fun fold f acc xs =

case xs of

[] => acc

| x::xs’ => fold f (f(acc,x)) xs’

fun sum\_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0

(\* tournamnet를 인자로 받아서 이긴 플레이어를 반환

\* 도우미 함수들을 let in end로

fun whosWinner(t) =

let (\* 함수 next는현재 rsp 를 반환하고 다음 상태로 ref를 업데이트하는 함수 \*)

fun next(strategyRef) =

let val Cons(rsp, func) = !strategyRef

in

strategyRef := func();

(\* 이 rsp는 func()를 통해 업데이트되기전의 값이므로 현재 rsp 정보임에 유의 \*)

rsp

end

[(fun? e)

(let\* ([f\_name (fun-nameopt e)]

[arg (fun-formal e)]

;[new\_env (cons (cons f\_name (closure env e)) env)]

;[body (eval-under-env (fun-body e) env)]

)

(cond [(and (string? f\_name) (string? arg))

;(closure new\_env e)]

(closure env e)]

[(and (false? f\_name) (string? arg))

(closure env e)]

[#t (error "arguments are not valid : string or #f")])

)

]

[(call? e) ; env for (f\_name . fun\_closure) pair extended when call function,,, not when evaluating fun

(let\* ([fun\_closure (eval-under-env (call-funexp e) env)]

[fun\_arg (eval-under-env (call-actual e) env)]

[function (closure-fun fun\_closure)]

[fun\_env (closure-env fun\_closure)]

)

(if (closure? fun\_closure)

(let\* ([f\_name (fun-nameopt function)]

[f\_formal (fun-formal function)]

[f\_body (fun-body function)]

[arg\_env (cons (cons f\_formal fun\_arg) fun\_env)]

[new\_env (if (false? f\_name)

arg\_env

(cons (cons f\_name fun\_closure) arg\_env))]

)

(eval-under-env f\_body new\_env)

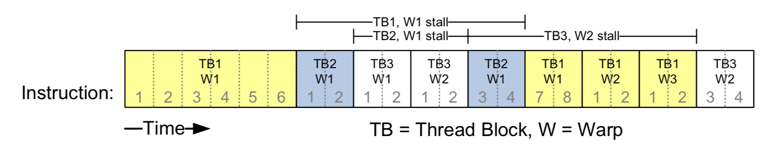
)

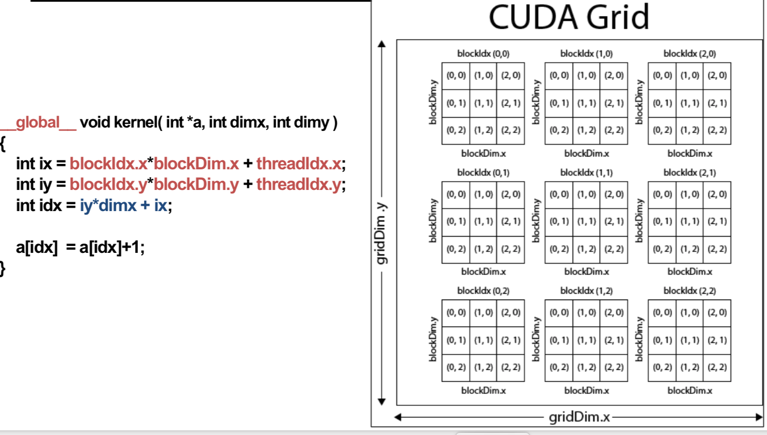
(error "first argument is not valid : closure")

)

)

]





• Global memory is separate hardware from GPU SMs

• The vast majority of memory on a GPU is global memory

• If data doesn’t fit in global memory, process it in chunks that fit

• GPUs have 1~32GB of global memory, with most having ~10GB

• Global memory latency is ~300ns on Kepler and ~600ns on Fermi // Memory coalescing

