Language Design issue

Reliability

Learning curve

Maintenance

Program development

Performance

Orthogonality

making decisions without worrying other decisions

Efficiency

Time, memory, network

Grammar

Problems

1. nonterminal not reachable from start

2. nonterminal never reach terminal string

3. ambiguous

different non-terminal for each precedence level

left/right associativity – left/right recursive

Language system

Just in time compilation

Some compilation down after running

Interpret & find part that run many times, compile it

Run and do profiling to find bottleneck, compile it

Binding

Language define – keywords

Language implementation – variable implementation

Compile – type + variable for statically typed language

Link – library

Load – memory address

early binding – faster & secure

late binding – runtime flexible

Functional programming

Motivation

1. Escape from order of evaluation

2. solve the bottleneck of RAM to CPU

Characteristics

No assignment, no side effects

Type

Set of possible value + operation on them

Design

Portability – int width, char signedness

Decision

Dynamic

flexible, easier for programmer

less efficient since need to keep runtime type info

Static

more reliable (less run time error)

mixed type checking

static fallback to dynamic when meet subtypes

dynamic allow type annotation to reduce overhead

Strongly typed vs weakly typed

Structure equivalence vs name equivalence

Type annotation

Supply type infor to language system + serve as doc

Subtype (subclass is subtype)

Have all the operation and value for super type

Polymorphism

Overload

Name mangling – functions for each type, compiler choose

Parameter polymorphism

Template – compile a copy of code for each type

+ optimize for each type

- have to keep copy

Generic – one copy of compiled code, check type at runtime

+ - above

Scoping

Definition – anything that establish binding for name

Scope of a definition = defining block - inner redefine block

Dynamic scoping – Callee access caller’s definition

Must look for binding at run time, less efficient

Static scoping – block (function) access definer’s space

Memory management

Static allocation activation record for each function

+ efficient

- only support one activation record of a function a time

Note

Array allocation

Static allocation

+ efficient

+ no crush due to memory exhaustion

+ compiler can optimize

- recompile when size change

- over allocate memory

- hard to have temp array

Stack static allocation

+ can have static array

Stack dynamic allocation

+ easy change size

+ less overallocation

+ can have temp array

- slow

- unreliable – negative size, stack overflow

Dynamic & static chain

Dynamic – caller base pointer

Static – definer base pointer

Compiler always know static chain

Nested function

When a f (who returns function g) is called, put it on heap

g has ep points to f (definer) on heap, when f return, can still find the variable defined there

+ can have higher order function

- heap allocation at function call

Garbage Collection

Conservative GC

Exclusion errors – should mark in use but didn’t

Ex. If a, b are pointer to heap stored as a^b, GC can’t

recognize, won’t work

Unused inclusion errors – mark something not used

Used inclusion errors – mark something used as other

Move heap blocks – no exclusion error or used inclusion error

Mark & sweep

+ tolerant both inclusion errors

- heap remains regimented

- freeze sometime

Copying collector

- sensitive to used inclusion error

- freeze sometime

Incremental collector

Collect some memory at a time

Reference count

+ Immediate free

- cycle

- need to change reference count

Scheme

Everything is object

Dynamic type checking

Static scoping

Call by value

Everything is represented as data, easy to write program generator

High level arithmetic – no overflow, no rounding error

Basic

#name(a b c …)

Vector

(cons 2 1) -> '(2 . 1)

(cons 'a '(1 2 3)) -> '(a 1 2 3)

(con? …) => test if can be result of con

(= Expr1 Expr2)

(> Expr1 Expr2)

…

(map expr1 expr2)

Map expr1 to expr2

(car '(+ 1 2 )) – ‘+

(cdr '(+ 1 2)) – ‘(1 2 )

cadr – car (cdr ..)

Control

(if *bool\_expr*

*true\_expr*

*false\_expr*)

(if *bool\_expr*

*(*begin *true\_expr1 true\_expr2)*

*(*begin *false\_expr1 false\_expr2)*

(cond

((*cond\_expr*) *expr*)

((*cond\_expr*) *expr*)

((*cond\_expr*) *expr*)

(else *expr*)

)

(and *expr exp*r …)

(or *expr expr* …)

Return first true

(not (*expr* …))

null?

empty?

zero?

pair?

symbol?

number?

string=?

