**FUNCTION PROGRAMMING**

**Lambda expression**: là hàm ẩn danh. Có đầy đủ đặc điểm của một hàm như tham số (parameters) và nội dung thực thi (body). Có thể áp dụng các tham số vào biểu thức lambda bằng cách đặt nó sau biểu thức

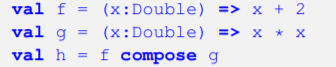
(lambda(x) x \* x \* x)(2) = 8

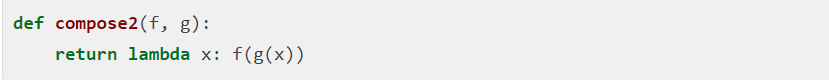
**High order function**: là hàm có thể nhận 1 hàm như là 1 tham số hoặc trả về 1 hàm như là kết quả hoặc cả hai

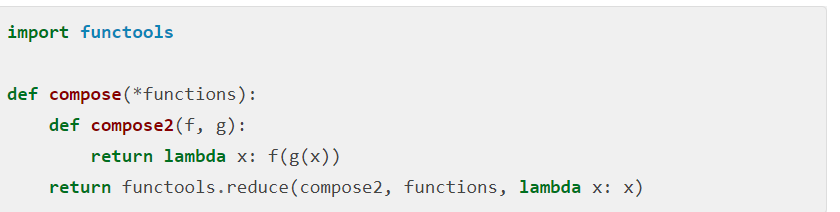
- **Function composition (hàm hợp)**

Nhận 2 hàm như 2 tham số và trả về 1 hàm mà giá trị của nó là giá trị của hàm thứ nhất áp dụng vào hàm thứ 2

VD: 

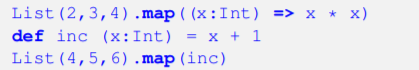
Scala: 

Python: 

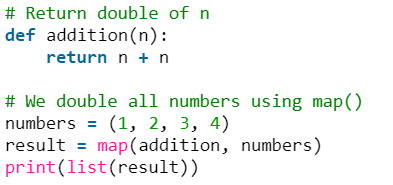


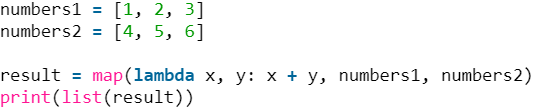
- **Apply-to-all**

Nhận 1 hàm như là 1 tham số và nhận 1 danh sách các tham số và trả về 1 danh sách các giá trị bằng cách áp dụng hàm vào mỗi tham số trong danh sách (map)

Scala: 

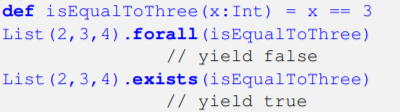
Python: map(fun, iter)





- **Forall/exists**

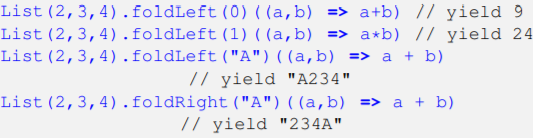
Nhận 1 hàm vị từ(true/false) như 1 tham số và 1 danh sách các tham số và trả về giá trị sau khi áp dụng hàm vào danh sách và nhận and/or của kết quả.

Scala: 

Python:  all/any

- **Insert left/insert right**

Fold left/fold right



- **Filter**

Nhận vào 1 danh sách và 1 hàm, lọc các giá trị thỏa mãn hàm và trả về 1 danh sách con

List(2,3,4,5,6).filter(x => x%2 == 0)

even = filter(lambda x: x % 2 == 0, integers)

- **Funtion as parameters**

Trong hàm do người dùng tự định nghĩa, hàm có thể đưa vào như là tham số

- **Closure**

An object is data with functions. A closure is a function with data

- **Curry function**

from functools import partial

def multiply(x,y):

return x \* y

# create a new function that multiplies by 2

dbl = partial(multiply,2)

print(dbl(4))