• Memory accesses done in large groups of Memory Transactions • Done per warp

• Fully utilizes the way IO is setup at the hardware level

• Coalesced memory accesses minimize the number of cache lines read

• GPU cache lines are 128 bytes and are aligned

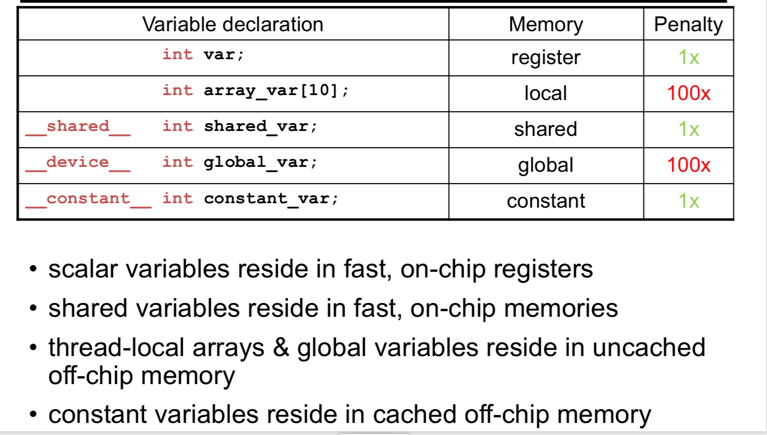
To fix the stride 32 case, we’ll waste a byte on padding and make the stride 33 :)

Don’t store any data in slots 32, 65, 98, .... Now we have

thread 0 ⇒ index 0 (bank 0)

thread 1 ⇒ index 33 (bank 1)

thread i ⇒ index 33 \* i (bank i)



//L1 cache

• Caches local and/or global memory

• Same hardware as shared memory

• For some GPU architectures, configurable (16, 32, 48KB) • Each SM has its own L1 cache

//L2 cache

• Caches all global & local memory accesses • ~1MB in size

• Shared by all SM’s

\_\_global\_\_ void adj\_diff(int \*result, int \*input){

int tx = threadIdx.x;

// allocate a \_\_shared\_\_ array, one element per thread

\_\_shared\_\_ int s\_data[BLOCK\_SIZE];

// each thread reads one element to s\_data

unsigned int i = blockDim.x \* blockIdx.x + tx;

s\_data[tx] = input[i];

// avoid race condition: ensure all loads

// complete before continuing

\_\_syncthreads();

...

}

\_\_global\_\_ void adj\_diff(int \*result, int \*input){

...

if(tx > 0)

result[i] = s\_data[tx] – s\_data[tx–1];

else if(i > 0)

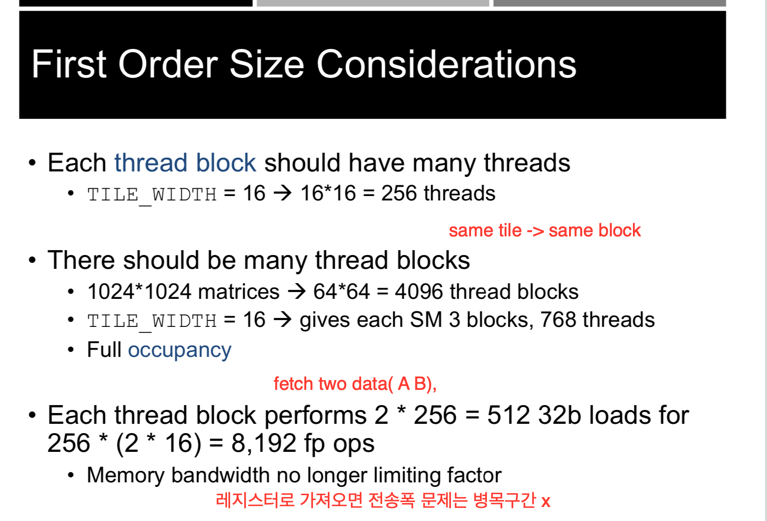
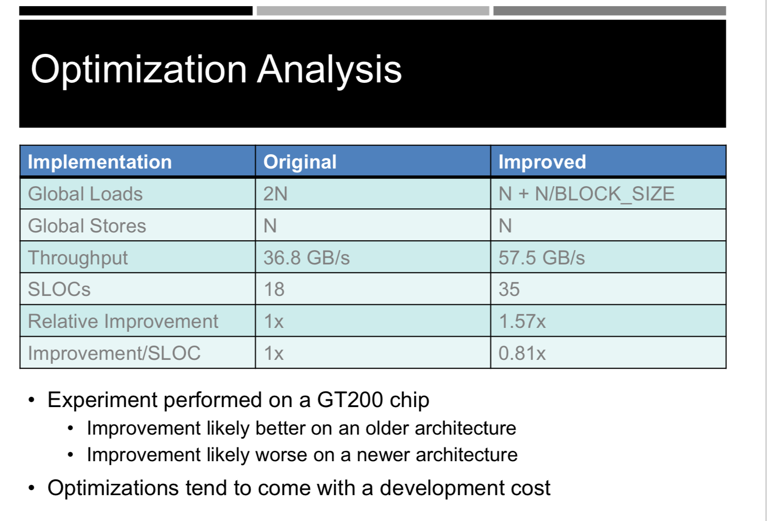
{