List & pairs

Proper list (list) is pairs end with empty list

Improper list (pair) is just pairs doesn’t end with empty list

(cons ‘a (‘b . ‘c)) => (a b . c)

‘(a . (b . (c . ()))) = (a b c)

Dot

(define (print format argues) …)

Take one argument

print “*format*” ‘(…)

(define (print format **.** argues) …)

Take multiple argument as list (i.e. argues is a list of args)

print “*format*” …

List function

(define (list . X) X)

Let

(let ([var1 expr1] [var2 expr2] …) …)

(lambda (arg1 arg2 …) …)

(let ([x Expr1] [y Expr2]) …) => ((lambda (x, y) … Expr1 Expr2)

(let ([g (lambda (x . y) ‘(x y))]) (g 1 2 3 4)) => (1 (2 3 4))

(let ([f (let ([x 'sam])

(lambda (y z) (list x y z)))])

(let ([x 'not-sam])

(f 'i 'am))) => (sam i am)

Named & rec let

(letrec ([var1 expr1] [var2 expr2] …) …can call vars here…)

Can’t have dependency as below example

(letrec ([y (+ x 2)]  
         [x 1])  
  y)

(let name ([var1 expr1] [var2 expr2] …) …)

Equivalent to

(define (name var1 var2 …) …)

(name expr1 expr2)

(let name ([var1 expr1] [var2 expr2] …) …call name…)

Equivalent to

(letrec ([name (lambda (var1 var2) …)]) …call with exprs…)

Define

(define cadr

(lambda (x)

(car (cdr x))))

(define var0

(lambda (var1 ... varn)

e1 e2 ...))

(define (var0 var1 ... varn)

e1 e2 ...)

(define var0

(lambda vars

e1 e2 ...))

(define (var0 . vars)

e1 e2 ...)

(define-syntax and

(syntax-rules ()

((or) #f)

((or x) x)

((or x1 x2 …) (if x1 x1 (or x2 …))))

Similar to prolog rules

set!

do not create new bindings, as with let or lambda, but change the values of existing bindings

(set! *symbol* *value*)

(set-car! *list value*)

(set-cdr! *list list*)

commonly used to implement procedures that must maintain some internal state

(define next 0)

(define count

(lambda ()

(let ([v next])

(set! next (+ next 1))

v)))

closure

(define make-counter

(lambda ()

(let ([next 0])

(lambda ()

(let ([v next])

(set! next (+ next 1))

v)))))

(define count1 (make-counter))

(define count2 (make-counter))

(count1) <graphic> 0

(count2) <graphic> 0

(count1) <graphic> 1

(count1) <graphic> 2

------------------------------------------------

(define shhh #f)

(define tell #f)

(let ([secret 0])

(set! shhh

(lambda (message)

(set! secret message)))

(set! tell

(lambda ()

secret)))

(shhh "sally likes harry")

(tell) <graphic> "sally likes harry"

secret <graphic> exception: variable secret is not bound

Error handling

(assertion-violation *location\_of\_error string\_indicator other*)

(define reciprocal

(lambda (n)

(if (and (number? n) (not (= n 0)))

(/ 1 n)

(assertion-violation 'reciprocal

"improper argument"

n))))

Primitives

eq? – compare memory location

eqv? – dereference and compare

(let ([x (cons 'a 'b)])

(eqv? x x)) <graphic> #t

(eqv? (cons 'a 'b) (cons 'a 'b)) => #f

equal? – check recursively for number

= – only be used for numbers

Continuation

Call/cc – Call with current continuation

capture a computation, a state of the call stack as it were, and resume that same state later

(define (f …)

(**call/cc**

(lambda (continuation)

…

…(continuation *return\_value*)…

…

)

)

))

(define retry #f)

(define factorial

(lambda (x)

(if (= x 0)

(call/cc (lambda (k) (set! retry k) 1))

(\* x (factorial (- x 1))))))

(factorial 4) <graphic> 24

(retry 1) <graphic> 24

(retry 2) <graphic> 48

The continuation retry "Multiply the value by 1…then multiply this result by 4." If we pass the continuation a different value, we will cause the base value to be something other than 1.

