi $j \neq i$.

Các biến toàn cục được định nghĩa bởi `define` có phạm vi là toàn bộ chầu làm việc. Người ta nói phạm vi của một định nghĩa là có hiệu lực trở về trước, vì rằng Scheme sẽ xem xét các định nghĩa trước đó trong chầu. Ví du :

```
(define add-m (lambda (x) (+ m x)))
(define m 10)
(add-m 5)
--> 15
```

Hàm *add-m* sử dụng liên kết cho *m*, mà *m* được định nghĩa sau hàm này. Cách hoạt động này cho phép lập trình thừa kế : người ta có thể định nghĩa các tiện ích sau khi đã định nghĩa các hàm chính. Điều này khác với ngôn ngữ như ML đòi hỏi các định nghĩa phải tuân theo một thứ tự nhận định.



Hçnh VI.1.1. Các môi trường của một chầu làm việc.

Thực ra, cách hoạt động trên hoàn toàn phù hợp với khái niệm tầm nhìn. Một liên kết trong cùng nhất có thể che dấu một liên kết khác của một biến có cùng tên. Trong trường hợp này, đó là liên kết mới nhất cần phải ghi nhận. Thực tế, người ta tìm liên kết theo thứ tự từ

trong ra ngoài trên các khối liên kết và ghi nhận liên kết tìm thấy đầu tiên. Cơ chế này được minh họa như sau :

```
(define a 1)            ;a=1
(define (f x)
    (+ x 1))
(let ((x a) (y 20))     ;x =1 ;y =20
    (let* ((n (+ x y)) ;n =21
           (v (+ a n)) ;v =22
           (a (f 5)))  ;a = 6
(list a (f a) x y n v)))
--> (6 7 1 20 21 22)
a --> 1
```

Trong chầu này, các môi trường được tạo ra như *Hçnh VI.1.1*. Để có giá trị của *f* mức thân của `let*`, cần phải ngược lên hai khối. Trái lại, giá trị của *a* tại điểm này được theo bởi một trong các liên kết của `let*` mà nó dấu giá trị của *a* đã định nghĩa ở mức đỉnh (là 1). Lúc này, giá trị của *a* vẫn là 1.

VI.1.3. Thời gian sống của một liên kết

Thời gian sống của một liên kết là thời gian liên kết có thể sử dụng được trong khi thực hiện chương trình (ngay cả khi không tiếp cận được tới liên kết này).

Trong Scheme, thời gian sống của một liên kết là không có giới hạn. Đây là đặc trưng của ngôn ngữ Scheme vì trong nhiều ngôn ngữ khác, thời gian sống chỉ thu hẹp ở lúc thực hiện chương trình với khối liên kết đang xét.

Ví dụ xét chầu làm việc sau :

```
(define f '?)     ;f có giá trị '?
(let ((a 10))
   (set! f (lambda (x) (+ x a))))
(f 5)
--> 15
```

Liên kết *a* cho 10 có phạm vi trong thân của `let`, nhưng trong thân này, lệnh `set!` đã tạo ra một phép đóng toàn cục để thu nạp liên kết này vào trong môi trường của nó.

Sau khi thực hiện `let`, lời gọi hàm *f* kéo theo việc sử dụng liên kết *a* cho 10 để tính lại giá trị của *f* trong thân hàm. Đặc điểm này đã làm cho việc biên dịch các chương trình Scheme trở nên rắc rối.

VI.1.4. Môi trường toàn cục

Tại một thời điểm của chầu làm việc, *môi trường toàn cục* (global environment) là tập hợp tất cả các liên kết nhìn thấy được tại những chu trình tương tác khác nhau. Đó là nơi chứa những đối tượng Scheme đã được định nghĩa, và tất cả những biến do người dùng định nghĩa lúc đầu.

Tuy nhiên, giá trị gắn với mỗi biến trong môi trường toàn cục có tính chất tạm thời vì có thể bị thay đổi trong chầu làm việc.

Chẳng hạn có thể thay đổi lại một số tên hàm trong thư viện của Scheme (tuy nhiên không nên làm như vậy) :

```
(define head car)
```

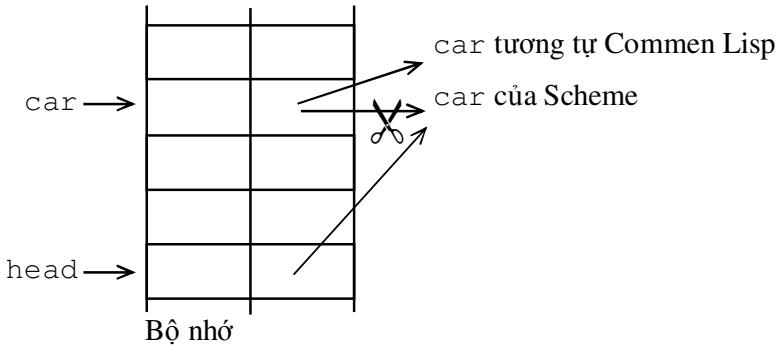
hoặc có thể thay đổi `car` để có cùng tính chất như `car` của ngôn ngữ Common Lisp : phần tử đầu của một danh sách rỗng không gây ra sai sót mà trả về một danh sách rỗng :

```
(define car
    (lambda (x)
        (if (null? x)
            '()
            (head x))))

(car '())
--> '()

(car '(a b c))
--> 'a
```

Tên hàm (biến) `car` không còn hiệu lực như car chuẩn của Scheme nữa nhưng không phải do tầm nhìn của nó, mà được minh họa qua sơ đồ sau :



Hình VI.6. Thay đổi `car` *của Scheme thành* `car` *của Common Lisp.*

Thủ tục định nghĩa `car` theo Common Lisp không làm thay đổi liên kết giữa tên `car` và vị trí lưu giữ nó trong bộ nhớ, nhưng làm thay đổi nội dung nhớ này. Rõ ràng, nên tránh dùng kiểu định nghĩa lại này vì sẽ gây ra rắc rối, nhầm lẫn khi đang đọc và bảo trì chương trình về sau. Việc khôi phục lại `car` được tiến hành như sau :

```
(define car head)
(car '())
--> ERROR: argument type
```

VI.2 Cấp phát bộ nhớ

Cho đến lúc này, ta mới chỉ làm việc với các đối tượng *thụ động* (passive objects), chưa nói đến cách Scheme cấp phát bộ nhớ như thế nào. Chẳng hạn, khi thêm một giá trị mới vào một cấu trúc dữ liệu, ta đã làm thay đổi trạng thái bộ nhớ. Người ta gọi những đối tượng thay đổi theo thời gian là những đối tượng *chủ động* (active objects) hay *đột biến* (mutable data). Sau đây ta xét hai ví dụ minh hoạ các đối tượng chủ động.