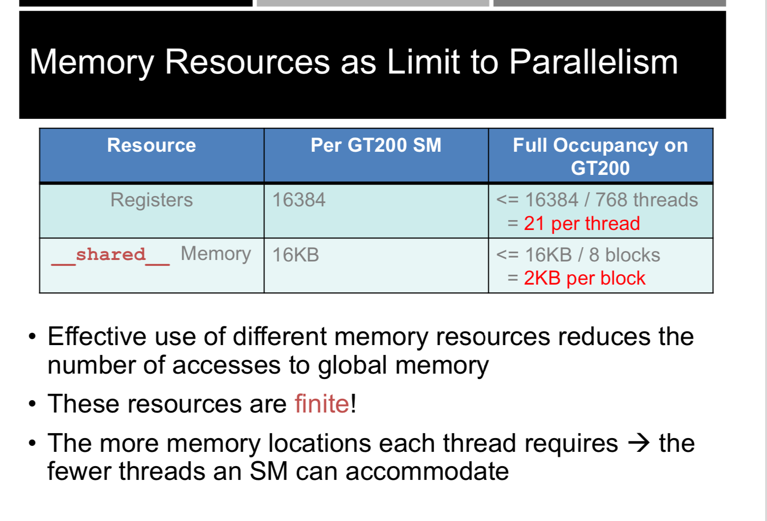
// handle thread block boundary

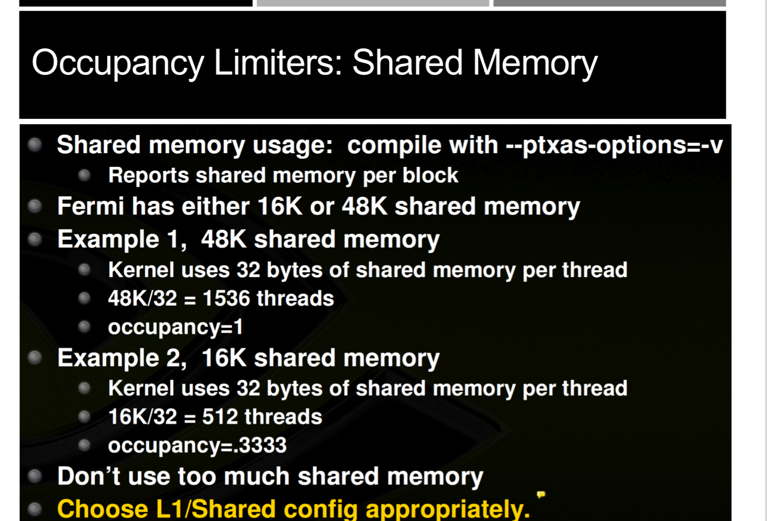
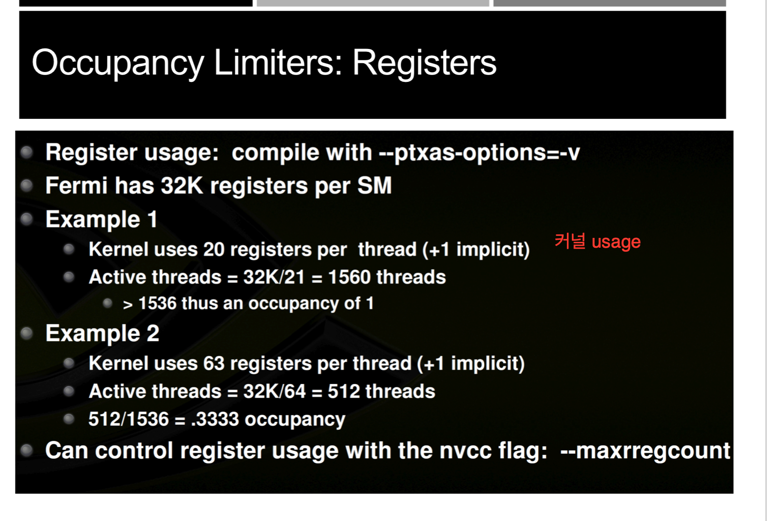
result[i] = s\_data[tx] – input[i-1];

