OCaml

Week 1 discussion

# print\_string "Hello World\n";;

Hello World

- : unit = ()

# let my\_value = 5;;

val my\_value : int = 5

# let numbers = [1;2;3;4;5];;

val numbers : int list = [1; 2; 3; 4; 5]

lists must have same types

# 0::[1;2;3];;

- : int list = [0; 1; 2; 3]

# 0::1::2::3::[4];;

- : int list = [0; 1; 2; 3; 4]

# let average a b = (a+b)/2;;

val average : int -> int -> int = <fun>

# average 5 7;;

- : int = 6

# let average\_f a b = (a +. b)/.2.0;;

val average\_f : float -> float -> float = <fun>

# average\_f 9.0 4.0

;;

- : float = 6.5

# average\_f 9 4;;

Error: This expression has type int but an expression was expected of type

float

Round down

# average 9 4;;

- : int = 6

# let average a b =

let sum = a +. b in

sum /. 2.0 ;;

val average : float -> float -> float = <fun>

# (fun x -> x\*x) 5;;

- : int = 25

# (fun x -> x+5) 10;;

- : int = 15

(\* lists must have same types \*)

head::tail: head is the first element, tail is all the remaining elements.

:: right associative, the previous statement becomes 0::(1::(2::[3]))

# List.map (fun x -> x\*x) [1;2;3;4;5];;

- : int list = [1; 4; 9; 16; 25]

Transform all elements of the list

# List.filter(fun x -> x<3) [1;2;3;4;5];;

- : int list = [1; 2]

Elements that satisfy

# List.for\_all (fun x -> x<3) [1;2;3;4;5];;

- : bool = false

# List.exists (fun x -> x=6) [1;2;3;4;5];;

- : bool = false

# List.map (fun x -> x+1) [1;2;3;4;5];;

# List.filter (fun x -> x mod 2 = 0) [1;2;3;4;5];;

even elements in a list

# List.for\_all (fun x -> x mod 2 = 0) [1;2;3;4;5];;

if all elements are even

# let is\_zero x = if x = 0 then true else false;;

# let is\_zero x = match x with

| 0 -> true

| \_ -> false -> otherwise

switch statement

# let rec factorial a = match a with

| x when x < 2 -> 1

| x -> x \* factorial (x-1);;

# let tuple\_matcher x = match x with

| (1,a) -> a

| \_ -> 0;;

val tuple\_matcher : int \* int -> int = <fun>

# tuple\_matcher(1,5);;

- : int = 5

# tuple\_matcher(0,5);;

- : int = 0

# let print\_name (p:person) = match p with

| (p\_age,p\_name) -> print\_string p\_name;;

# let my\_person = (111, "bob": person);;

val my\_person : person = (111, "bob")

# print\_name my\_person;;

bob- : unit = ()

# type ccle =

| Student of string

| Professor of string

| TA of string;;

(\*Week 2\*)

(\*Pattern matching: \*)

# let first x = match x with

| (left, \_) -> left;;

val first : 'a \* 'b -> 'a = <fun>

# first (1,2);;

- : int = 1

# let first = function

| (left, \_) -> left;;

val first : 'a \* 'b -> 'a = <fun>

# first (1,2);;

- : int = 1

# let first (left,\_) = left;;

val first : 'a \* 'b -> 'a = <fun>

# first (1,2);;

- : int = 1

Factorial:

# let rec f = function

| 0 -> 1

| x -> x\*(f(x-1));;

val f : int -> int = <fun>

# let f = function

| a,b when a > 5 -> b

| \_ -> "No match";;

val f : int \* string -> string = <fun>

# f (5, "hello");;

- : string = "No match"

# f (6, "hello");;

- : string = "hello"

# f (fun x -> x+1);;

- : int = 3

# f (fun x -> x\*2);;

- : int = 4

Type recap:

# type ('nonterminal, 'terminal) symbol =

| N of 'nonterminal

| T of 'terminal;;

type ('nonterminal, 'terminal) symbol = N of 'nonterminal | T of 'terminal

# type awksub\_nonterminals = Expr | Lvalue | Incrop | Binop | Num;;

type awksub\_nonterminals = Expr | Lvalue | Incrop | Binop | Num

# [T "("; N Expr; T")"];;

- : (awksub\_nonterminals, string) symbol list = [T "("; N Expr; T ")"]