VI.2.1. Ví dụ 1 : mô phỏng máy tính bỏ túi

Trong một máy tính bỏ túi (calculator), thanh tích luỹ (accumulator), gọi tắt là ACC, là một bộ nhớ dùng cho nút M+. Ta mô phỏng cách làm việc với máy tính như sau : dùng lệnh `rtz` (reset to zero) để đặt ACC về giá trị `0`, lệnh `add` cộng thêm vào ACC một số nguyên, lệnh `value` đọc giá trị (nội dung) của ACC. Đây là một ví dụ đơn giản về một đối tượng có một trạng thái là giá trị hiện hành của ACC.

Yêu cầu đặt ra là định nghĩa các hàm Scheme tương ứng thoả mãn các lệnh vừa nêu trên :

```
(rtz)          ; đặt ACC về giá trị 0
(add 5)        ; cộng ACC với giá trị 5
(add 3)        ; cộng ACC với giá trị 3
(value)        ; đọc giá trị hiện hành của ACC
--> 8
(add 2)        ; cộng ACC với giá trị 2
(value)        ; đọc giá trị hiện hành của ACC
--> 10
```

Giả sử chỉ với lệnh định nghĩa biến `define`, ta xây dựng các hàm `rtz`, `add` và `value` như sau :

```
(define current-value 0)
(define (rtz) 0)
(define (add n)
     (+ current-value n))     ; chưa chạy được
(define (value) current-value)    ; luôn luôn trả về 0
```

Tuy nhiên, khi thực hiện các hàm trên thì không thoả mãn yêu cầu :

```
(rtz)
--> 0
(add 5)
--> 5
(add 3)
--> 3
(value)
--> 0
```

Ta luôn luôn nhận được kết quả 0 dù thực hiện bao nhiêu lần phép cộng mà không làm thay đổi giá trị ACC, tức là biến `current-value`. Để khắc phục tình trạng này, ta cần thay đổi cách gán giá trị cho biến, bằng cách sử dụng lệnh `set!`. Bây giờ ta có thể viết lại các hàm cho ACC như sau :

```
(define current-value 0)
(define (rtz)
     (set! current-value 0))
(define (add n)
     (set! current-value (+ n current-value)))
(define (value) current-value)
```

Lúc này, các hàm `rtz`, `add` và `value` hoạt động như yêu cầu đặt ra.

VI.2.2. Ví dụ 2 : bài toán cân đối tài khoản

Ví dụ sau đây xét bài toán cân đối tài khoản tại một ngân hàng. Ta sẽ làm việc với một đối tượng mà trạng thái của nó được đặc trưng bởi số dư, hay số chênh lệch (balance). Ta có thể định nghĩa kiểu trừu tượng `account` như sau :

```
Types account
operations
   create-account                    : Integer    → Account
   balance        : Account          → Integer
   add!           : Account × Integer → 0
   withdrawal!    : Account × Integer → 0
preconditions
   create-account(n)      chỉ xác định nếu và chỉ nếu  n>0
   add!(C, n)             chỉ xác định nếu và chỉ nếu  n>0
   withdrawal!(C, n) chỉ xác định nếu và chỉ nếu  balance(C)>n>0
```

Yêu cầu đặt ra là định nghĩa các hàm thoả mãn các hoạt động trao đổi tài chính như sau :

```
; Tạo một tài khoản mới để cân đối với giá trị khởi động là 1000
(define C1 (create-account 1000))
; Rút số tiền 100 khỏi tài khoản
(withdrawal! C1 100)
; Cân đối lại tài khoản
(balance C1)
--> 900
; Tạo một tài khoản khác để cân đối với giá trị khởi động là 500
(define C2 (create-account 500))
; Rút tiếp số tiền 100 khỏi tài khoản C1
(withdrawal! C1 100)
; Rút số tiền 100 khỏi tài khoản C2
(withdrawal! C2 100)
; Cân đối lại tài khoản C1
(balance C1)
--> 800
; Cân đối lại tài khoản C2
(balance C2)
--> 400
```

Để định nghĩa các hàm, ta không thể sử dụng một biến toàn cục như trường hợp thanh tích luỹ trước đây (là `current-value`), vì rằng mỗi tài khoản sẽ cần một biến toàn cục như vậy.

Ở đây ta sử dụng các biến cục bộ trong phạm vi một hàm. Tuy nhiên, các hàm `add!`, `withdrawal!` và `balance` phải tiếp cận được tới các biến cục bộ này. Muốn vậy, ta cần sử dụng kỹ thuật lập trình truyền thông điệp và các hàm được định nghĩa như sau :

```
(define (create-account S)
   (define (switch m)
```

```
      (cond
        ((eq? m 'balance)
              S)
        ((eq? m 'add!)
             (lambda (n) (set! S (+ S n))))
        ((eq? m 'withdrawal!)
             (lambda (n)
                (if (> n S)
                   (error "withdrawal's upper than balance"))
                (set! S (- S n))))))
   switch)
```

; Từ tài khoản `C`, gọi `create-account` với thông điệp `'balance`

```
(define (balance C) (C 'balance))
```

; Từ tài khoản `C`, gọi `create-account` với thông điệp `'add!`

```
(define (add! C n) ((C 'add!) n))
```

; Từ tài khoản `C`, gọi `create-account` với thông điệp `'withdrawal!`

```
(define (withdrawal! C n) ((C 'withdrawal!) n))
```

Các hàm vừa định nghĩa thoả mãn yêu cầu đặt ra. Sau khi tạo mới một tài khoản, có thể thực hiện ngay việc cân đối :

```
(define C3 (create-account 100))
(balance C3)
--> 100
(define C4 (create-account 0))
(balance C4)
--> 0
```