**(left to right)**

**Immutability: cannot change**

Java string: toUpper() return new string not change old string

Scala: val is immutable

Pure functional programming: no mutations -> inherently parallelizable

**Pattern matching**

Like switch case but more powerful

Recursion

**Lazy evaluation**

lazy val x = 1 + y : 1 + y đc sử dụng chỉ khi x đc dùng lần đầu

**Type Inference**

Kiểm tra kiểu tĩnh -> chặt chẽ

Có thể override kiểu suy diễn

Phải khai báo kiểu tham số

Kiểu trả về ko cần định nghĩa

Ngoại lệ đệ quy cần quy định kiểu trả về

**C3 tuyến tính**

**Tên / ràng buộc / tầm vực**

**Tên**: chuỗi kí tự biểu diễn

Định danh (identifiers)

Toán tử(operators)

Dùng kí tự thay cho địa chỉ để biểu diễn thực thể

Tính trừu tượng

**Ràng buộc (binding)**

**Alias(bí danh):** nhiều tên ràng buộc cùng 1 thực thể

VD: int \*p = new int

int \*q = p

New int tạo 1 đối tượng có thể có nhiều tên đó là \*p, \* q

**Binding time**: Thời gian xảy ra ràng buộc

Language design time(TG thiết kế ngôn ngữ): do người thiết kế ngôn ngữ

Language implementation(TG hiện thực ngôn ngữ): trong thời gian viết chương trình dịch/ không có trong design time

Programming time(TG lập trình) ràng buộc của người lập trình

Complication time(TG dịch) ràng buộc thực hiện trong thời gian dịch

Linking time(TG liên kết) liên kết các file object -> các địa chỉ tương đối

Load time(TG nạp) bộ nhớ ngoài -> bộ nhớ trong / nạp từ file (vd địa chỉ biến tuyệt đối trong bộ nhớ)

Runtime (TG chạy)

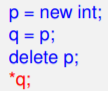
**Object lifetime**: khoảng tg giữa lúc object đc tạo và object bị phá hủy

Binding lifetime: thời gian sống của ràng buộc/ thời gian mà ràng buộc đc tạo ra cho đến khi nó bị hủy.

Leak memory - Garbage(rác) đối tượng không thể truy xuất



Dangling reference(tham chiếu treo) đối tượng dữ liệu bị mất nhưng tên vẫn còn



**Object allocation:**

Static: Vùng nhớ tĩnh ngay khi dịch

Lifecycle: từ khi chương trình chạy cho đến khi chương trình kết thúc -> biến toàn cục, hằng, chuỗi

Stack dynamic: dùng chung giữa stack và static

Stack: LIFO -> chỉ cần quản lí đỉnh -> biến cục bộ

Heap: dùng chung stack

Explicit heap dynamic: tạo ra đối tượng dùng lệnh(tường minh)

Implicit heap dynamic: tạo ra đối tượng theo suy diễn -> ko theo quy luật rõ ràng

**Blocks**

Early binding: compile time

Late binding: refer to function calls that are not resolved until runtime -> virtual functions

Static binding

Dynamic binding

**Tầm vực (Scopes)**

Là vùng văn bản mà ràng buộc đó có hiệu lực

VD: khai báo -> ràng buộc tên và kiểu

**Static scope(lexical scope)** Khai báo quyết định vào thời gian dịch

Current binding + global scope + local static scope

**Dynamic scope:** được quyết định vào thời gian thực thi

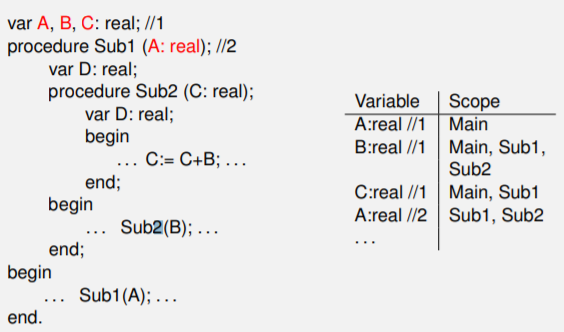
**STATIC SCOPE RULES**

- tham khảo tới identifier luôn ràng buộc với khai báo local nhất

- 1 khai báo ko thể thấy ngoài block nó xuất hiện

- khai báo trong các block lồng nhau nhìn thấy đc bởi block nằm trong trừ khi nó đc khai báo lại

- block có thể đc đặt tên/ tên của block thuộc về block bên ngoài (vd function)