No return value: (option)

# let divide a b = match a b with

| x,y when y = 0.0 -> None

| x,y -> Some(x /. y);;

List module:

# List.mapi(fun i x -> (i,x)) ["A";"B"];;

- : (int \* string) list = [(0, "A"); (1, "B")]

get list elements in indices 2-4:

# let a = List.mapi(fun i x -> (i,x)) [10;5;7;2;6;9;4];;

val a : (int \* int) list =

[(0, 10); (1, 5); (2, 7); (3, 2); (4, 6); (5, 9); (6, 4)]

# let b = List.filter (fun x -> (fst x) >= 2 && (fst x) <= 4) a;;

val b : (int \* int) list = [(2, 7); (3, 2); (4, 6)]

# let c = snd (List.split b);;

val c : int list = [7; 2; 6]

# snd (List.split hobbits);;

- : int list = [129; 51; 37; 29]

# List.fold\_left (fun acc x -> acc+x) 0 (snd (List.split hobbits));;

- : int = 246

Homework 2:

"Mark drinks"

Phrase Phrase -> noun verb

Noun Verb Noun -> mary

mary Verb Backtrack

Noun Verb Noun -> mark

mark Verb Verb -> eats

mark eats Backtrack

mark Verb Verb -> drinks

mark drinks

OCaml

Type inference

Pattern matching

Currying

Prolog

7 = 5 + 2

no

X = 5 + 2

X=5+2

yes

A + B = 5 + 2

X is 5 + 2

X = 7

yes

7 is 5 + 2

yes

5 + 2 is 7

no

X is 5 + Y

uncaught exception

5 + 2 =:= 4 + 3

yes

X =:= 4 + 3

uncaught exception

X = 5, Y = 5, X =:= Y

X = 5

Y = 5

yes.

%Arithmetics

x = y X =:= Y.

x != y X =\= Y.

%Backtracking

If one choice fails, it backtracks.

Recursion.

% 1

flight(lax, alt).

flight(atl, jfk).

flight(jfk, lhr).

can\_travel(X, Y):- flight(X, Y).

can\_travel(X, Y):- flight(X, Z), can\_travel(Z, Y).

can\_travel(lax, lhr).

yes.

can\_travel(lhr, lax).

no.

% 2

father(john, paul).

father(john, henry).

mother(paul, mary).

mother(mary, susan).

ancestor(X, Y):- father(X,Y); mother(X,Y). %parents

ancestor(X, Y):- (father(X,Z), father(Z,Y)); (mother(X,Z), mother(Z,Y)). %grandparents

ancestor(X, Y):- father(X,Y); mother(X,Y). %parents

ancestor(X, Y):- (father(X,Z), ancestor(Z,Y)); (mother(X,Z),ancestor(Z,Y)).

List.

[val1, val2, val3, ... , valn].

p([H|T], H, T).

exists(X, []):- fail.

exists(X, [X|\_]).

exists(X, [H|T]):-exists(X, T).

append([], Y, Y).

append([XH|XY], Y, [XH|RT]):-append(XT, Y, RT).

remove(x, [], []).

remove(x, [X|T], TN):- remove(X, T, TN).

remove(x, [H|T], [H|TN]):- remove(X, T, TN).

%Debug

trace

Week 8

===============================

(define print)

(printf "%d abc")

(lambda (a b . args) - ) -> one or more argument

(lambda args) -> zero or more argument

(define list (lambda args args))

(define (list . args) args) -> if last is preceeded a dot, at least n arguments

(list 'a b c)

=> (a b 97) -> assume c=97

'(a b c)

(lambda id (cons 'x d))

(define (foo . id) (cons 'x id))

(let (a (+ 8 4)) (b 3))

(sqrt (+ (\* a a) (\* b b)))

==>

(lambda (a b) (sqrt (\* (\* a a) (\* b b))) (+ 8 4) 3 )

evaluate: (+ 8 4) --> 3 --> (sqrt (\* (\* a a) (\* b b)))

(define (revapp l a)

(if (null? l)

a

(revapp (cdr l) (cons (car l) a) )))

((lambda (b) (\* 3 b)) (+ 7 b))

(and E F) => (if E F #f)

(define-syntax and

(syntax-rules ()

((and) #t)

((and x) x)

((and x y ...) (if x (and y ...) #f)

)))

(define-syntax or

(syntax-rules ()

((or) #f)

((or x) x)

((or x y ...) (if x x (or y ...))