Do không xử lý cấp phát bộ nhớ khi thực hiện các phép gán biến, các hàm Scheme luôn luôn trả về cùng kết quả khi được gọi cùng những tham đối như nhau. Phép gán biến đã làm mất tính chất thuận lợi này. Một trong những nguyên nhân sai sót chính của lập trình là sự vi phạm tính *đồng nhất thức* (identity). Chẳng hạn, ta đưa ra hai định nghĩa cho hai tài khoản *C1* và *C2* như sau :

```
(define C1 (create-compte 1000))
(define C2 (create-compte 1000))
```

Hai đối tượng *C1* và *C2* là giống hệt nhau ? Lúc đầu, *C1* và *C2* được xây dựng như nhau, và như nhau chừng nào các tác động tài chính trên chúng còn giống nhau, nhưng với các tác động tài chính khác nhau :

```
(withdrawal! C1 500)
(withdrawal! C1 100)
(withdrawal! C2 100)
(balance C1)
--> 400
(balance C2)
--> 900
```

thì chúng trở nên khác nhau. Bây giờ ta xét ví dụ sau :

```
(define C1 (create-compte 1000))
(define C2 C1)
(withdrawal! C1 100)
--> 900
(balance C2)
--> 900
```

Lần này, *C1* và *C2* giống hệt nhau. Như thế, người ta có thể tiếp cận đến một đối tượng bởi nhiều tên gọi khác nhau, là những *biệt hiệu* (aliasing). Người ta gọi hiệu ứng phụ là việc thay đổi trạng thái của một đối tượng nhờ một biệt hiệu : ở đây, lời gọi `(withdrawal! C2 100)` đã gây ra một hiệu ứng phụ đối với *C1*.

Trong một số trường hợp, người ta cần hiệu ứng phụ, chẳng hạn việc quản lý một tài khoản chung. Tuy nhiên, do các các hiệu ứng phụ xảy ra ngoài ý muốn, nên rõ ràng rất khó phát hiện ra chỗ sai sót để xử lý. Các biểu thức Scheme được tính toán theo một trình tự có nghĩa và đòi hỏi một không gian nhớ đáng kể.

```
(define x 5)
(define (f) (set! x (- x 1)) x)
(- (f) (f))
--> 1 hay -1 hay ???
```

Kết quả tính toán tuỳ thuộc vào bộ diễn dịch Scheme. Chẳng hạn với Scheme 48 thì kết quả là 1. Để tránh tình huống này, cần bảo đảm một trình tự thực hiện thống nhất như sau :

```
(define (g)
   (define r1 (f))
   (define r2 (f))
   (- r1 r2))
```

Lúc này, ta thấy giá trị của *f* giảm dần -1, -2, ... tuy nhiên luôn luôn có :

```
(- (f) (f))
--> 1
(- (g) (g))
--> 0
```

VI.3 Mô hình sử dụng môi trường

Sau đây ta xây dựng một mô hình phức tạp hơn sử dụng khái niệm *môi trường*. Môi trường được xem là một *từ điển* (dictionary) gồm một dãy các *khung* (frame) (C_1, C_2, ... , C_n). Mỗi khung là một tập hợp các cặp *biến = giá trị*. Giá trị của một biến trong một môi trường là giá trị tương ứng với khung bên trái nhất trong danh sách các khung (một khung C_i có thể che khuất các biến của các khung C_j, $j > i$).

Có thể xem mỗi hàm tương ứng với một khung cục bộ chứa các định nghĩa bên trong hàm và có thể được bao bọc bởi một số môi trường nào đó. Lúc mới khởi động Scheme, môi trường toàn cục chứa tất cả các tên hàm sơ khởi. Sau đó, môi trường được cập nhật bởi các định nghĩa biến và định nghĩa hàm của người sử dụng.

Khi Scheme tìm kiếm một giá trị tương ứng với một tên, Scheme sẽ xem xét trước tiên khung tương ứng với các định nghĩa cục bộ, rồi sau đó, nếu cần thiết, Scheme sẽ xem xét khung tương ứng với các định nghĩa toàn cục, *v.v...* Như vậy, một định nghĩa sẽ cất dấu tại chỗ một định nghĩa toàn cục.

```
(define x 1)
(define (f x) x)
(f 2)
--> 2
x
--> 1
(define (g L)
    (define car cdr)
    (car L))
(g '(1 2 3 4 5))
--> '(2 3 4 5)
```

Ta đã thay đổi, một cách cục bộ, hàm tiền định `car` bởi `cdr` bên trong định nghĩa hàm `g`, điều này không ảnh hưởng đến `car` khi ở ngoài hàm `g` :

```
(car '(1 2 3))
--> 1
```

Tuy nhiên ta không nên thay đổi các hàm tiền định ở môi trường toàn cục (mức đỉnh), đại loại như :

```
(define car cdr)
```

Lúc này, hoạt động của `car` không còn đúng nữa :

```
(car '(1 2 3))
--> '(2 3)
> (cdr '(1 2 3))
--> '(2 3)
```

Môi trường toàn cục được rút gọn về một khung chứa các biến tiền định nghĩa (`cons`, `car`, `cdr`, ...) và các biến của người sử dụng. Scheme tính giá trị một biểu thức trong môi trường toàn cục này.

Một hàm được biểu diễn bởi một cặp *[I ; E]*, trong đó *I* là danh sách các lệnh của hàm, còn *E* là môi trường trong đó hàm được tính. Khi bộ dịch Scheme gọi một hàm trong một môi trường, nó tạo ra một *khung mới C* chứa các cặp :

(*tham đối hình thức, tham đối thực sự*)

Tiếp theo, bộ dịch thực hiện hàm trong môi trường ($C, C_1, C_2, \ldots, C_n$), trong đó, danh sách các khung ($C_1, C_2, \ldots, C_n$) là môi trường xử lý của hàm. Môi trường này có thể không liên quan gì đến môi trường gọi của nó. Khi hàm tạo ra các biến (nhờ các lệnh define) để bắt đầu được thực hiện, bộ dịch Scheme thêm các cặp *biến = giá trị* tương ứng vào trong khung *C*. Ví dụ :

```
(define x 1)
```

```
(define (f y)
     (define x 2)
     (+ x y))
```

Ngoài các biến đã định nghĩa trước đó, lúc này khung *C* của môi trường toàn cục chứa các cặp :
x = 1,
f = *P*, trong đó *P* = *[I ; E]* ,
I = `(lambda (y) (define x 2) (+ x y))`, và
E = (*C*)

Lời gọi `(f 5)` sẽ gây ra :

- Tạo ra khung *C'* = `{ y = 5}`
- Thực hiện các lệnh *I* trong (*C', C*)
- Sau khi `(define x 2)`, thì *C'* = `{ x = 2, y = 5}`
- Thực hiện `(+ x y)`, kết quả là 7. Ở đây, *x* và *y* được định nghĩa trong khung *C'*, hàm + được định nghĩa trong *C*.