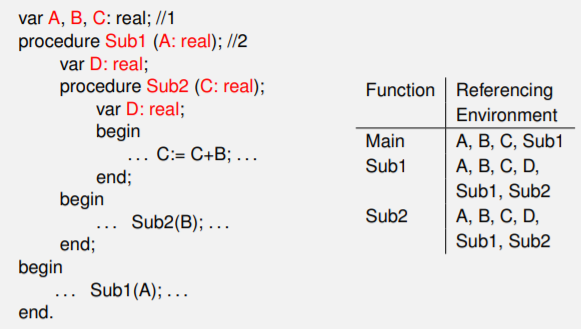
Static:

Dynamic: the compiler first searches the current block and then successively all the calling functions. (gần nhất)

**Referencing environment: MT tham khảo của 1 phát biểu (statement)**

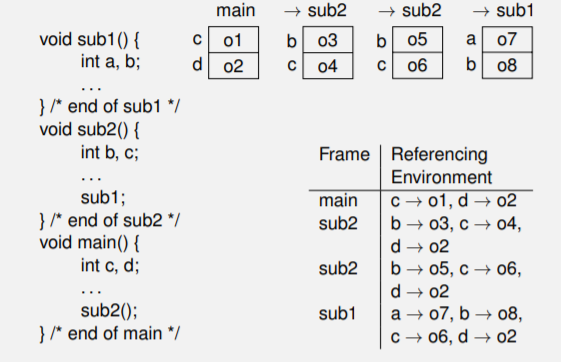
Tập hợp tất cả các tên mà nhìn thất đc bởi statement đó

- Static: tất cả local name cộng với tất cả biến visible với tầm vực bao tầm vực đó



- Dynamic: MT tham khảo: ràng buộc local -> tất cả ràng buộc nhìn thấy đc của chương trình con đang hoạt động (active)

A subprogram is **active** if its execution has begun but has not yet terminated



**Avoid dangling references**

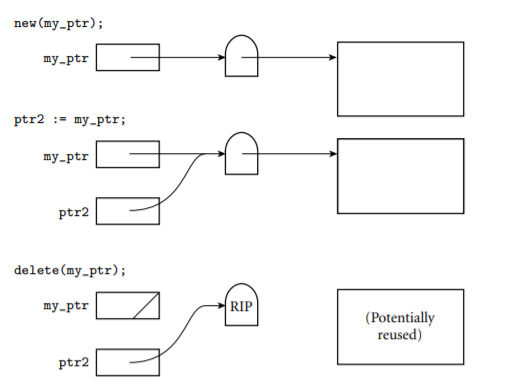
TOMBSTONE:

Khi đối tượng đc cập phát vùng nhớ trong heap/ khi con trỏ đc khởi tạo đến đối tượng trong stack -> tạo ra 1 tombstone

Biến con trỏ ko trỏ trực tiếp đến object mà thực tế chỉ trỏ vào tombstone, tombstone chứa địa chỉ đến object

Khi object giải phóng, tombstone vẫn giữ lại nhưng đc thiết lập giá trị null

Khi so giá trị nếu gặp null từ tombstone -> tham chiếu ngưng tránh tham chiếu treo



**Lock and keys**

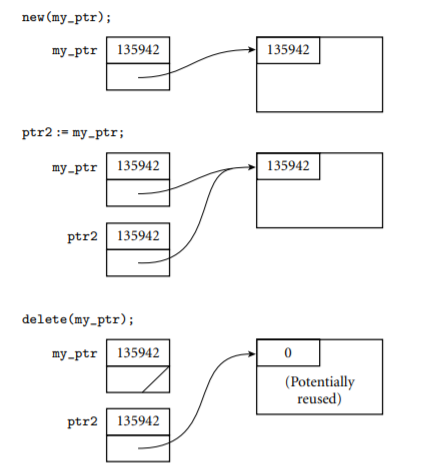
Các giá trị con trỏ đc đại diện cho các cặp (key, address)

Biến con trỏ chứa thêm giá trị khác

Đối tượng được con trỏ trỏ đến cần thêm 1 vùng nhớ

Khi tham chiếu, để sử dụng giá trị của đối tượng đc trỏ đến cần so khớp (match key)

Khi delete, 1 thành phần của đối tượng đc trỏ đến thay đổi (lock) vì vậy con trỏ tham chiếu treo ko match key -> ko tham chiếu đc.



