Ocaml

1. Operator

^ string concatenate

:: insert an element in list

List use ; to separate elems

Quicksort

```
let neg = fun -> fun x -> neg f (x);;
let rec filter f xs = match xs with
| [] -> []
h::t->if f h then (h::filter f t ) else filter f t;;
let rec sort l = match l with
| [] -> []
|h::t->let (l,r)=partition (>) h t in (sort
l)@[h]@(sort h);;
```

2. Datatype

Expression

```
type Expr =
| Const of int
| Var of string
| op of string *expr *expr
```

Linked List

```
type int_list =
Nil
|Cons of int*int_list
```

Binary Tree

```
type tree =
| Empty
|Node of a' * a' tree list;;
```

3. Useful Functions

List.fold_left

```
Let fold f curr l =
Match l with
| [] -> curr
|h::t->fold f (f curr h) t;;
```

List.map

```
let rec map f l =
 match l with
| [] -> curr
| h::t->(f h)::map f t
```

4. Examples

```
let rec tree_zip t1 t2 =
  match (t1,t2) with
  | (Empty,Empty) -> Empty
  | (Node(r1,l1),Node(r2,l2)) ->
  Node((r1,r2),
 (List.map (fun (t1,t2) -> tree_zip t1 t2)
 (zip l1 l2)))
  | _ -> raise Mismatch
```

Prolog

```
Bagof(X,p(X),R) (**Put all X satisfy p(X) into R**)
Setof(X,p(X),R)  (**Same Bagof, no Duplicate**)
Cutoff  !  (**One Way Door **)
```

Examples

1. Split into odd and even

```
split([],  [],  []).
split([X], [X], []).
split([X|T],[X|R],L) :- split(T,L,R).
```

2. Path no cycle using path_helper

```
path(A, B) :- path_helper(A, B,[]).
path_helper(A, B, Seen) :-
link(A,B),not(member(B, Seen)).
path_helper(A, B, Seen) :-
link(A,C),not(member(C,Seen)),
path_helper(C,B,[C|Seen]).
```

3. Quicksort

```
part([],_,[],[]).
part([H|T],P,[H|R1],R2) :- H=<P,part(T,P,R1,R2).
part([H|T],P,R1,[H|R2]) :- H>P,part(T,P,R1,R2).
qsort([H|T],R) :-
part(L,H,L1,L2),qsort(L1,R1),qsort(L2,R2),
append(R1, [H|R2], R).
```

4. Prefix and Segment

```
prefix([],_).
prefix([H|T1],[H|T2]) :- prefix(T1,T2).
segment(A,B) :- prefix(A,B).
segment(A,[_|T]) :- prefix(A,T).
```

Python

1. Decorator

Decorator (Parameter)

```
def profiled(I):
   def deco(f):
         def g(*args):
            count += I
           return f(*args)
      count = 0
      return g
   return deco
```

```
def profile(I):
  class deco(object):
    def _init_(self,f):
      self._f = f
    def _call_(self,*args):
      return self._f(*args)
  return deco
```

Decorator (No Parameter)

```
def decorator(f):
 def g(*args):
  return f(*args)
 return g
```

```
class decorator(object):
    def _init_(self,f):
        self._f = f
    def _call_(self,*args):
        return self._f(*args)
```

2. Essence

```
@E           def g
def g  => g = E(g)
```

```
@A        Get in deco_a
@B        Get in deco_b
fun       Get in inner_b
          Get in inner_a
A(B(fun)) Get in f
```

3. Examples

(1) Reverse l with reduce

```
def rev(l):
 def fold_fn(acc,elm): return [elem]+acc
 return reduce(fold_fn, l, [])
```

(2) transpose matrix

```
def transpose(m):
 height = len(m)
 width = len(m[0])
 return [ [  m[i][j] for i in range(height)] for j in
range(width)]
```

<<Formal Certification of a Compiler Back-end>>

This paper reports on the development and formal certification of a compiler from Cminor to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Reports on the completion of one half of this program: the certification, using the Coq proof assistant [2], of a lightly-optimizing back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. A front-end translating a subset of C to Cminor is being developed and certified, and will be described in a forthcoming paper.

- Certified, certifying, verified compilation

The formal verification of a compiler consists of establishing a given correctness property Prop(S,C) between a source program S and its compiled code C. Examples of correctness properties include: “if S has well-defined semantics (does not go wrong), then S and C are observationally equivalent”;

<<Checking System Rules Using System-specific, Programmer-Written Compiler Extensions>>

1) Meta-level compilation (MC) to write simple, system-speific compiler extensions that automatically check their code for rule violations.
2) This paper demonstrates the effectiveness of the MC **Real systems:** Linux, OpenBSD, the Xokexokernel, the FLASH machine embeded software.

- Introduction

(1)The most common method used to detect rule violations is **testing;** another common method to detect rule violations is **manual inspections(2)**In our MC system, implementors write extensions in a high-level state-machine language, metal.(3)Resuls:**a.** MC checkers find serious erros in system; **b.** MC extensions can also be used into optimizers; **c.** MC extensions are simple.

- Meta-level Compilation

(1)MC compilaer extensions can check them by searching for the corresponding operations and verifying that they obey the given ordering and/or contectual resrictions. **For example, “for speed, if a shared variable is not modified, protect it with read locks”** can search for each write-lock critical section, examine all variable uses and if no stores occur to protected variables, demote the locks or suggest alternative usage.(2) “assert expectation to be true” ETC., assert “x++=5”

- A Simple Meta-language

(1)Assertions should not have non-debugging side-effects. If has, the program will behave incorrectly (2) Assertion conditions should not fail. (3)**Checking assetions statically**: With MC, evaluating these conditions statically, quickly and precisely finding errors. We wrote such an extension on top xg++. At a high level, it uses xg++’s **dataflow routines** to track the values of scalar variables. At each assert use, it evaluates the assertions expression against these known sets of values.

- Temporal Ordering

(1)This allows a metal extension to find violations by searcing for operations and transitioning to states that allow, disallow, or require other operations.(2)**Enforcing “X before Y”** and the second checks **that code obeys a set of ordering rules for memory allocation and deallocations.(3)** Checking memory management: a. “allocation can fail, kernel code must check whether pointer is valid”, b.“memory can be used after freed”, c.paths ,d.“size of allocated memory can’t be less then size of the object”

- Enforcing Rules Globally

Many rules are **context dependent** and apply gloabally across functions: kernel code can’t call blocking functions; A dynamically loaded kernel module can’t call blocking functions until the module’s reference count has been properly set.

- Linux Mutual Exclusion

False Positive: (1) Code **intentionally violates** the convention for efficiency (2) our checker **only performs local analysis** (3)doesn’t prune simple imposible paths

<<A Few Billion Lines of Code Later>>

The ideal: check millions of lines of code with little manual setup and find the maximum number of serious true errors with the minimum number of false reports.

- Laws of Bug finding

Assuming a reasonable tool, the first order bound on bug counts is just how much code can be shoved through the tool. Law: You can’t check code you don’t see. **Initial,** read-only replay of the build commands: **run make, record its output in a file,** and rewrite the invocations to their compiler **Error: rm -rf ***

Build interposition on Windows requires running the compiler in the debugger. Unfortunately, doing so causes a very popular windows C++ compiler to prematurely exit with a bizarre error message. **Law: You can’t check code you can’t parse.** Many build engineers have a single concrete metric of success: **that all tools terminate with successful exit codes.** Realistically, **diagnosing a compiler’s divergences requires having a copy of the compiler.**