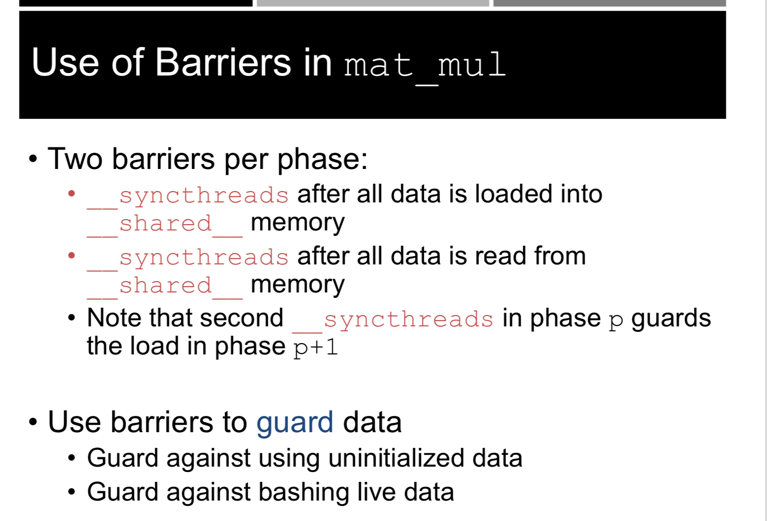
}

}









// global sum via per-block reductions

float sum(float \*d\_input, size\_t n)

{

size\_t block\_size = ..., num\_blocks = ...;

// allocate per-block partial sums

// plus a final total sum

float \*d\_sums = 0;

cudaMalloc((void\*\*)&d\_sums,

sizeof(float) \* (num\_blocks + 1));

... // reduce per-block partial sums

int smem\_sz = block\_size\*sizeof(float); block\_sum<<<num\_blocks,block\_size,smem\_sz>>>(d\_input, d\_sums, n);

// reduce partial sums to a total sum

block\_sum<<<1,block\_size,smem\_sz>>> (d\_sums, d\_sums + num\_blocks, num\_blocks);

// copy result to host

float result = 0;

cudaMemcpy(&result, d\_sums+num\_blocks, ...);

return result;

}

\_\_global\_\_ void inclusive\_scan(int \*data)

{

extern \_\_shared\_\_ int sdata[];

unsigned int i = ...

// load input into \_\_shared\_\_ memory

int sum = input[i];

sdata[threadIdx.x] = sum;

\_\_syncthreads();

...

for(int o = 1; o < blockDim.x; o <<= 1){

if(threadIdx.x >= o)

sum += sdata[threadIdx.x - o];

// wait on reads

\_\_syncthreads();