**Data types : tập hợp các giá trị đồng nhất với nhau**

**Scalar types**

Atomic(nguyên tử)

Dùng để tạo thành các kiểu dữ liệu khác

Đôi khi đc hỗ trợ bởi phần cứng

Boolean, char, int, floating point

**INTEGER**

Hỗ trợ nhiều kích thước kiểu nguyên: byte, short, int, long

Một số ngôn ngữ hỗ trợ kiểu ko có dấu

Hỗ trợ bởi phần cứng chuỗi bits

**Biểu diễn số âm: số bù 2**

**FLOATING-POINT**

Mô hình hóa số thực nhưng chỉ xấp xỉ

Ngôn ngữ dùng cho khoa học

Hỗ trợ 2 kiểu float, double

**Chuẩn IEEE 754**

**DECIMAL**

Dùng cho ứng dụng tài chính

Chứa số chữ số thập phân ko đổi

Ưu: chính xác / nhược: tốn bộ nhớ, giới hạn range

**BOOLEAN**

Đơn giản, thường hỗ trợ trên bytes, có thể trên bits

**CHAR**

Chứa ở dạng mã hóa

Thường là ASCII

16-bit: unicode

**User defined ordinal types**

**Enumeration types (kiểu liệt kê)**

Dễ đọc hơn

Tính tin cậy

Tác vụ bị giới hạn -> tránh nhầm lẫn

Gán giá trị cho tập danh hiệu đã định nghĩa -> ko nằm ngoài miền cho phép

Không bị ép về kiểu nguyên

Hiện thực thành những giá trị nguyên

**Subrange types (kiểu miền con)**

Định nghĩa 1 miền của kiểu ordinal

Biểu hiện như kiểu cha, có thể dùng cho biến/ mảng

Là kiểu cha vs đoạn mã đc chèn thêm để hạn chế gán biến vào miền con

**Composite types**

Chứa nhiều thành phần, có thể truy cập đến những thành phần 1 cách độc lập

Các thành phần có thể giống nhau (homogenerous) hoặc khác nhau (heterogenerous)

VD: STRUCT (khác nhau)

Số lượng thành phần cố định hoặc thay đổi

Có thể có tác vụ trên toàn đối tượng hoặc trên thành phần

VD: gán dãy, phép toán trên tp

Có thể có tác vụ thêm/bớt (số lượng các thành phần thay đổi)

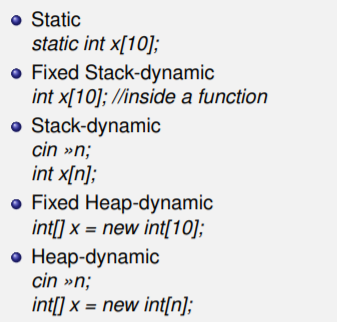
Kích thước lớn -> có tác vụ khởi tạo/ hủy bỏ

**Array types**

Tập hợp các thành phần giống nhau

Mỗi thành phần đc phân biệt bằng vị trí liên hệ với phần tử đầu tiên và sử dụng bởi chỉ mục

**Subscript binding and array categories**



**Jagged array/ rectangular array**

**Slices**

Một bộ phần của array, dùng trong các ngôn ngữ có tác vụ trên array

VD python

**Implementation of arrays**

Đặt các phần tử trên vùng nhớ -> hàm biến đổi chỉ số thành địa chỉ.

Dãy 1 chiều: xếp trên bộ nhớ theo thứ tự tăng dần

Address(list[k]) = address(list[lower bound]) + ((k-lower bound) \* element size)

Địa chỉ phần tử đầu + kích thước tất cả phần tử nằm trước phần tử k

Dãy 2 chiều: row-major order (chỉ số thấp tăng chậm) trong hầu hết ngôn ngữ

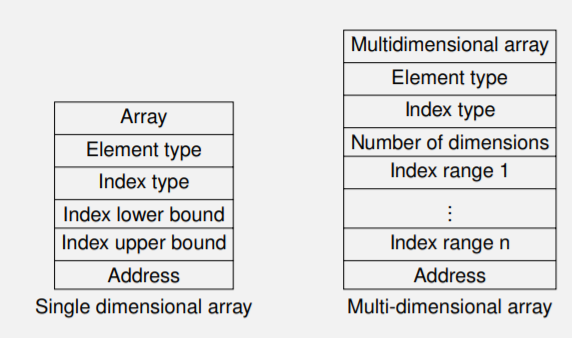
Column-major order

Tính địa chỉ (row-major order)