Prolog

Variables – start with uppercase

Atom – constant we don’t evaluate

Arithmetic

X<Y, X>Y – X<Y, Y>Y

X=<Y – X<=Y

X>=Y – X >= Y

X =:= Y – X == Y

X =\= Y – X != Y

X is Expr

Expr must be bounded number (X Is 1 + Y => error)

X == Y – structural equal

X \== Y – structural unequal

X = Expr – unification

X \= Expr – success if there is no unification possible

Flexible structs

append(X, Y ,Z) – Z is [X, Y]

member(X, Y) – provable if list Y contain element X

nth(X, Y, Z) – provable if Z is Xth element in Y, start index 0

length(X, Y) – provable if X is list length Y

permutation(X, Y) – X and Y are permutation

prefix(X, Y) – if X is Y’s prefix

suffix(X, Y) – same

maplist(P, L) – map P to list L

Inflexible

reverse(X, Y) – Y is reverse of list X

sort(X, Y) – Y is sorted list of X

not(Expr) – true if Expr can’t be proved

IO & database

write(…)

read(X) – read to X

assert(X) – add to database

retract(X) – remove from database

Unification

Before unifying X->Expr, check X not occur in Expr

Cut

! – when called, success. When backtrack, fail

Finite Domain Solver

#=, #<, #>, #\= – add constrain onto variables

fd\_domain(X, L) – X only take variable in List L

fd\_domain(X, N1, N2) – X only take variable in [N1, N2]

fd\_domain\_bool(X) – X only take bool value

fd\_all\_different(L) – everything in list need to be different

fd\_labeling(…) – solve … according to constrains

ex.

?- X + Y #= 5.

X = \_#21(0..5)

Y = \_#39(0..5)

fd\_labeling([X, Y])

Match

[x , y | z] – x::(y::z)

Python

Multiple inheritance

Built-in method

a is b – compare address

type(a)

id(a)

isinstance(a, class)

Built-in types  
numbers

sequences

mappings

callables

misc

Name space

Class is an object that has a member \_\_dict\_\_ containing all the names associated with the class

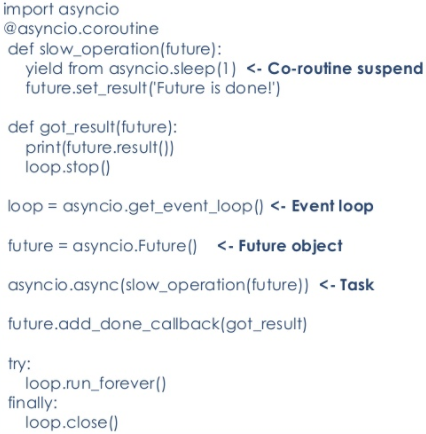
Import

1. create a namespace

2. read & execute source file in the namespace

3. create new names in caller’s namespace

Asyncio



execute procedural style code, and every time get to a yield from statement the execution of that procedural code is suspended.

This may go on for several levels of yield from call, but eventually a Future will be yielded and make its way back up to the Task, and we will start a new pass through the EventLoop

The EventLoop will then run any call\_soon callbacks. When all call\_soon callbacks have run, the EventLoop uses a selector to wait for the next IO event or the next callback that was scheduled to run at a specific time.

Those IO or timed events will provide values that will be set on certain Future objects, which will trigger the scheduling of call\_soon callbacks which will in turn cause the corouties that were waiting for those Futures to be scheduled via call\_soon to have next called on them and thus get another chance to run.

This continues until all Futures are complete, including the Task or Tasks that the main EventLoop is waiting for

**event loop** – each time through the loop, it calls any callback in the list of ‘ready’ callbacks (the call\_soon list), and then uses a selector to wait either for the next pending IO operation to complete or the time for the next scheduled task to arrive

**coroutine** – special function that can give up control to its caller without losing its state (not much memory compares to thread)

**future** – an object that represents the result of work that hasn’t completed, and also for a list of callbacks to be called when it is “done”

signalling by attaching callbacks to be scheduled for execution by the EventLoop when the Future‘s set\_result method is called

**Task** – a Future, and it wraps a coroutine. When a Task is created, it adds a callback to the EventLoop‘s call\_soon queue that starts the iteration of the coroutine it is wrapping. (arranges to call next on the coroutine)

Result to call to next:

Exception – the Task schedules a call\_soon callback with the EventLoop that, on the next pass through the loop, will throw the exception into the coroutine. This means that the exception will be raised at the point where the (innermost) yield from call was made.

Future – Task schedules a callback on the Future to call the Task when the Future has completed. When some other thread of control eventually causes the Future to move to the “done” state, the Future will schedule that callback to run. That callback in turn will schedule another call\_soon callback that will call next on the coroutine

StopIteration exception – the Task sets the value associated with the exception as its result via set\_result

The server is built using the Protocol interface as defined by the asyncio.Protocol base class. The class defines a set of callbacks which are called when an event occurs on the server, the most important of which is the data\_received(self,data) method. My definition of the Protocol class is passed into the loop.create\_method as a callable. For every connection made to the server, a new Protocol instance is created. The intent is that the Protocol is used to parse incoming data received by the connection, which is abstracted by the Transport class. In turn, the Transport handles any network I/O required by the Protocol.

