RefLang

October 27, 2018

Lambda Calculus Review

① λ calculus is a smallest, Universal Programming Language

② grammar: define the lambda calculus

name x
function λx. x
function application e0 e1
} e

③ β-reduction: 3steps
Identify
① syntactic structure: formal parameters
actual parameters
function body

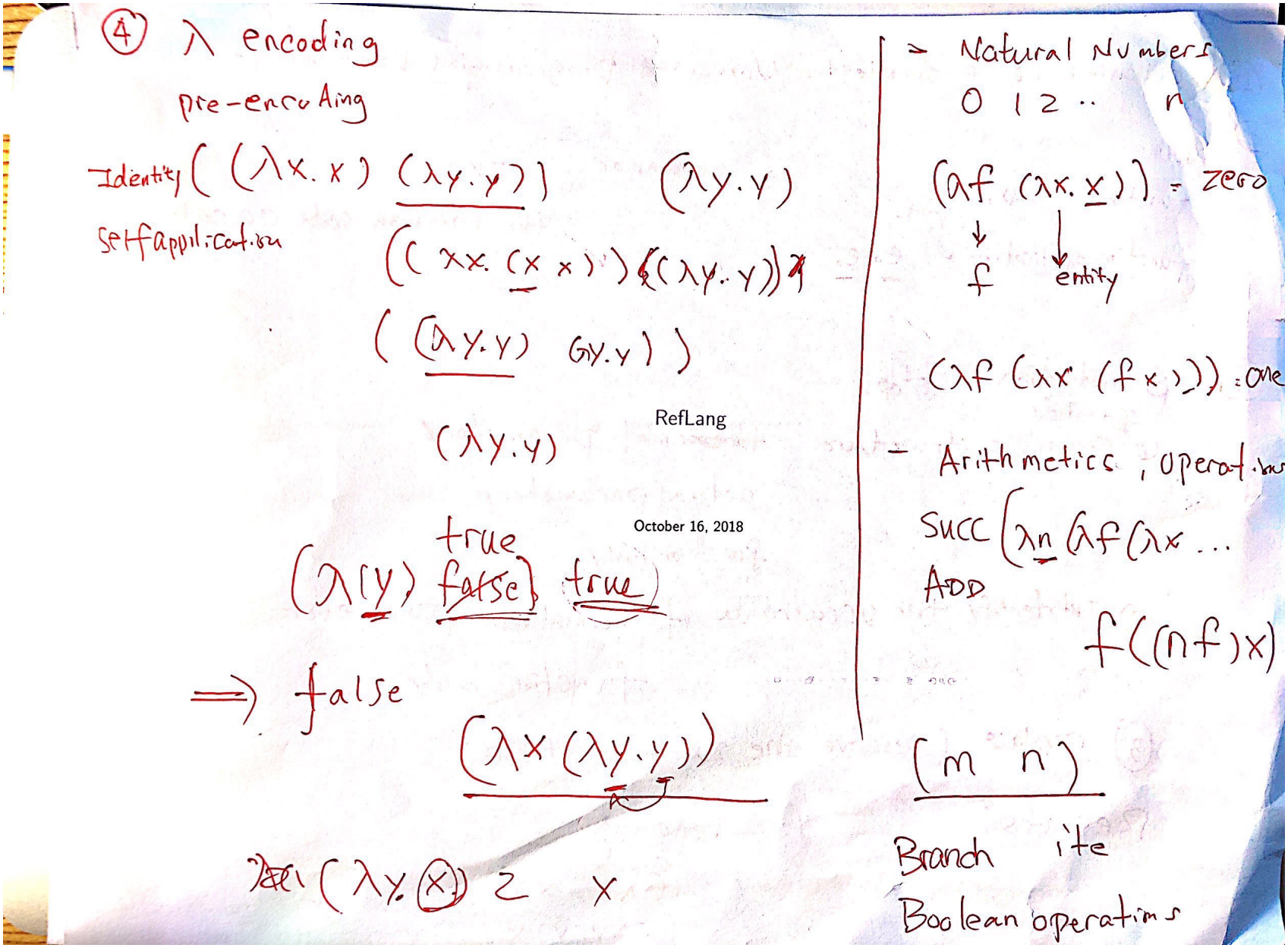
② Identify the occurrance of formal parameter in the function body

③ replace (remove the λx + parathesis)