Do vậy lời gọi `(f 5)` cho kết quả 7 :

```
(f 5)
--> 7
```

Bây giờ ta xét lại bài toán cân đối tài khoản ngân hàng (ví dụ 2 ở mục trước). Lời gọi :

```
(define account (create-account 1000))
```

sẽ gây ra :

- Hàm `create-account` được gọi trong môi trường (*C', C*)
- *C'* = `{S = 1000}`, *C* là khung toàn cục
- Sau khi thực hiện `(define (switch m) ... )`, thì
  C' = { `switch` = *P*, `S = 1000` }, với
  P = [các lệnh của `switch ;` (*C', C*)]
- Hàm `create-account` trả về giá trị của `switch`, là *P*

Biến `account` có giá trị là hàm :

P = [các lệnh của `switch ;` (*C', C*)]

Khi có lời gọi :

```
(balance account)
```

cũng có nghĩa là :

```
(account 'balance)
```

bộ dịch Scheme thực hiện các lệnh của `switch` trong môi trường :

(`{m = solde}`, *C'*, *C*)

Giá trị của `S` là trong *C'*, tức là `1000`. Và khi gọi :

```
((withdrawal! account) 100)
```

cũng có nghĩa là thực hiện lời gọi :

```
(account 'withdrawal!)
```

bộ dịch Scheme thực hiện hàm tạo tài khoản bởi `(withdrawal! account)`, rồi gọi hàm tham số là `100` :

– Các lệnh của `switch` được tính trong môi trường :
E = ({ *m* = `withdrawal!` }, *C'*, *C*)

– Thông điệp `withdrawal!` gây ra việc tính một hàm mới *[I' ; E],*
trong đó *I'* = `(lambla (n) (if ... ) (set! S (- S n)))`

– Lời gọi của hàm này thực hiện các lệnh trên trong môi trường :
(*C''*, { *m* =`withdrawal!` }, *C'*, *C*), trong đó *C''* = { *n* =`100` }

– `(set! S (- S n))` thay đổi giá trị của `S` trong khung chứa `S`, là *C'*

Khung *C'* đã bị thay đổi :

C' = { `switch` = *P*, `S = 900` }

Để sử dụng các xử lý cân đối tài khoản sau này, là `add!` và `withdrawal!` trong hàm `account`, *S* có giá trị `900`.

VI.4 Vào/ra dữ liệu

Các thủ tục vào/ra của Scheme cho phép đọc dữ liệu vào từ cổng vào (input port) và đưa dữ liệu ra cổng ra (output port). Cổng có thể là bàn phím, màn hình, hay tệp (file).

Trong chương trước, chúng ta đã được làm quen với các hàm `read` (đọc dữ liệu vào từ bàn phím), `write` và `display` (đưa dữ liệu ra màn hình). Các hàm này xem bàn phím, màn hình là các cổng vào/ra mặc định (console). Cổng vào có giá trị `'current-input-port'` và cổng ra có giá trị `'current-output-port'`. Các cổng vào/ra mặc định này có thể đặt làm tham đối *port* trong hàm vào/ra. Chẳng hạn :

```
(define e (read))
(write e)
```

hay có tham đối *port* :

```
(define (read (current-input-port)))
(write e (current-output-port))
```

là hiệu quả như nhau.

VI.4.1. Làm việc với các tệp

Tệp là phương tiện lưu trữ một dãy các dữ liệu bất kỳ trên một thiết bị nhớ ngoài như đĩa mềm, đĩa cứng, CD-ROM, băng từ, v.v... Tệp được đặt tên theo quy ước của hệ điều hành để dễ dàng truy cập đến trong hệ thống thư mục tổ chức kiểu phân cấp. Để làm việc (đọc/ghi) với tệp, cần thiết lập sự liên kết giữa tệp với một hệ thống Scheme. Sự liên kết này tạo ra một luồng hay *dòng dữ liệu* (data flow) giữa tệp và các hàm vào/ra của Scheme.

Khi đọc ra (read) một tệp, cần liên kết tệp này với dòng vào (input flow) của Scheme để cung cấp các hàm đọc. Khi ghi lên (write) một tệp, cần liên kết tệp này với dòng ra (output flow) của Scheme để cung cấp các hàm đưa (ghi) ra.

Scheme có các hàm liên kết một dòng dữ liệu với một tệp như sau :

`(open-input-file` *`filename`*`)`
`-->` liên kết dòng vào với tệp *`filename`* đã có mặt trên thiết bị nhớ.

`(open-output-file` *`filename`*`)`
`-->` liên kết dòng ra với tệp *`filename`* cần tạo ra.

Sau khi sử dụng một dòng dữ liệu, cần phải đóng lại bởi :

`(close-input-port` *`flow`*`)`
`-->` đóng dòng vào *`flow`*.

`(close-output-port` *`flow`*`)`
`-->` đóng dòng ra *`flow`*.

Để liên kết một tệp với một dòng, cần sử dụng các hàm sau đây :

`(call-with-input-file` *`filename fct`*`)`
`(call-with-output-file` *`filename fct`*`)`

Các hàm này sẽ đóng tự động các dòng sau khi sử dụng xong. Tham biến *`fct`* phải là một hàm có một tham biến sao cho tham biến này được liên kết với dòng trong khi thực hiện thân hàm và dòng sẽ tự đóng lại sau khi thực hiện xong.