The concept of the event loop as used with the Protocol is as follows: When a connection is made, a Protocol instance is created to serve that connection. After this, the

event loop reads data from that connection. This is asynchronously performed with reads from other connections and coroutines that have been scheduled. (See 3.5 WHATSAT and 3.6 Information Propagation (AT) ). Once all data from that connection has been received, the reader readies the associated callback. This is the reaction caused by the associated ‘event’ of a read finish. The event loop then executes the callback synchronously its next available iteration. Any Tasks scheduled follow a similar pattern: a portion of the Task is completed asynchronously, and then sends an event signal that informs the scheduler that it is ready to execute the next step of the task.

The use of “event-driven” to describe asyncio’s event loop From building this server infrastructure for a simple network, I can conclude that the asyncio library should be suitable for use in a lightweight serve within an application server herd in which servers receive and echo gps locations. This is because of strong loop performance in what is perceived to be many similar asynchronous tasks which do not depend on each other or perform deep coroutine yield froms. It is also easily expandable as new message parsing can be added by simply adding another branch within the data\_received callback of the server protocol. Last but not least, Python features native support for generators and coroutines, which should allow for faster performance as opposed to other non-python asyncio implementations. Memory wise, Python

refence count based garbage collection should ensure that unused objects are immediately made available as memory space. This is especially helpful as Protocol instances are constantly being created and destroyed. New Protocol instances can occupy spaces left by dead Protocol objects without much respacing required.

However, precautions must be taken to ensure high performance and lack of data-races. In particular, the performance of an unmodified asyncio library may slightly suffer. The first reason for this is that python’s duck-typing means that high-level sematic checking is included in asyncio functions to ensure the argument passed has the correct type. Of course, these may be removed to increase performance at the cost of reliability. Secondly, asyncio in its base form is mostly incompatible with multithreading, as specified by the documentation itself. Even though the library may be locally modified to run callbacks on other threads, the main event loop, as well as reading and writing global variables/mutable objects is not thread-safe. As such, coroutines are still restricted from reading and writing to globals. implementation is actually somewhat misleading. What actually happens is that a scheduled callback records the time it finishes. During a single iteration of the loop, the event loop collects all finished callbacks and executes any callbacks associated with the finished callbacks. This type of implementation is optimized for multiple asynchronous tasks with many order-independent callbacks as 1 loop iteration can perform many callbacks. However, this results in poorer performance for small groups of asynchronous tasks with deep coroutine calls (i.e many instances of coroutines within coroutines), due to one loop iteration required to perform each callback.

The issue with opening a connection is that the server may need to wait for the connection to be established and for results to arrive from the server. This will block event loop execution if executed synchronously. The solution to this is to parse the message in the data\_received callback, but to place the Google APIs connection and client response inside a coroutine. This coroutine is scheduled in the event loop using asyncio.ensure\_future() as a Future, a type of Task. As a result, the data\_received callback can finish completely without blocking the event loop. The following log file shows this

However, there still lies the problem of waiting for the Google Places API connection to be established, as well as waiting for messages to be written to the API and received from the API. The solution to this is the ‘yield from’ keyword. This is implemented as a Python feature (See PEP 380) and has an opcode, to be used as follows:

yield from <coroutine>

The open\_connection method and reading methods are both coroutines. When they are called using yield from, the main coroutine containing the yield from is paused at its location and transfers control to the called coroutine. This point is also used as a callback point for the event loop. In a sense, it yields to the event loop and transfers program execution to the callee. It is the called coroutine’s responsibility to complete its asynchronous execution in the event loop and call the next segment of the main coroutine after creating a result. This allows the connections, reads, and writes to be performed asynchronously in a non blocking manner. In theory, were another client to send a message to the server, the message can be parsed between the connection to the Google Places API, the HTTP request to the API, and the HTTP response to the API, as well as right after it. As the ‘yield from’ keyword is a part of the Python 3.4+ , a high degree of performance is expected.