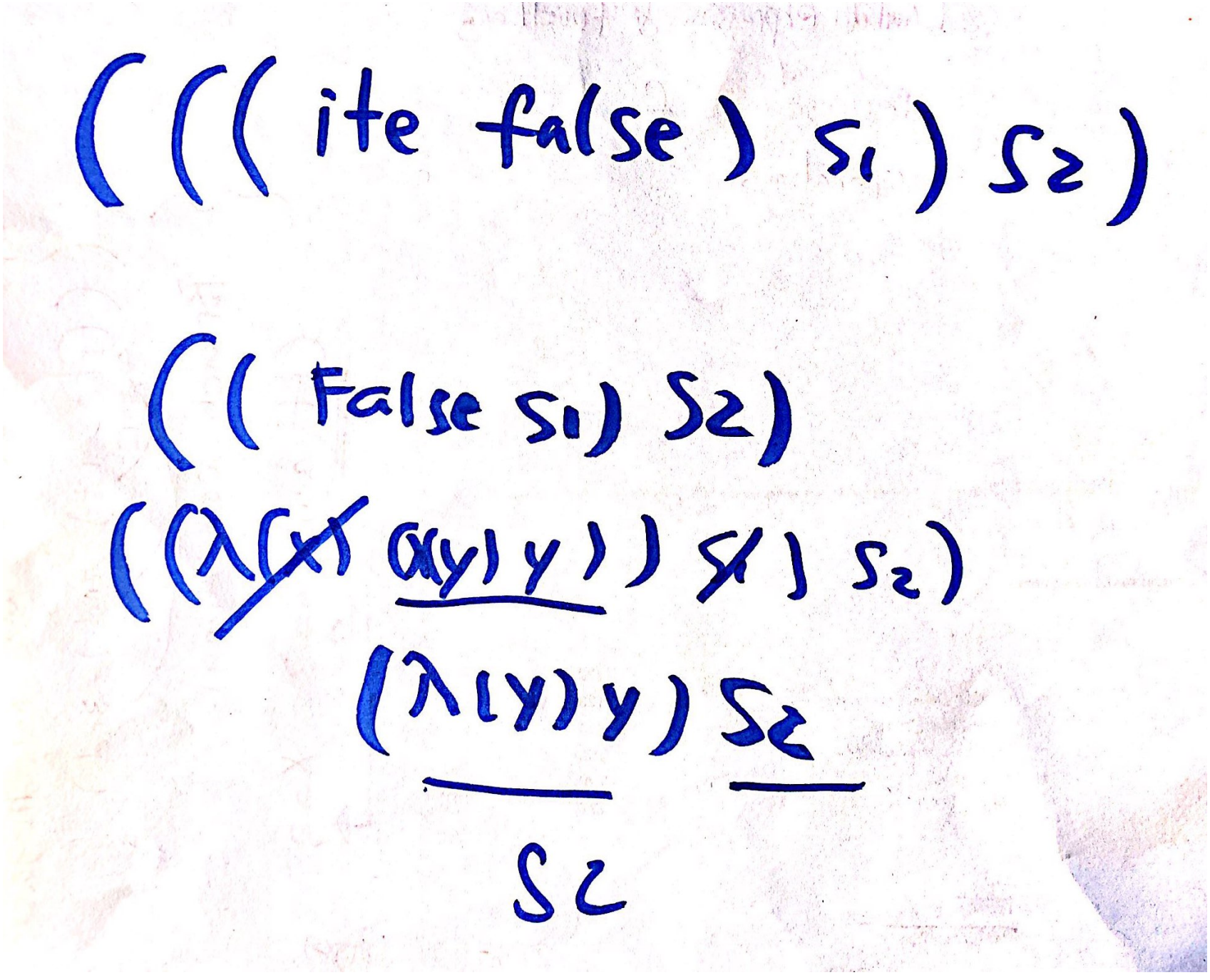
Results? (α-renaming)