)))

(define-syntax let

(syntax-rules ()

((let ((name val) ... ) body1 body2 ...)

((lambda (name ...) body1 body2 ...) val ...))

((let tag ((name val)...) body1 body2)

((letrec ((tag (lambda (name ...) body1 body2 ...)))

tag)

val ...))))

Tail recursion

(define f (lambda (x) (a x) (b x) (c x)))

(lambda ... (expr)) -> expr is a tail call

(if expr expr expr) -> the last two expr are also tail call

(lambda (x) (a x) (if x (b x) (c x))) -> (b x) (c x)

(lambda (x) (a x) (if (a x) (b x) (c x))) -> stack will grow, need to check return value of (a x)

(and E1 E2 ... En)

(or E1 E2 ... En)

En is a tail call

(let ((a ...)

(b ...))

(expr))

(define (revapp l a)

(if (null? l)

a -> tail call

(revapp (cdr l) (cons (car l) a)))) -> tail call

callq revapp

addq...

ret

jmp revapp, reuse return address

(define revapp

(lambda (l a)

(if (null? l).........))) -> tail call

Continuations

data structures representing the planned future execution of your program

at machine level:

ip %rip -> code

env\_p %rbp or %rsp-k -> frame of current procedure

create a continuation in scheme: create ip+ep

(call-with-current-continuation p)

p is a procedure that takes one argument

(1) create a continuation k

(2) (p k)

(3) return whatever step (2) returns

To use a continuation k

do this (k 42)

(1) sets interpreter's ip, ep from k,

(2) arranges for 42 to be returned from call-with-current-continuations

(define (prod ls)

(call/cc (lambda (break)

(let pr ((ls ls))

(if (null? ls) 1

(if (zero? (car ls)) (break 0)

(X (car ls) (pr (cdr ls)))))))))

(define lwp-list '())

(define (lsp thunk)

(set! lwp-list (append lwp-list (list thunk))))

(define (start)

(let ((first (car lwp-list)))

(set! lwp-list (cdr lwp-list))

(first)))

thunk: parameterless procedure executed for side effects

(define (yield)

(call/cc

(lambda (k)

(lwp (lambda () (k 42))) -> (k 42) set ip to k'ip, ev to k'ep, return value is 42

(start))))

(lwp (lambda () (let f() (display "h") (yield) (f))))

(lwp (lambda () (let f() (display "i") (yield) (f))))

(lwp (lambda () (let f() (newline) (yield) (f))))

(start)

==>

hi

hi

hi

...

i

Continuation passing style

All functions take an extra continuation argument

(define (prod ls k)

(let (break k))

(let pr ((ls ls) (k k))

(if (null? ls) (k 1)

(if (zero? (car ls)) (break 0)

(pr (cdr ls)

(lambda (r) (k (\* r (car ls))))

)

)

)

)

)

Week 9

===============================

Storage (Memory, RAM) management

What gets stored in RAM/VM?

Different types of

1) Allocated by linker (addresses hardwired into executable)

2a) Allocated at function entry (fixed-size activation record, standard in c/c++)

int foo(int x) {

int x = x+1;

if (x\*2 < y) {

int a = ...

char c [1000];

}

else

int b = ...

}

STACK of fixed size:

activation record:

40 local variables

overhead

frame pointer

0 return address

2b) Allocated block entry (varying size frame)

Problem of a fixed size frame:

- Can't deal with array with size not known at compile time

- Two registers: where is the frame pointer, how big is the call frame?