Đối với trường hợp `call-with-input-file`, tệp truy cập đến đã có mặt trên thiết bị nhớ, còn đối với trường hợp `call-with-output-file`, lời gọi này không có tác dụng nếu tệp đã tồn tại.

Dòng dữ liệu là một đối tượng Scheme do đó nó có thể được kiểm tra bởi vị từ :

`(input-port ?` *`s`*`)`
`-->` `#t` nếu *`s`* là một dòng vào.

`(output-port ?` *`s`*`)`
`-->` `#t` nếu *`s`* là một dòng ra.

Như đã nói ở trên, dòng vào hiện hành cho bởi hàm `(current-input-port)`, dòng ra hiện hành cho bởi hàm `(current-output-port)`.

VI.4.2. Đọc dữ liệu trên tệp

1. Các hàm đọc tệp

Các hàm đọc tệp của Scheme có một tham đối tuỳ chọn (optional) là một dòng. Hàm :

`(read [`*`flow`*`])`

đọc s-biểu thức đầu tiên tại vị trí đầu đọc của tệp liên kết với dòng *`flow`* và dời đầu đọc đến vị trí bắt đầu của s-biểu thức tiếp theo, sau ký tự cuối cùng của s-biểu thức đã đọc. Khi đầu đọc đi đến cuối tệp, một đối tượng đánh dấu kết thúc tệp (end of file) được nhận biết bởi vị từ :

`(eof-object ? s)`

Giả sử tại thư mục hiện hành đang lưu giữ tệp "`mailto.txt`" có nội dung :

```
University of Danang
41, Le Duan St., Danang City, Vietnam
Tel: (84.511) 822041, 832678 - Fax: (84.511) 823683
```

Sử dụng `read` để đọc một s-biểu thức nhờ liên kết dòng vào với tệp như sau :

```
(define flow (open-input-file "mailto.txt"))
flow
--> '#{Input-port #{Input-channel "mailto.txt"}}
(define s1 (read flow))
s1
--> 'university
(define s2 (read flow))
s2
--> 'of
```

Hoặc sử dụng kết tệp với dòng vào :

```
(call-with-input-file "mailto.txt"
     (lambda (flow)
   (let ((s (read flow))) s)))
--> 'university
```

Chú ý tham biến thứ hai của `call-with-input-file` là một hàm một biến, ở đây là định nghĩa hàm bởi lambda theo biến `flow`. Hàm `read` thực tế là một bộ phân tích cú pháp (syntax analyser) cho các s-biểu thức, bằng cách chuyển cách biểu diễn bên ngoài của s-biểu thức có mặt trên tệp thành biểu diễn bên trong để sau đó được xử lý bởi Scheme.

Tiếp theo đây là một ví dụ minh hoạ quá trình đọc tệp. Ta xây dựng vị từ kiểm tra một s-biểu thức `s` đã cho (trong trường hợp này là toàn bộ một định nghĩa hàm hợp thức) có mặt trong một tệp `f` gồm các s-biểu thức hay không (`s`∈`f`?). Vị từ là sử dụng một vòng lặp nhờ `letrec` để đọc từng s-biểu thức và so sánh với s cho đến khi thoả mãn hoặc gặp ký hiệu kết thúc tệp.

```
(define (expr-present? s f) ; f là tên hàm sẽ đọc
   (letrec ((loop
             ; xây dựng vòng lặp cho đến khi gặp ký hiệu kết thúc tệp
             (lambda (flow)
                 (let ((expr-read (read flow)))
                   (cond ((eof-object? expr-read) #f)
                         ; kiểm tra bằng nhau hai s-biểu thức
                         ((equal? expr-read s) #t)
                         (else (loop flow)))))))
         ; liên kết tệp f với dòng dữ liệu vào
         (call-with-input-file f loop)))
```

Việc gọi vị từ được thực hiện như sau :

```
(expr-present? 'Danang "mailto.txt")
--> #t
```

Scheme còn có các hàm đọc từng ký tự trên dòng vào. Hàm :

```
(read-char [flow])
```

đọc vào một ký tự và di chuyển đầu đọc sang ký tự tiếp theo (người đọc có thể tìm hiểu sâu hơn trong tài liệu «*Revised(5) Report on the Algorithmic Language Scheme*»). Hàm :

```
(peek-char [flow])
```

đọc vào một ký tự nhưng không di chuyển đầu đọc sang ký tự tiếp theo.

Scheme không cho phép sử dụng `read` để đọc một tệp dữ liệu bất kỳ (kiểu Pascal), mà phải đọc từng ký tự một bởi `read-char`. Chẳng hạn :

```
(define flow (open-input-file "mailto.txt"))
(read-char flow)
--> #\U
```

Lời gọi sau đây đọc 4 ký tự đầu của tệp `"mailto.txt"` :

```
(call-with-input-file "mailto.txt"
   (lambda (flow)
      (let* ((c1 (read-char flow))
             (c2 (read-char flow))
             (c3 (read-char flow))
             (c4 (read-char flow)))
        (list c1 c2 c3 c4))))
--> '(#\U #\n #\i #\v
```

Sau đây ta xây dựng hàm `readfile` thực hiện đọc và chép nội dung một tệp lên màn hình. Hoạt động của `readfile` tương tự vị từ `expr-present?` vừa định nghĩa trên đây : sử dụng một vòng lặp nhờ `letrec` để đọc lần lượt các ký tự bởi `read-char` rồi đưa ra màn hình cho đến khi gặp ký hiệu kết thúc tệp *eof* :

```
(define (read-file f)
   (letrec
     ((loop
       (lambda(flow)
              (let ((char  (read-char  flow)))
                (if  (not  (eof-object? char))
                   (begin  (display  char)
                     (loop  flow)))))))
      (call-with-input-file  f loop)))
(read-file "mailto.txt")
--> University of Danang
    41, Le Duan St., Danang City, Vietnam
    Tel: (84.511) 822041, 832678 - Fax: (84.511) 823683
```

2. Tệp văn bản

Một số tệp có cấu trúc là một chuỗi liên tiếp các *khối* (block). Chẳng hạn tệp văn bản là một chuỗi các *dòng* (line), mỗi dòng là một khối được kết thúc bởi dấu qua dòng. Trong Scheme dấu qua dòng là ký tự quy ước `#\newline`.