OOP

Class based

vs

Prototype based

+ flexible

+ simplicity

+ friendly to in time compiling

- performance since need to keep track type of object

Inheritance

Single vs multiple inheritance

Can subclass delete superclass method

What to inherit

Subtyping vs subclass

Parameter Passing

Call by value

+ simple

- low performance for large variable

- callee change doesn’t reflect

Call by result

Opposite of call by value

Callee give the uninitialized parameter variable values

Call by value result

Parameter can have values, but callee can also change them

Call by reference

+ efficient for large variables

+ can communicate

- slow for small variables

- alias – loss potential optimization

Call by macro expansion

Copy paste code (what C does for macro)

+ efficiency

+ flexibility

- recursion not possible

- capture make scope tricky

- debug is hard

Call by name (lazy evaluation +safe, - efficiency)

Pass anonymous function that runs in caller’s frame

Short circuit – if parameters never used, function never called

+ safer in some cases

integral(x, 0, 100, sin(x)){…}

each time evaluate sin(x) in function, x will be updated

Error handling

Error – programmer define

Fault – latent problem in program

Failure – bad execution

Precondition

Total definition

+ performance

- debugging

Fatal error – dump core, not ideal OOP (object should only

have local effect)

Exception

+ beautiful

- need a structure for exception

- debug (g(x) + f(x) //can’t tell which throw exception)

Cost model

Scheme

Append to front – O(1)

Extract front – O(1)

Extract tail – O(1)

Append to end – O(N)

Concatenate – O(N)

Unifying list – O(N)

Prolog

X = Y

If X or Y is just variable – O(1)

If X is complicate structure O(|X|)

If Y is complicate structure O(|Y|)

If both are complicate structure O(min(|X|,|Y|))

Function Call

Well organized on paper

Java

Note

== & != compare address

basic

public class A implements B, C

B b; //B is an interface, this is allowed

b = new A();

Generics

public class Node<T>{

private T value;

Node(T init){

value = init;

}

…

}

Node<Integer> n = new Node<Integer>(0);

Node<String> n = new Node<String>("node");

Exception

public class custom\_exception extends Exception{

public custom\_exception(String details){

super(details)

}

}

try{

throw new custom\_exception(…);

…

}catch(*Exception\_Name* e){

e.getMessage();

}finally{

…

}

Scope

public -- anywhere

protected -- class and subclass

private -- class

static -- associate to the class not object of the class

Inheritance & Polymorphism

interface -- implements , forced implementation of abstract, can't have constructor

superclass – extends

single inheritance – store reference to other classes to work around(forwarding)

Multi-thread

synchronized

*scope\_indicator* synchronized *return\_type function\_name*(){…}

only one thread can access this function at a time

synchronize counter

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {c++;}

public synchronized void decrement() {c--;}

public synchronized int value() {return c;}

}

volatile

never cached, but not guaranteed to be atomic

private static volatile int x = 0;

Thread

public class *runnable\_class\_name* implements Runnable {

public void run() {

…

}

}

…

Thread t = new Thread(new *runnable\_class\_name*);

t.start();

t.join();

binding

Dynamic binding -- The dynamic type of x determines which method is called

Static binding -- The static type of x determines which method is called

final Keyword

final fields

can only be initialized once

must be definitely assigned in every constructor of the class

final static fields

must be definitely assigned in a static initializer of the class(in the class definition)

final classes

can't be extended

final methods

can't be changed by subclass

static binding

finalize

An empty method in object class (thus in every class)