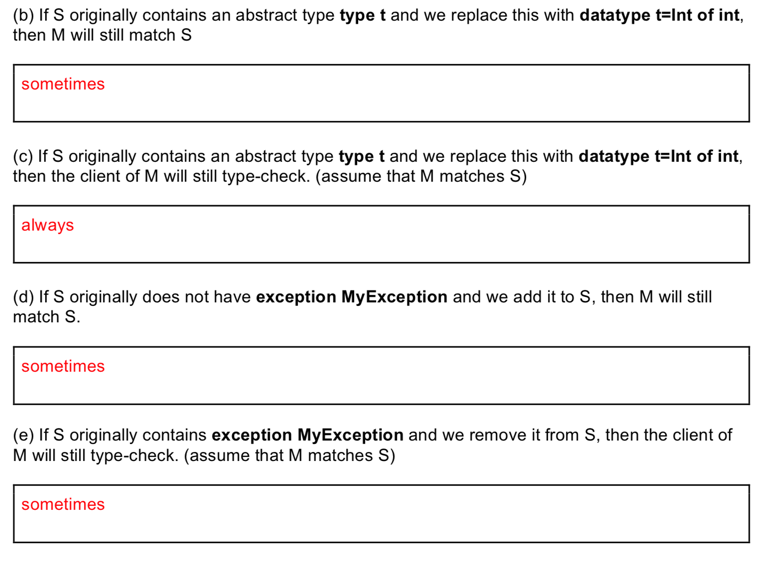
// write my partial sum

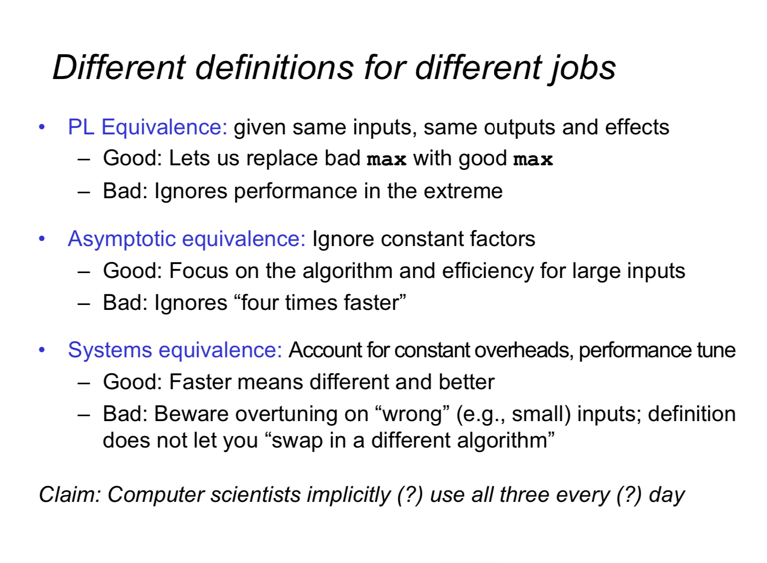
sdata[threadIdx.x] = sum;

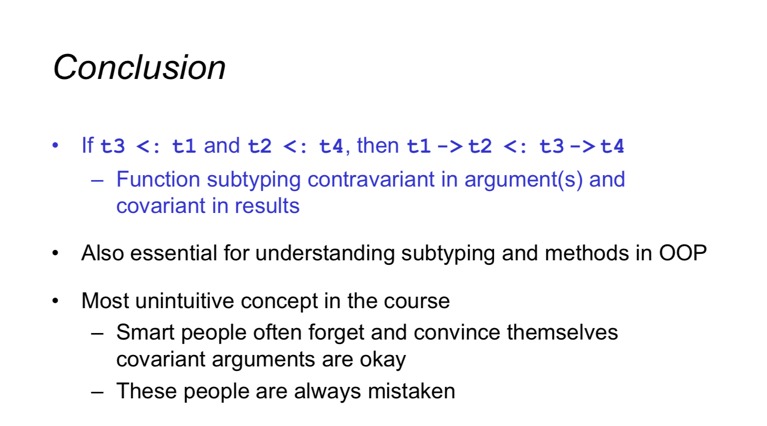
// wait on writes

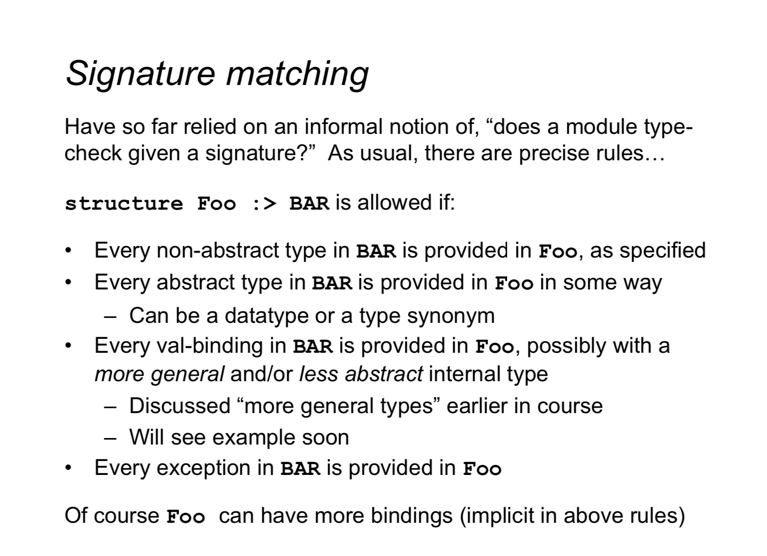
\_\_syncthreads();