2c) Nested functions

Count fixed size from frame pointer

Dynamic chain

length: number of times a function is called, depends on execution

Static chain:

length: depth of nested functions in the source code, compile time

int f(int a) {

int g(int b) {

int h(int c) {

return c+b+a

}

}

}

STACK:

h's frame:

frame pointer

dynamic chain of caller's frame pointers

return address

static chain points to g's frame

g's frame:

f's frame:

C-style curring:

func\* f (int y) {

int g(int x) {

return x+y;

}

return g;

}

To implement function pointers in the presence of nested function,

you must have two words:

ip + ep

Save ep into the static frame slot of the call function and set the ip

Heap management

Q1. How does the heap manager "know" where the roots are?

A1a. There is just one root.

A1b. 1 table of roots(static)

1 AR table/function, layout of a frame for that function

1 struct table/struct(class), layout of that object

Mark+sweep algorithm

1) Clear all mark bits of all objects in the heap

|2) Find all roots, and mark all objects they point to

mark|3) Find all marked objects, mark all they point to

|4) Repeat step 3 until you find a fixed point

sweep

5) Reclaim all unmarked objects

Q2. How does the heap manager keep track of the free space?

free list pointer

0) find first free block that is big enought

split it to used and free area: shrink the size of free area, return the used address

if exactly same size, remove

1) 0, except: use a pointer + circular freelist

1. Quick lists

struct pair {

void\* car;

void\* cdr;

};

struct pair\* freepairs;

struct pair\* pairalloc(void) {

if (freepairs) {

struct pair\* r = freepairs;

freepairs = r->cdr;

return r; // use the free space if found

}

else {

return malloc(sizeof(struct pair));

}

}

void pairfree (struct pair\* p) {

p->cdr = freepairs; // keep the free space in your own free list

freepairs = p;

}

Incompatible with idea #3: own private free list will be copied by generation-based copying collectors.

2. Python's approach: link count

link count: count number of links pointing at the object

If reference counting field reaches zero,

CPython automatically calls the object-specific deallocation function.

If an object contains references to other objects, then their reference count is decremented too.

Thus other objects may be deallocated in turn.

For example, when a list is deleted the reference count for all its items is decreased.

3. Generation-based copying collectors

Newer objects are more likely to be removed

Week 10

===============================

Cost models

- mental model of how much the program will cost

Development

Runtime

Power/Energy

CPU time

Network latency

Memory, RAM, Secondary storage

Cost of lisp list

(car x) O(1)

(cdr x) O(1)

(length x) O(N)

In scheme

(append a b ... y z)

- the first N-1 elements form a list, the last element can be any value

- creates copies of the first N-1 elements, no copy of z

- O(len(a) + len(b) + ... + len(y))

(eq? a b) same object? pointer comparison

(= a b) numerically equal

(eqv? a b) look at contents of the referenced objects

(equal a b) recursive comparison of data structures

Cost of prolog unification

?- X = Y -> O(|X|+|Y|)

-> O(min(|X|, |Y|))

(let ((a (+ 2 2))

(b (+ 3 1))))

strcmp(a, b)

works faster if args are aligned

AVX2 extension of intel slows down your clocks

Array size

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

a[i][j] = b[i][j+1]; => how to accelerate this?

}

}

int a[100][6]

What is the best array size for performance?

Fit into cache

cache line size = 64 bytes

a[i][j] = xxx + i\*sizeof(a[0])+j\*sizeof(a[0][0])

Cost model of function call

caller:

evaluate arguments,

store result in parameter,

I jump to function start

callee:

tI allocate frame,

tI save registers as needed,

real work

tI copy results to return register

tI deadlock frame

I jmp back

function inlining: (I)

+small functions

-code bloat

-recursion

-breaks.encapsulation

tail call optimization: (T)

Escape analysis

Compiler looking into source codes and noticing new object is used in places that can't be saved

So dont execute new, put the object in the stack, faster allocation, free is free

Common for OOL to create new objects and use and throw it away

OO language

OO language != OO style

Wide variation

Type checking

static type checking:

C++, java, ocaml

reliability

dynamic type checking:

easier to write, more bugs

Python

Class/Prototype

class based

subtyping/specialization

inheritance

constructor

namespace

types: new()constructor, clone() cloner

prototype based

clone()

easy to optimize, simplicity

dynamic type checking

Parameter passing (simple+efficient+clear)