Garbage Collector always calls just before the deletion/destroying the object

class A{

…

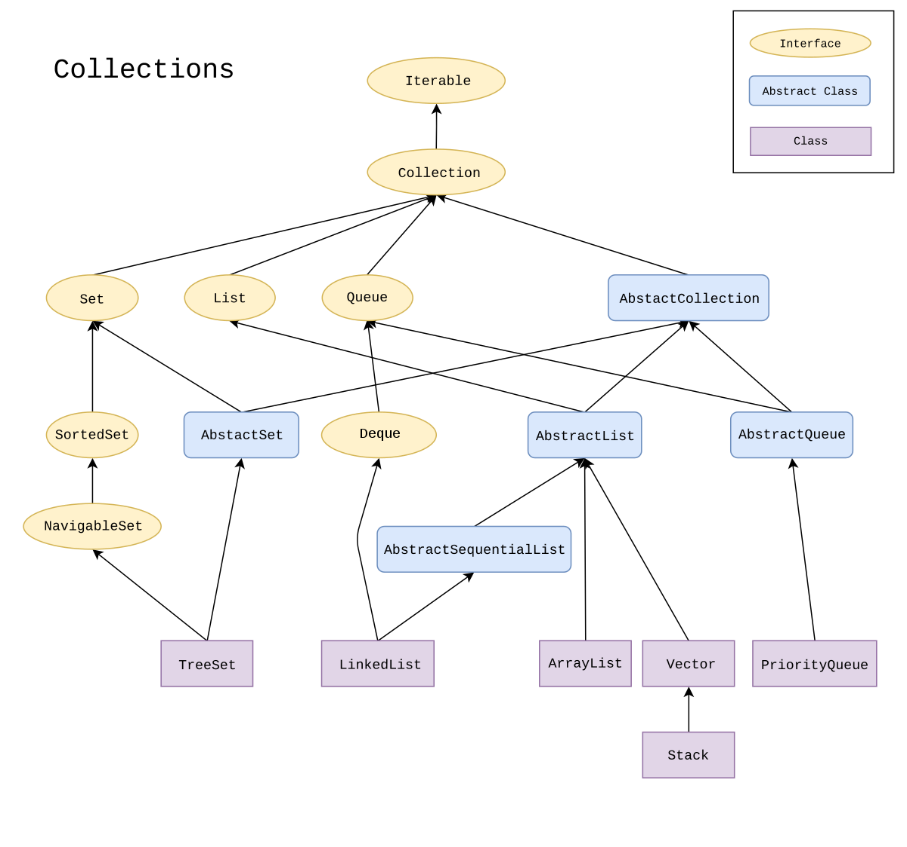
protected void finalize(){**//protected good practice**

…//ran before garbage collection

}

}

Collection



Serialization

mechanism of converting the state of an object into a byte stream

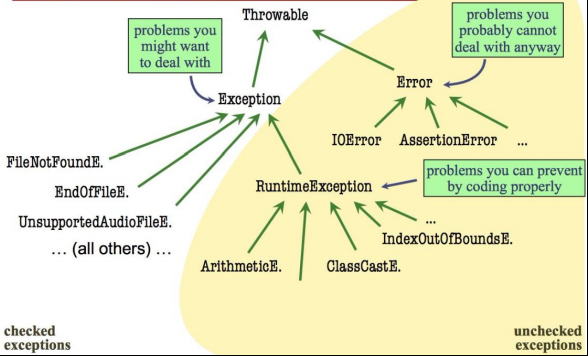
save/persist state of an object

 travel an object across a network

Java Memory Model

Thread Stack 
methodOne() 
Local variable 1 
Local variable 2 
method Two() 
Local variable 1 
Thread Stack 
methodOne() 
Local variable 1 
Local variable 2 
hodTwo() 
Local variable 1 
Object 1 
Object 2 
m 
Object 3 
Object 4 
Object 5 
Heap 
JVM 

1. local variable primitive type -- totally kept on the thread stack.
2. local variable reference to an object -- reference (the local variable) is on the thread stack, object itself if on the heap.
3. object contain methods and methods contain local variables -- local variables on the thread stack
4. object's member variables on the heap
5. Static class variables on the heap along with the class definition



Ocaml

Note

Function call high precedence

f x+y => (f x)+y

Precedence: list > tuple > function return

int\*int list=> int\*(int list)

Variable

let *var* = …;;

let *var*=

let *var1*= [*element1*; *element2*;…]

and *var2*= …

in …;;

List

List.rev *list*];;

List.flatten *list*;;

List.map *function list*;;

List.filter *function list*;;

Type Define

type *name* = *value1*|*value2*|*value3*…;;

type *name* = *name* of *value\_contructor1*|*value\_contructor2*|…;;

type binary\_tree = Leaf | Node of int \* binary\_tree \* binary\_tree;;

ex\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

type point =

Cartesian of (float \* float)

| Polar of (float \* float);;

let x = Cartesian (0.5, 0.5);;

match x with

Cartesian (0.5, 0.5) -> true

| \_ -> false;;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Control Flow

if .. then …

else if … then

else …

Match

match … with

… -> …

| … -> …

|(*element1*, *element2*) -> …

|*head*::*tail* -> …

| \_ -> …

match … with

… when *conditions* -> …

| …

Function

let *var* = fun *para* -> …;;

anonymous function

let *function\_name para1* *para2*= …;;

let *function\_name* = function

|0 -> ..

| \_ -> … ;;

Recursion

let rec *function\_name* x = …;;

let rec fact (x:int) : int =

if x = 0 then 1

else x \* (fact (x-1));;

Conditional

let *FunctionName Para* =

If … then …

else …;;

= -- Traditional equality

== -- Compares physical location (and deprecated)