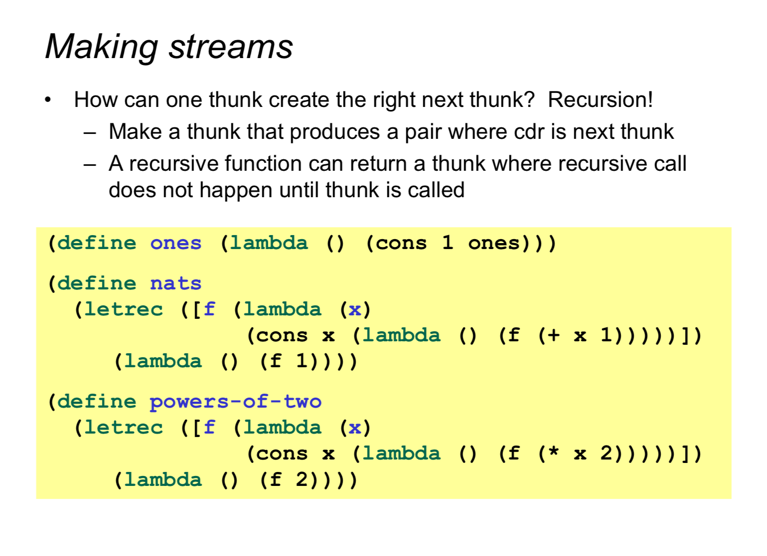
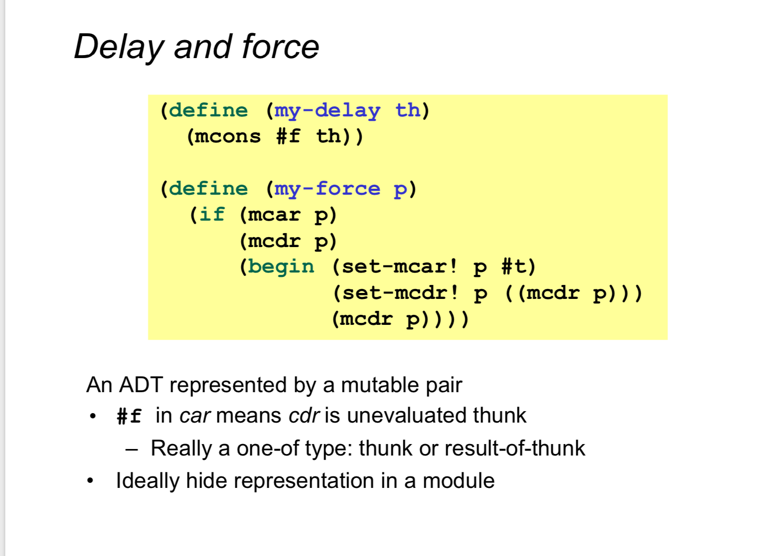
}

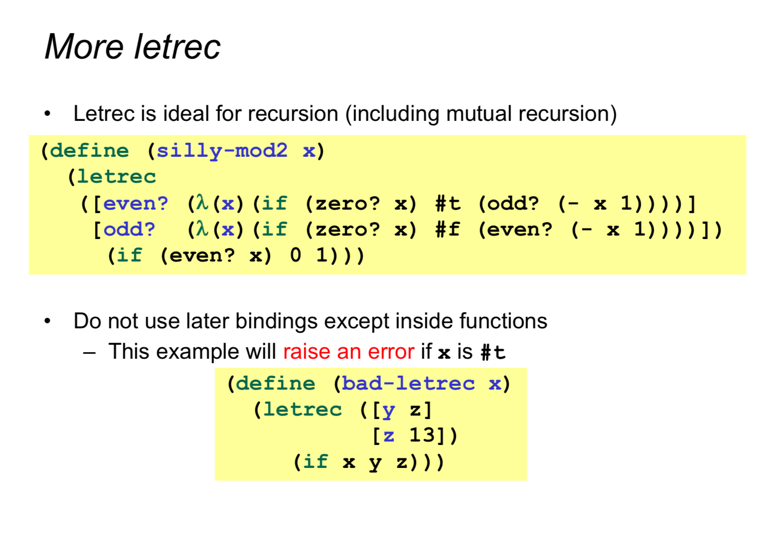










(define stream-factorial

(let ([factorial-stream

(let loop ([i 1]

[a 1])

(stream-cons

a

(loop (add1 i) (\* a i))))])

(λ (n)

(stream-ref factorial-stream n))))