Location(a[I;j]) = alpha + ((i-row\_lb)\*n + (j-col\_lb)) \* E

Với alpha là địa chỉ của a[row\_lb,col\_lb] và E là element size

**Compile time descriptors**



**Associative array**

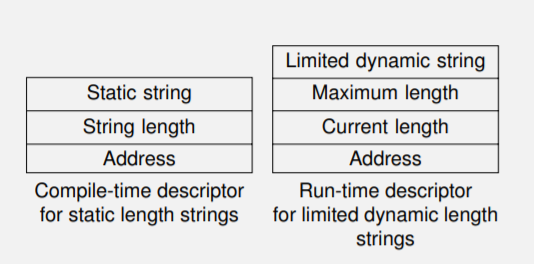
**String types**

Giá trị là chuỗi các kí tự

Tác vụ cơ bản: gán, so sánh chuỗi, nối, chuỗi con, tìm chuỗi con trong chuỗi, so trùng mẫu (biểu thức chính quy)

Chiều dài chuỗi: Static: cố định vào tg thực thi / dynamic: có giới hạn: có thể thay đổi nhưng giới hạn; không giới hạn.

Descriptors



**Record types(bản ghi)**

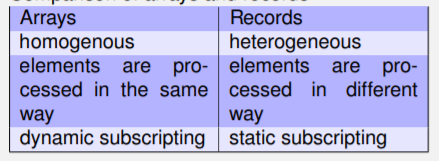
Kết hợp nhiều kiểu thành phần khác nhau

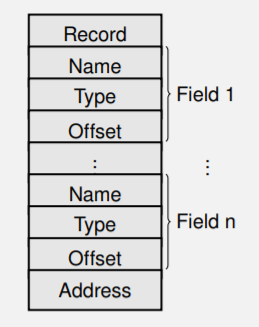
Các thành phần độc lập và đc phân biệt bởi tên

Reference: dot notations / keyword based

Format: fully qualified references: bao gồm tất cả record names/ elliptical references: có thể truy cập record names nếu ko bị nhập nhằng

Operation: assignment:copy vùng nhớ qua vùng nhớ khác/ so sánh/ khởi tạo





**Data alignments**

Char: 1 byte

Short: 2 byte

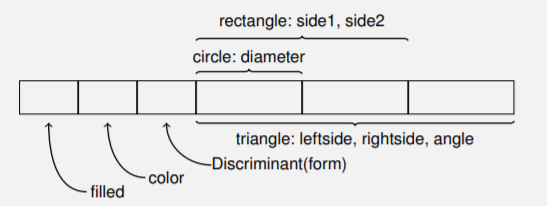
Int/ long/float: 4 bytes

Padding: khi 1 thành phần của cấu trúc sau 1 thành phần khác có alignment lớn hơn/ thêm vào cuối 1 cấu trúc để làm kích thước lớn hơn thành bội số của thành phần

**Data structures padding**

**Union types**

Kiểu union là một kiểu mà những biến trong kiểu này là đc cấp những giá trị kiểu khác nhau ở những thời điểm khác nhau trong suốt quá trình thực thi. Do đó vùng nhớ cấp phát cho một kiểu union được dùng chung cho các thành phần của nó.

****

**Set types**

Biểu diễn tập hợp

Có các tác vụ trên tập hợp như: xác định phần tử thuộc tập hợp, giao, hợp, hiệu

**Pointer types**

Biến kiểu con trỏ nhận 1 khoảng các giá trị chứa địa chỉ ô nhớ và 1 giá trị đặc biệt(nil)

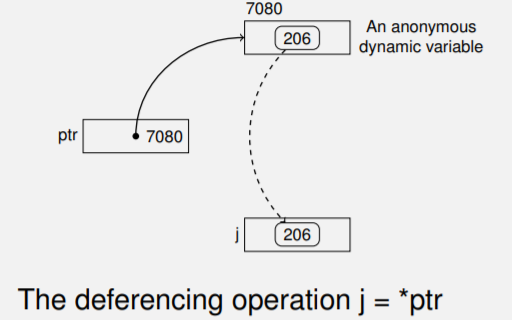
Cho khả năng truy xuất gián tiếp

Quản lí bộ nhớ động: con trỏ dùng để truy xuất vùng nhớ đc tạo ra động (heap)