Để đánh dấu vị trí cuối của một khối, người ta thường sử dụng một hoặc nhiều ký tự phân cách (separator mark). Việc đọc được thực hiện lần lượt cho các khối của tệp. Dưới đây là hàm `read-block` đọc tất cả các khối trong một dòng ra để trả về kết quả là một chuỗi ký tự. Hàm sử dụng hai tham đối, một tham đối là dòng vào liên kết với tệp văn bản và một tham đối là vị từ `separator?` cho phép xử lý được các ký tự phân cách khác nhau.

Hàm bên trong là một vòng lặp tích lũy các ký tự của khối vừa đọc vào một danh sách để sau đó chuyển đổi danh sách thành chuỗi trước khi đặt vào cuối chuỗi kết quả. Trước khi đọc một khối, cần gọi hàm `read-separator` để xoá hết các ký tự phân cách :

```
(define (read-block flow separator?)
   (letrec
```

```
((loop
   (lambda (Lcar)
    (let ((c (peek-char flow)))
       (if (separator? c)
           (list->string (reverse Lcar))
           (loop (cons (read-char flow) Lcar)))))))
(read-separator separator? flow)
(loop '())))
```

Nếu muốn đọc từng dòng trong tệp văn bản, ta sử dụng vị từ `séparator?` được định nghĩa như sau :

```
(define (separator?)
   (lambda(char)
        (eq? #\newline char))
```

Tuy nhiên ta cũng có thể đọc tệp theo từng *từ* (word). Thông thường các từ đặt cách nhau bởi các ký tự : khoảng trống hay dấu cách (space), ký tự thoát (escape), nhảy cột (tabulation), dấu qua dòng, dấu kết thúc trang. Trong Scheme, các ký tự này được nhận biết bởi vị từ `char-whitespace?`. Nếu sử dụng các dấu chấm câu trong một văn bản thì chỉ cần bổ sung thêm các ký tự phân cách.

```
(char-whitespace? #\space)
--> '#t
```

VI.4.3. Ghi lên tệp

1. Các hàm ghi lên tệp

Các hàm ghi lên tệp được liên kết với một dòng ra :

```
(write s [flow])
(display s [flow])
(newline [flow])
(write-char char [flow])
```

Sau khi thực hiện ghi lên tệp, các hàm trên trả về một giá trị không xác định và không đưa gì lên màn hình. Trong Scheme, việc ghi lên tệp bởi `display` làm người đọc dễ đọc kết quả, nhưng ghi lên tệp bởi `display` thì Scheme lại dễ đọc (dễ xử lý) hơn. Ta hãy so sánh các lời gọi sau :

```
(write #\n)
--> '#\n
(display #\n)
--> n
(write  "tom and jerry")
--> "tom and jerry"
(display  "tom and jerry")
--> tom and jerry
```

Như vậy `write` giữ nguyên tất cả các ký tự đặc biệt của chuỗi khi ghi lên tệp. Scheme còn có hàm ghi lên tệp từng ký tự trong dòng ra. Giả sử ta cần tạo ra trong thư mục hiện hành tệp `"mailto.txt"` có nội dung sau :

```
(define flow (open-output-file "mailto.txt"))
(display "University of Danang" flow)
```

```
(newline flow)     ; ghi ký tự qua dòng
(display
  "41, Le Duan St., Danang City, Vietnam" flow)
(newline flow)     ; ghi ký tự qua dòng
(display
  "Tel: (84.511) 822041, 832678 - Fax: (84.511) 823683"
  flow)
(close-output-port flow)
```

Gọi hàm `readfile` để xem lại nội dung :

```
(readfile "mailto.txt")
University of Danang
41, Le Duan St., Danang City, Vietnam
Tel: (84.511) 822041, 832678 - Fax: (84.511) 823683
```

Khi tạo tệp, để ghi ký tự qua dòng, ta cũng có thể sử dụng hàm `write-char` :

```
(write-char #\newline flow)
```

2. Lệnh sao chép tệp

Sao chép tệp (copy) thường xuyên được sử dụng khi làm việc với tệp. Ta xây dựng hàm `copy-file` tương tự các hàm đọc tệp `read-file`. Hàm sao chép tệp sẽ xử lý lần lượt từng ký tự từ đầu tệp đến cuối tệp, cho đến khi gặp ký hiệu *eof*. Trong hàm, ta dùng một vòng lặp dạng `do` để duyệt tệp.

```
(define (copy-file source target)
   (call-with-input-file source
     (lambda (input-flow)
       (call-with-output-file target
         (lambda (output-flow)
             (do
               ((char
                    (read-char input-flow)
                    (read-char input-flow)))
               ((eof-object? char))
               (write-char char output-flow)))))))
(copy-file "mailto.txt" "uodn.txt")
#t
```

Khi cần làm việc với các tệp chứa các hàm Scheme, ta có thể thay thế việc đọc từng ký tự bởi đọc nguyên một s-biểu thức. Tuy nhiên, theo cách này, các dòng chú thích sẽ không được sao chép. Một cách tổng quát, nếu ta có một tệp được tổ chức theo cấu trúc dạng khối, ta có thể sao chép từng khối bằng cách sử dụng chức năng đọc hoặc ghi trên khối. Sau đây là một chương trình thực hiện sao chép trên tệp theo khối :

```
(define (copy-flow flow-in flow-out rd-block wr-block)
   (letrec
    ((loop
       (lambda()
           (let ((block (rd-bloc flow-in)))
              (if (eof-object? block)
                  'undefine
                  (wr-bloc bloc flow-out))))))
    (loop)))
```

VI.4.4. Giao tiếp với hệ thống

Quá trình giao tiếp giữa NSD với hệ thống phụ thuộc cách cài đặt các trình thông dịch của Scheme. Tuy nhiên, Scheme chuẩn có hàm `load` cho phép tải một tệp chứa các chương trình Scheme vào bộ nhớ để sẵn sàng gọi chạy. Cú pháp của hàm `load` như sau :

```
(load filename)
```

Ví dụ : Giả sử tại thư mục hiện hành có chứa tệp chương trình lời giải bài toán «Tháp Hà Nội», NSD nạp vào bộ nhớ và gọi chạy như sau :

```
(load "hanoi.scm")
--> hanoi.scm
(hanoi 3 #\A #\B #\C)
     Move one disk from A to B
     Move one disk from A to C
     Move one disk from B to C
     Move one disk from A to B
     Move one disk from C to A
     Move one disk from C to B
     Move one disk from A to B
```

Chú ý rằng giá trị trả về của `load` không ảnh hưởng đến giá trị trả về của `current-input-port` và `current-output-port`.