(1) call by value

c, java, scheme, ocaml

f(a+b), caller evaluates to 27, passes a copy to callee

int f(int n) = 27

Problem:

pass in a big array

(2) call by reference

c++

int f(int& n)

-> lower level int f(int\* n), caller calls f(&j)

affect compiler optimization, aliases problem -> disable caching

(3) call by result

(4) call by value-result = call by value + call by result

pass a copy of

read(buf, sizeof bug)

buf is a result parameter

(5) call by unification

(6) call by macro calls

has problem of capture

(7) call by name

- thunks - call by reference: pointers, parameter is procedure

unit -> 'a

f (lambda() ...body...)

>> (f)

- The caller does not evaluate its argument, instead pass in each arg as a thunk

c++: int f(int& n) -> int f(int\* n)

my\_c++: int f(int name n) -> int f(int\* n), same underlying structure

int f(int (\*n)()) { return n()++; }

scheme:

int v = 12;

f( (lambda() v))

void printarg(int arg, int n) {

...

print("Arg is " + arg);

return arg;

}

int main() {

int sum = 0;

for (i = 0; i < n; i++)

sum += a[i];

printarg(sum/n, n); // crash if n==0 in call by value

}

(8) call by need

call by name + cache result of thunks

avoid repeat calling, more efficient

Haskell

- uses exclusively call by need, like OCaml, but ocaml uses call by value

- eager execution

- lazy evaluation

Errors and behaviors

Scheme:

(1) Implementation restrictions (run out of memory)

(2) Unspecified behavior

(eq? (x) '(x)) => #t

(eq? (list 'x) (list 'x)) => #f

(eq? 0 0) => dc

(3) Error is signaled

(open-input-file "foo") => if "foo" does not exist

(4) Undefined behavior

(car 27)

c++: \*(int\*) 0

How to address problems of errors?

A. Compile time checks (static checking)

- most reliable

- not always practical, less flexible

Python is dynamic typed, Ocaml is static.

B. Preconditions e.g. constraints on callers

sqrt(n), n>=0

C. Total definition

sqrt(n), n does not have to be >=0

if n < 0, return NaN

D. Errors are fatal

E. Throw an exception

Easy to do with continuations

Problem: type checking

In java, exceptions are typed, checked statically

Problem: unexpected exceptions

In java, method signatures tell you the exceptions it can throw

Semantics: What do programs mean?

Syntax semantics

Static semantics: determined before execution

Dynamic semantics: determined during run time

- Operational semantics: see what operations the program does

>>> To define a language L, write a program on interpreter for L in a language M that you already know

(1) <En, C> -> v1, <E2 C> -> v2

<E1 + E2, C> -> v1 + v2

In prolog,

eval(E1+E2, C, R):-

eval(E1, c, v1),

eval(E2, c, v2),

R is v1 + v2

(2) <v, C> -> lookup(v, C), v is a variable

In prolog,

eval(v, C, R):-

atom(v),

member(V = R, C).

(3) <let x=E1, in E2, C> ->

<E1,C> -> v1, <E2, bind(x,v1)+C > -> v2

In prolog,

eval(let(x,E1,E2),C, v2) :-

eval(E1, C, v1),

eval(E2, [X=v1|c], v2)

\* Note: let is equivalent to lambda

- Axiomatic semantics: prove a program satisfies a boolean

>>> {P} S {Q}, P and Q are boolean expressions, side effect free, S is a statement

If P is true before S is executed, and if S finishes, then Q is true aftewards

e.g. {n < 0} n = n+1; {n <= 0}

{Q[x/E]}, x = E, {Q}

substitude E everywhere Q has an X.