Operations:

Gán (assignment) lấy giá trị biến con trỏ này gán cho địa chỉ khác

Tham chiếu(deferencing) dựa vào giá trị con trỏ tìm vị trị giá trị con trỏ chỉ đến (bằng \*)



Bản chất: biến con trỏ ptr cất địa chỉ 7080 của giá trị 206

Ngược lại \*ptr = j -> cất giá trị của j vào địa chỉ 7080 mà ptr trỏ đến

Ptr: 7080

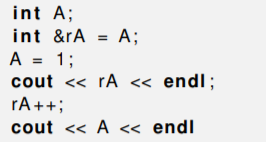
\*ptr: 206

&j = 123

J = 206

**Reference types**

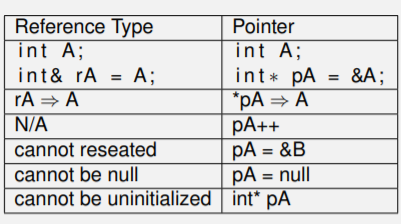
Kiểu tham chiếu không chứa giá trị trực tiếp mà chứa địa chỉ mà giá trị đc chứa.



Truy cập rA và A là như nhau

Pointer -> address

References -> object/value



**Recursive type**

**Type expression (biểu thức kiểu)**

Kiểu cơ bản là biểu thức kiểu

Tên kiểu là biểu thức kiểu

Khởi tạo kiểu áp dụng vào 1 biểu thức kiểu là 1 biểu thức kiểu

Array(I,T) I : index trype; T: element type

Product: T1 x T2

Records: record(name1 x T1) x (name2 x T2) x …)

Pointer: pointer(t)

Function: T1 -> T2

Biến kiểu là 1 biểu thức kiểu

Int => int

Typedef int size => size

Int t[10] => array(0..9,int)

Int foo(int a, float b) => (int x float) -> int

Int\* p => pointer(p)

Template <class T> struct vd{T a; T b[3];} => Record((a x T) x (b x array(0..2,T)))

**Type checking**

Static type checking: compiling time

Dynamic type checking: runtime

**Type inference**

Khai báo ko cần khai báo đầy đủ, ko khai báo 1 số -> suy diễn kiểu cho có phần tử đó

Gán kiểu vào điểm lá trên cây ast

Sinh ra ràng buộc kiểu với mỗi node trên ast

Giải các ràng buộc kiểu

**Type equivalence**

Kiểm tra 1 kiểu có tương thích vs nhau k mà ko cần ép kiểu

Tương đương theo tên /tương đương cấu trúc (dùng cho static check)

**Type Compatibility**

Kiểu T tương thích vs kiểu S nếu T xuất hiện bất kì chỗ nào cho phép kiểu S ( 1 chiều)

VD int và float

Giá trị T là tập con giá trị S

Tất cả tác vụ trên S dùng đc trên T

Giá trị T tương ứng vs S

Giá trị T có thể biến thành giá trị S

**Type Conversion**

Ép kiểu

**Polymorphism**

Đơn hình: mỗi đối tượng ngôn ngữ chỉ có 1 kiểu

Đa hình: cùng 1 đối tượng có thể có nhiều hơn 1 kiểu

VD: int x int -> int or float x float -> float

**Điều khiển trình tự**

**Cấp biểu thức**

Biểu thức – đơn vị văn phạm có nhiệm vụ tạo 1 giá trị; nếu có sai sót -> undefined

Cơ chế tính toán – chồng chập hàm: tính toán các biểu thức con và truyền lên cho biểu thức phía trên

Expression syntax

Infix: nằm giữa các toán hạng -> tốt cho phép toán 2 ngôi -> dùng cho nhiều ngôn ngữ thủ tục (vấn đề khi có nhiều hơn 2 toán hạng)

Tính ưu tiên: ngôn ngữ tự định nghĩa tính ưu tiên dựa theo toán học -> những quy tắc ưu tiên khác nhau gây khó hiểu

Tính kết hợp: thường là left to right trừ exponentiation operator (lũy thừa)