Bài tập chương 6

1. Cho biết giá trị trả về từ :

```
((lambda (a)
(let ((a 1))
   ((f (lambda (x) (+ a x))))
   (f a))) 5)
--> ?
```

2. Vẽ các khối liên kết cho chầu làm việc sau đây

```
(let*((x 2) (y (+ x x)) (z (* x y)))
(let ((x y) (y x) (+ *))
   (+ x y z)))
```

   Chỉ ra liên kết hiện hành của các biến trong `(+ x y z)` ?

3. Tại sao không thể dùng `car` thay vì `head` nếu định nghĩa lại `car` trong phần trình bày lý thuyết ? Giải thích vì sao người ta không thể sử dụng định nghĩa đây :

```
(define car
    (lambda (L)
        (if (null? L)
            '()
            (car L))))
```

4. Viết hàm :
   a) Tính số lượng các ký tự trong một tệp ?
   b) Tính số lượng các ký tự không phải là ký tự phân cách (không phải là các dấu cách, nhảy cột, hay qua dòng) ?
5. Viết hàm so sánh hai tệp văn bản để trả về dòng văn bản đầu tiên khác nhau giữa hai tệp ?
6. Từ bốn số nguyên dương *A*, *B*, *C*, *D* sao cho $1 \le A \le 31$, $1 \le B \le 12$, $1000 \le C \le 2000$, $1 \le D \le 7$ (ý nghĩa của bốn số này là *ngày A tháng B năm C là ngày thứ D*, trong đó *D*=1 là ngày Chủ nhật, *D*=2 là ngày thứ Hai, ... *D*=7 là ngày thứ Bảy) và ba số nguyên dương *A1*, *B1*, *C1* thoã mãn các điều kiện tương ứng đối với *A*, *B*, *C*, hãy cho biết ngày *A1* tháng *B1* năm *C1* là ngày thứ mấy ? Chú ý đến các năm nhuần.
7. Cho số nguyên dương *n* và biến thực *x* thay đổi từ 0 đến 1 với bước tăng là *h* = 0.01.

   Hãy tính tích phân : $I = \int_0^{\frac{\pi}{2}} \sin^n x \, dx$

   Với các giá trị của *n* được tính theo công thức :

$$I = \begin{cases} \frac{1}{2}.\frac{3}{4} \ldots \frac{n-1}{n}\frac{\pi}{2} & \text{với } n \text{ chẵn, } n \ge 2 \\ \frac{2}{3}.\frac{4}{5} \ldots \frac{n-1}{n} & \text{với } n \text{ lẻ, } n \ge 3 \end{cases}$$

8. Từ danh sách 5 số nguyên a_0, a_1, a_2 , a_3, a_4, hãy trả về mọi nghiệm nguyên có thể của phương trình đa thức bậc 4 hệ số nguyên :

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 = 0$$

9. Cho số thực *x* thay đổi từ 0 đến 1 với bước tăng là *h* = 0.01 và *n* nguyên dương, bằng cách sử dụng cả hai phương pháp đệ quy và lặp, tính giá trị của đa thức Tsebusep bậc *n* được cho bởi công thức truy hồi như sau :

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \qquad \text{với } n = 2, 3, 4, \ldots$$

   Trong đó : $T_0(x) = 1$, $T_1(x) = x$.

$$U_{n+2}(x) = xU_n+1(x) - 0.25U_n(x) \qquad \text{với } n = 0, 1, 2, \ldots$$

   Trong đó : $U_0(x) = 1$, $U_1(x) = x$.

10. Cho số thực *x* thay đổi từ 0 đến 1 với bước tăng là *h* = 0.01 và *n* nguyên dương, bằng cách sử dụng cả hai phương pháp đệ quy và lặp, tính giá trị của đa thức Legendre bậc *n* được cho bởi công thức truy hồi như sau :

$$L_{n+2}(x) = xL_{n+1}(x) - \frac{(n+1)^2}{(2n+1)(2n+3)} L_n(x) \qquad \text{với } n = 0, 1, 2, \ldots$$

    Trong đó : $L_0(x) = 1$, $L_1(x) = x$.

11. Cho số thực *x* thay đổi từ 0 đến 1 với bước tăng là *h* = 0.01. Tính tổng :

$$S = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^n}{n!} + \ldots$$

    với độ chính xác ε cho trước, chẳng hạn $\varepsilon = 10^{-5}$

12. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$ và n nguyên dương. Tính giá trị :

$$y = \sqrt{x + \sqrt{x + \ldots + \sqrt{x}}} \quad \text{có } n > 1 \text{ dấu căn}$$

13. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$ và n nguyên dương. Tìm nghiệm phương trình vi phân :

$$f(x) = \cos x - \frac{1}{x}$$

biết rằng nghiệm ở giữa các điểm 0 của hàm $\cos x$, nghĩa là $\frac{3\pi}{2}$, $\frac{5\pi}{2}$, ...

Dùng phương pháp lặp Newton : $x^{n+1} = x^n - \frac{f(x^n)}{f'(x^n)}$, với $f'(x)$ là đạo hàm của f.

14. Dãy Fibonaci được định nghĩa như sau :

$$f_0 = f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \text{ với } n \geq 2$$

Viết vị từ kiểm tra một số nguyên m cho trước có phải là số Fibonaci không ?