Q = x<=0

E = x+1

{x+1<=0} precondition, {x=x+1}, {x<=0} postcondition

{P&B}T{Q},{PQ or B}E{Q}

{P} if(B) T else E{Q}

{P} while(B) S{Q}

- Denotational semantics: the program stands for a mathematical function

Attribute grammars (Knuth)

- is a BNF grammar + other stuff that specifies semantics

- for each rule within grammar, has equation that defines attributes for each node in the parse tree

example: E1 -> E2 + T

synthesized attr: type(E1) = if type(E2) == int && type(T) == int, then int, else float

inherited attr: symbolTable(E2) = symTab(E1) = symTab(T

Project

asyncio — Asynchronous I/O

asyncio is a library to write concurrent code using the async/await syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level structured network code.

asyncio provides a set of high-level APIs to:

run Python coroutines concurrently and have full control over their execution;

perform network IO and IPC;

control subprocesses;

distribute tasks via queues;

synchronize concurrent code;

Additionally, there are low-level APIs for library and framework developers to:

create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc;

implement efficient protocols using transports;

bridge callback-based libraries and code with async/await syntax.

Event loop

Week 5

#### Prolog

:-op(700, xfx, [==, \=, =<]). % (non-associative)

:-op(500, yfx, [+, -]). % (left-associative)

:-op(400, yfx, [\*, /]).

:-op(200, xfy, [\*\*]). % (right-associative)

:-op(200, fy, [+, -]).

(\* matcher \*)

let rec mm = function

| Or (a, b) ->

let ma = mm a

and mb = mm b

in fun acc frag ->

match ma acc frag with

| Some x -> Some x

| none -> mb acc frag

| Concat (a, b) ->

let ma = mm a

and mb = mm b

in fun acc ->

ma (mb acc)

### abstract class and final class:

abstract class X {

int i;

public void incr() {i++;} // regular

public void decr(); // interface

}

public class SX extends X {

puglic void decr() {i--;}

}

final class F;

// cannot subclass it

// prevent child class from mistakes

// compiler can inline calls to final methods

// Object class

public class Object {

public Object();

public boolean equals(Object obj); // default ==

public int hashCode(); // default (int)this

// o1.equals(o2);

// o1.hashCode() == o2.hashCode();

public String toString();

// System.out.println(foo) for debugging

public final Class getClass();

// Class c = o.getClass();

protected void finalize() throws Throwable; // objects' last words

protected Object clone() throws CloneNotSupportedException;

}

Week 6

member(X,[X|\_]).

member(X,[\_|L]) :- member(X,L).

?-member(3,[5,3,1])

?-member(Q,[5,3,1])

?-member(3,[A,B,C])

?-member(9,L)

% L=[9|\_19];

% L=[\_29,9|\_39]

revapp([],A,A).

revapp([X|L],A,R):-revapp(L,[X|A],R).

reverse(L,R):-revapp(L,[],R).

?-is(x,'+'(2,3))

% x=5

Week 7 Lec 1

#### Prolog Syntax:

\* atom

\* number

\* variable

\* string

\* structure

#### Program:

\* facts: member(X,[X|\_]).

\* rules: member(X,[\_|L]) :- member(X,L).

\* quries: ?- member(a, R).

#### Unification:

\* Variable binding.

\* Unification is two-way

\* Ocaml-like pattern matching

\* "reverse" pattern matching: binds data variables to values

'='(X,X).

X = X.

p(X,X).

p(f(X,g(Y),Z), f(h(i(K)),L,j(M))).

X1 = f(X2,g(Y),Z), X2 = h(i(K)), L = g(Y), Z = j(M).

Week 7 Lec 2

### Theory

#### propositional logic

proposition: p "It rained today"

q "405 was busy today"

connectives: & and

| or

! not

->, :- implication

(p<=q)

p q p->q q<-p p<->q

0 0 1 1 1

0 1 1 0 0

1 0 0 1 0

1 1 1 1 1

#### Tautology

((p->q)^(q->p)) -> (p<->q)

((p->q)^(q->r)) -> (p->r)

#### First order logic

Predicate calculus

1. logic variables as arguments to predicates

2. quantifiers (for all) (there exists)

for all X man(X)->mortal(X)

formula -> theorem prover -> yes (tautology)

-> no (counterexample)

-> idk (infinite loop)

#### clausal form

B1 | B2 | ... | Bn < - A1 & A2 & ... & Am

consequents < - antecedents

dog(X)|cat(X)|pig(X) < - petlicensed(X) & inLA(X)

##### Horn clauses: n <= 1

Prolog subset of logic:

\* n=1, m=0. fact

\* n=1, m>0. rule

\* n=0. query