Parentheses: thay đổi tính ưu tiên và tính kết hợp

Ưu: đơn giản / nhược: khó đọc; khó viết

Biểu thức điều kiện

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Prefix | | | Postfix | | |
| Polish prefix | Cambridge polish prefix | Normal prefix | Polish postfix | Cambridge polish postfix | Normal postfix |
| \*+ab-cd | (\*(+ab)(-cd)) | \*(+(a,b),-(c,d)) | ab+cd-\* | ((ab+)(cd-))\* | ((a,b)+,(c,d)-)\* |

**Tiền tố bình thường:** Viết theo dạng *toán tử \*(toán hạng, toán hạng)*

|  |  |
| --- | --- |
| (a + b) \* (c - d) | \*(+(a,b),-(c,d)) |

**Tiền tố Polish:** Đẩy hết dấu ra, toán hạng dính liền lại, bao nhiêu phép toán thì biễu diễn bấy nhiêu toán tử liền nhau

|  |  |
| --- | --- |
| a - b - c \* e \* f - g | ---ab\*\*cefg |

**Tiền tố Cambridge Polish:** Toán tử giống nhau được lượt bỏ, các phép toán ghi trong bộ () để biết cần có bao nhiêu toán hạng được thực hiên tương ứng với 1 toán tử.

|  |  |
| --- | --- |
| a \* b \* ( c - e - f) \* g | (\*ab(-cef)g) |

**Evaluation mechanism**

Eager evaluation: đầu tiên tính tất cả các toán hạng -> phép toán

a == 0 ? b : b/a -> lỗi chia 0

Lazy evaluation: truyền toán hạng chưa đc tính vào phép toán; phép toán chọn toán hạng nào cần tính trước -> chi phí cao hơn

Dùng lazy cho conditional và eager cho còn lại

Short-circuit evaluation: rút ngắn quá trình tính toán; nếu 1 toán hạng được tính toán là true cái thứ 2 sẽ đc rút gọn (vd: a == 0 || b/a > 2)

**Cấp phát biểu**

Là đơn vị văn phạm

Ko trả về giá trị

Thay đổi trạng thái của hệ thống

A := B -> A là address và B là value

Tính toán trái trước hay phải trước tùy người viết (vd c-based : xem phép gán vừa là biểu thức vừa trả về trạng thái vừa trả về giá trị)

**Control structures**

Control statement:

Chọn nhánh điều khiển khác nhau

Cho phép thực thi nhóm phát biểu nhiều lần

Two way selection: if them else

Dangling else : ko biết else thuộc if nào -> thêm block vào trong mọi trường hợp

Python: indentation matters

Multiple selection

Cho phép chọn 1 trong nhiều phát biểu hoặc nhóm phát biểu

Vd switch case

Iterative statement(phát biểu lặp) 1 hay nhiều phát biểu đc thực thi 0,1 hay nhiều lần

Counter-controlled loop(dùng biến đếm): cần có biến đếm / giá trị bắt đầu; kết thúc (initial/terminal) / step size(bước tăng)

Logically controlled loops (dùng bt luận lí): dựa trên kết quả luận lí thay đổi biến đếm

Iteration based om data structures (dựa trên cấu trúc dữ liệu) có iterator; gọi tại đầu mỗi vòng lặp; trả về 1 giá trị mỗi lần đc gọi (foreach)

Unconditional branching(nhảy ko đk) goto điều khiển trình tự dễ hơn nhưng nguy hiểm khó đọc, khó bảo trì ; ko tin cậy; khó tìm lỗi

**Control abstraction(trừu tượng hóa điều khiển)**

Điều khiển trình tự ở cấp đơn vị

**Subprogram definition**

Bao gồm:

Đặc tả : Tên chương trình con

Các thông số(parameters): input/output; order(thứ tự); type; cơ chế truyền

Hành vi của chương trình con (behaviour)

**Subprogram activation**

Đc tạo khi chương trình con đc gọi và bị hủy khi nó đã thực hiện xong

Gồm 2 phần: static part: code segment /

dynamic part: activation record(bản ghi hoạt động):

Thông số hình thức(formal parameters)

Dữ liệu cục bộ (local data)

Địa chỉ trả về(return address)