15. Cho số n nguyên dương, hãy tìm giá trị a_n được xác định từ dãy số nguyên $a_1, a_2, ..., a_n$ như sau :

$$a_1 = 5$$
$$a_2 = 8$$
$$a_n = a_{n-1} + 3 * (n-1)$$

16. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$. Tính tổng :

$$1 + \frac{x^2}{2} + \frac{2}{3}\times\frac{x^4}{4} + \frac{2}{3}\times\frac{4}{5}\times\frac{x^6}{6} + \ldots$$

với độ chính xác ε cho trước, chẳng hạn $\varepsilon = 10^{-5}$

17. Tìm tất cả các số có 3 chữ số sao cho tổng lập phương của các chữ số bằng chính số đó. Chẳng hạn gọi a, b, c là 3 chữ số của số cần tìm, khi đó điều kiện trên đươc mô tả :

$100\times a + 10\times b + c = a\times a\times a + b\times b\times b + c\times c\times c$

Ví dụ $153 = 1^3 + 5^3 + 3^3$.

18. Nhập vào một số nguyên ngẫu nhiên bất kỳ để in ra theo mọi hoán vị của các chữ số có thể trong số đó. Ví dụ nhập vào 195 thì in ra 159, 915, 951, 519, 591.

19. Cho trước một tháng bất kỳ trong năm, hãy cho biết tháng này thuộc quí nào ? Tháng này có bao nhiêu ngày ? Cho trước một ngày bất kỳ trong năm *dd/mm/yyyy*, hãy cho biết số ngày kể từ ngày đầu năm 01/01/*yyyy* (chú ý năm nhuần) ?

20. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$. Tính tổng :

$$x + \frac{1}{2}\times\frac{x^3}{3} + \frac{1}{2}\times\frac{3}{4}\times\frac{x^5}{5} + \frac{1}{2}\times\frac{3}{4}\times\frac{5}{6}\times\frac{x7}{7} + \ldots$$

với độ chính xác ε cho trước, chẳng hạn $\varepsilon = 10^{-5}$.

21. Viêt chương trình tính số π với độ chính xác 10-4 biết rằng :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots \quad \text{cho đến khi } \left|\frac{1}{2n-1}\right| < 10^{-4}$$

22. Cho x thực. Tinh tổng :

$$S = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \ldots + \frac{x^{2n}}{(2n)!} + \ldots \quad \text{cho đên khi } \left|\frac{x^{2n}}{(2n)!}\right| < 10^{-5}$$

23. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$. Hãy tính :

$$f(x) = e^{-x} \sin(2px)$$

$$g(x) = e^{-2x} \sin(2px) + \cos(2px)$$

24. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$. Xây dựng hàm log(a, x) tính logarit cơ số a của x. Dùng hàm này tính giá trị của biểu thức sau :

$$y = \begin{cases} \log_{1-x^2}(1+x^2) & \text{khi } |x| < 1 \\ 0 & \text{khi } |x| = 1 \\ \log_2(x^2-1) & \text{khi } |x| > 1 \end{cases}$$

25. Dãy số nguyên a_0, a_1, a_2... a_n,... xác định bằng quy nạp như sau :

$a_0 = 1$

$a_n = n\, a_{n-1}$ nếu n chẵn ($n = 2, 4, 6...$)

$a_n = n + a_{n-1}$ nếu n lẽ ($n = 1, 3, 5...$)

Cho trước n, hãy tìm giá trị a_n.

26. Tìm nghiệm của hệ phương trình :

$ax + by + c = 0$

$px + qy + r = 0$

Chú ý trả về : «*không xác định*» nếu aq - bp = 0

và «*phụ thuộc tuyến tính*» nếu $\frac{a}{p} = \frac{b}{q} = \frac{c}{r}$.

Các hệ số được cho dạng danh sách ($a\ b\ c\ p\ q\ r$).

27. Cho các danh sách phẳng, số là các vectơ n chiều X, Y và Z :

$X = (x_1, x_2, ..., x_n)$

$Y = (y_1, y_2, ..., y_n)$

Tính tích vô hướng của hai vectơ X và Z, với vectơ Z có các thành phần z_i, $i = 1..n$, được xác định như sau :

$z_i = 0$ nếu $0 < x_i + y_i < A$, $A > 0$

$x_i + y_i$ nếu $x_i + y_i \geq A$

$(x_i + y_i)^2$ nếu $x_i + y_i \leq 0$.

28. Tính đa thức Hermit $H_n(x)$ theo các phương pháp sơ đồ đệ quy và sơ đồ lặp :

$H_0(x) = 1$

$H_1(x) = 2x$

$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$

29. Cho số thực x thay đổi từ 0 đến 1 với bước tăng là $h = 0.01$ và n nguyên dương, bằng cách sử dụng cả hai phương pháp đệ quy và lặp, tính giá trị của đa thức Laguerre bậc n ($a = 0$) được cho bởi công thức truy hồi như sau :

$$L_{n+2}(x) = (x - 2n - 3)L_{n+1}(x) - (n + 1)^2 L_n(x) \qquad \text{với } n = 0, 1, 2, ...$$

trong đó : $L_0(x) = 1$, $L_1(x) = x - 1$.

30. Các nghiệm của phương trình bậc ba quy về dạng chính tắc : $x^3 + px + q = 0$

được tính theo công thức :

$$\varphi = \frac{1}{3}\text{Arc}\cos\left(\frac{-q}{2\sqrt{-\frac{p^3}{27}}}\right) \qquad A = 2\sqrt{-\frac{p}{3}}$$

$$x_1 = A\cos\varphi$$

$$x_2 = A\cos\left(\varphi + \frac{2\pi}{3}\right)$$

$$x_3 = A\cos\left(\varphi - \frac{2\pi}{3}\right)$$

Đọc vào n nguyên dương và các cặp giá trị p_i, q_i, $i = 1..n$.

Đưa ra các giá trị nghiệm tương ứng x_{1_i}, x_{2_i} và x_{3_i}.

31. Cho N là một số dương, tính căn bậc hai $\sqrt{N}$ theo phương pháp Newton (không sử dụng hàm `sqrt`). Nội dung phương pháp : Nếu gọi s là giá trị gần đúng của căn bậc hai của n thì $0.5*\left(\frac{N}{s} + s\right)$ là giá trị gần đúng hơn của N. Quá trình tính toán dừng lại khi đạt tới độ chính xác ε cần thiết. Giả sử chọn $\varepsilon = 10^{-6}$, khi đó : $\left|\frac{N}{s^2} - 1\right| < \varepsilon$