Other links

**Subprogram Mechanism**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Simple call- return | -không đệ quy(no recursion) | Lệnh gọi tường minh(explicit call site) | 1 điểm vào (single entry point) | Đơn thực thi(single execution) tại 1 thời điểm chỉ có 1 chương trình chạy | Kiểm soát truyền ngay lập tức(Immediate Control Passing) |
| Recursive call | Có thể gọi đệ quy  + trực tiếp(chính nó)  +gián tiếp(thông qua ct khác) | Lệnh gọi tường minh(explicit call site) | 1 điểm vào (single entry point) | Đơn thực thi(single execution) tại 1 thời điểm chỉ có 1 chương trình chạy | Tạo ra nhiều bản ghi hoạt động |
| Exception |  | Không có lệnh gọi tường minh |  |  | Dùng trong event-driven programming(hướng sự kiên) ; error handler |
| Coroutines |  |  | Có nhiều entry point |  | Có thể trì hoãn điều khiển và thực thi và quay lại caller; thực thi tiếp tại lúc trì hoãn |
| Task |  |  |  | Có thể xử lí song song với task khác | Chạy trên máy có đã xử lí(multiprocessor) hoặc trên máy đơn xử lí có time-sharing(~coroutines) |
| Scheduled subprogram |  |  |  |  | Thực thi của chương trình con ko bắt đầu lúc nó đc gọi  -> trì hoãn thời gian/trì hoãn thứ tự ưu tiên  Điều khiển bằng bộ định thời (scheduler) |

Exception mechanism: cần đặc tả

Loại biến cố nào đc xử lí và nó định nghĩa như thế nào

Làm thế nào để ném 1 biến cố

Làm thế nào để xử lí biến cố

Raising exception

By user interaction: click, text change

By OS

By an object(timer)

By programmer(throw)

**Parameter passing**

2 loại: formal parameters / actual parameters

2 loại formal-actual corresponding:

By position

By name

Phương pháp truyền

|  |  |  |
| --- | --- | --- |
| Input – output | Input only parameter(chỉ truyền vào) | Output only parameters |
| Pass by value-result(truyền bằng trị kết quả)  Đưa tham số thực -> tham số hình thức; sau khi kết thúc mới chép ngược lại | Pass by value  Tương tự nhưng chỉ chép vào  Khi kết thúc x,y bị hủy bỏ; a,b không thay đổi | Pass by result  Không truyền giá trị vào  Chỉ trả giá trị về a,b thay đổi |
| Pass by reference(truyền tham khảo)    x, y như là alias của a,b; thay đổi trực tiếp trên a và b  khi kết thúc x,y bị hủy bỏ | Pass by constant reference(truyền bằng tham khảo hằng)  Hằng -> ko đổi | As a result of function  Không có tham số thực: chỉ có trả về  Vd:  Int foo() … return 0;  foo() + 1 |
| Pass by name    Nói đến x chính là nói đến a |  |  |

**High order function**

Một hàm là 1 hàm bậc cao khi nó nhận hàm

như là tham số đầu vào

như tham số trả về( thường là functional programming)

trong dynamic binding:

**shallow binding: tìm từ môi trường của thằng gọi hàm đó gần nhất**

**deep binding: tìm từ môi trường của thằng khai báo và đặc tả hàm đó**

**JVM**

Stack-based machine : dựa trên stack != trên register

Stack chứa phép toán và ko trả về biểu thức

Pass argument và nhận giá trị trả về

Dễ hơn register-based

Data types

|  |  |
| --- | --- |
| Boolean | Z |
| Byte | B |
| Short | S |
| Int | I |
| Long | L |
| Char | C |
| Float | F |
| Double | D |
| Class reference | Lclass\_name |
| Interface reference | Linter\_name |
| Array reference | [[..[component-type |
| Void | V |
| Object | Ljava/lang/Object; |
| String | Ljava/lang/String; |
| String[] | [Ljava/lang/String; |
| Float[][] | [[F |
| Void main(String[] args) | ([Ljava/lang/String;)V |
| Char foo(float a,Object b) | (Fljava/lang/Object;)C |
| Int gcd(int a, int b) | (II)I |

**Method**

Operand stack:

Local variable array:

1 dãy biến cục bộ đc tạo mỗi lần phương thức đc gọi

Instance: (method của đối tượng)

chỉ số 0 gán vào this

các tham số khác tính từ 1 -> các biến

Class:

Tham số đánh từ 0

Sau đó đến các biến

1 slot chứa 1 giá trị boolean,byte,char,short,int,float,reference và return address

1 cặp slot chứa giá trị của long/ double

**Instructions**

**Code generation**