((λx. x) x) x
((λy. y) x) x

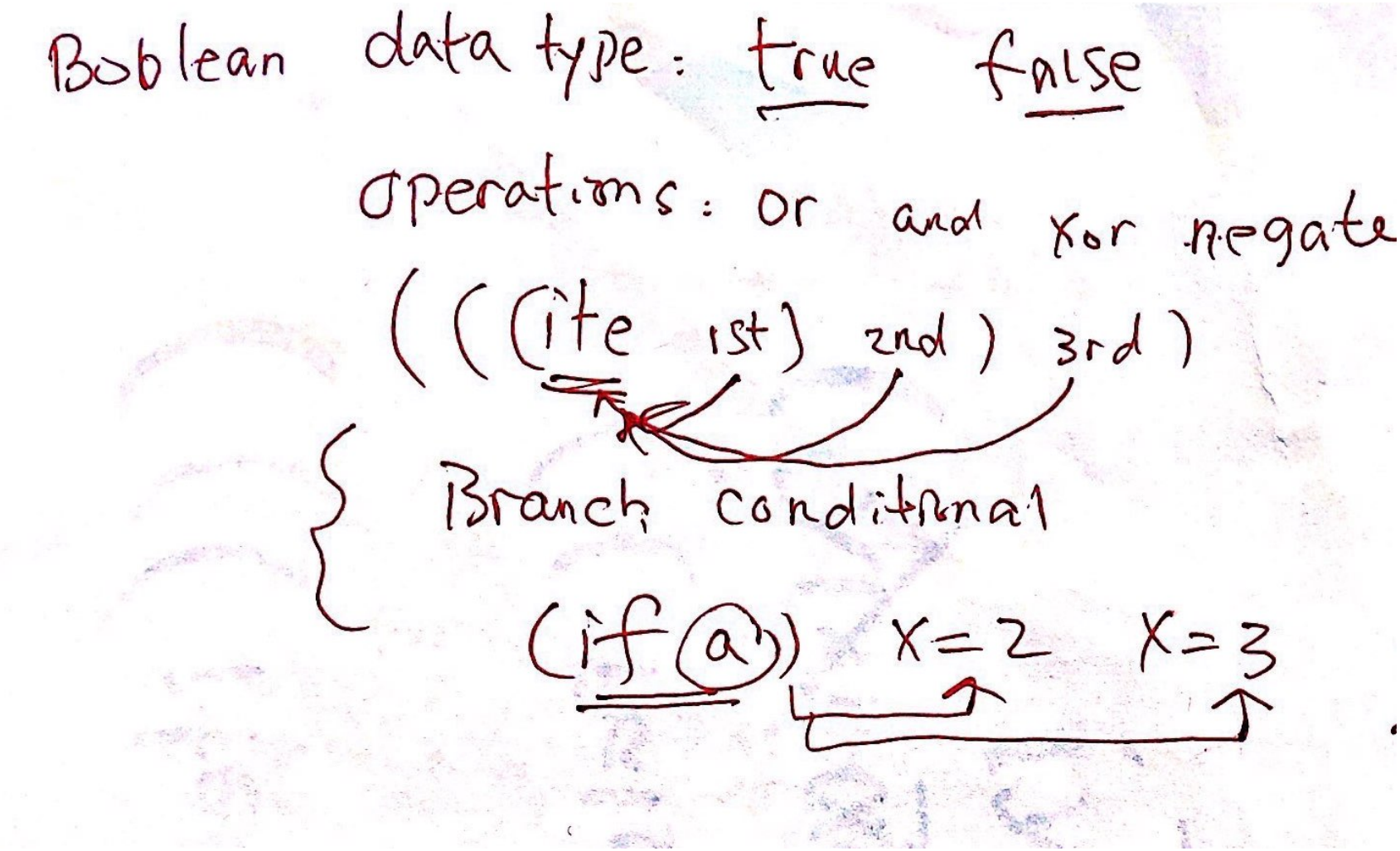
Lambda Calculus Review



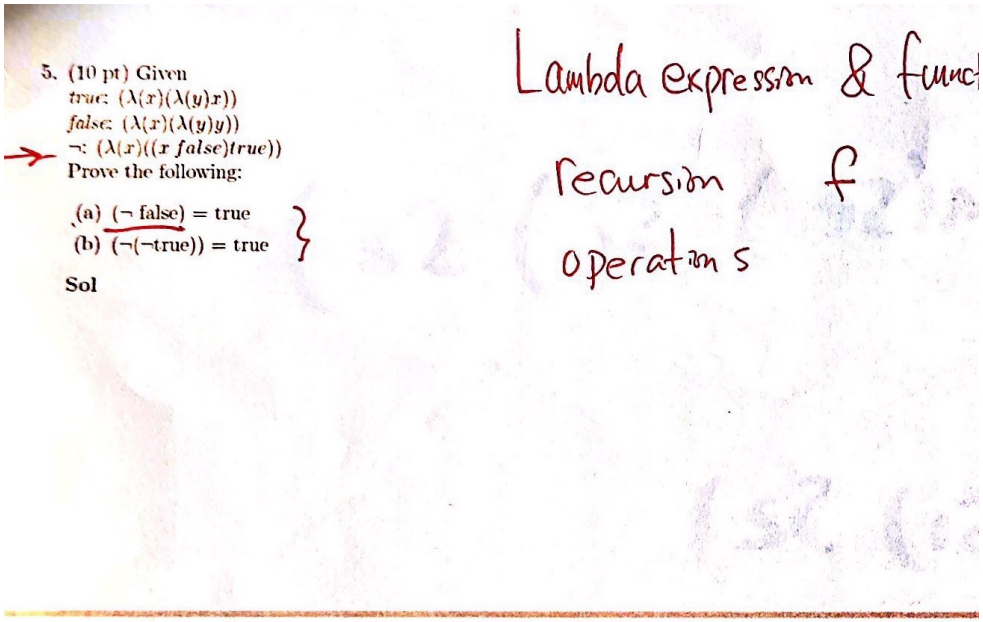
Lambda Calculus Review



Lambda Calculus Review



Lambda Calculus Exercise

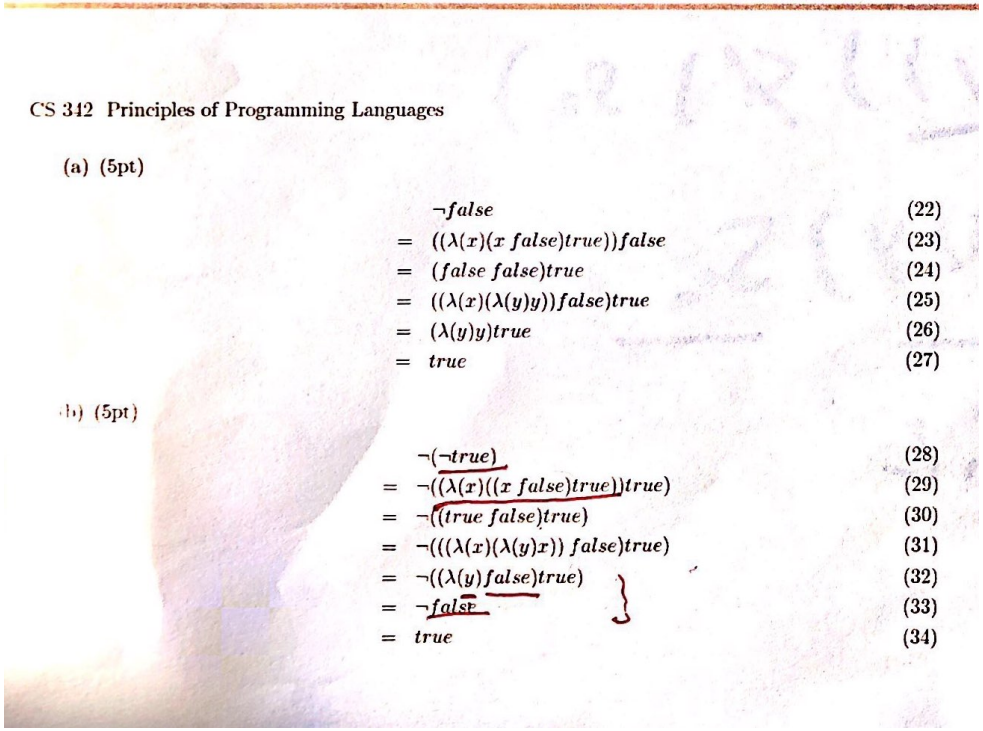


5. (10 pt) Given
true: $(\lambda(x)(\lambda(y)x))$
false: $(\lambda(x)(\lambda(y)y))$
¬: $(\lambda(x)((x\ false)true))$
Prove the following:

(a) $(\neg\ false) = true$
(b) $(\neg(\neg true)) = true$

Sol

Lambda expression & func
recursion f
operations



CS 342 Principles of Programming Languages

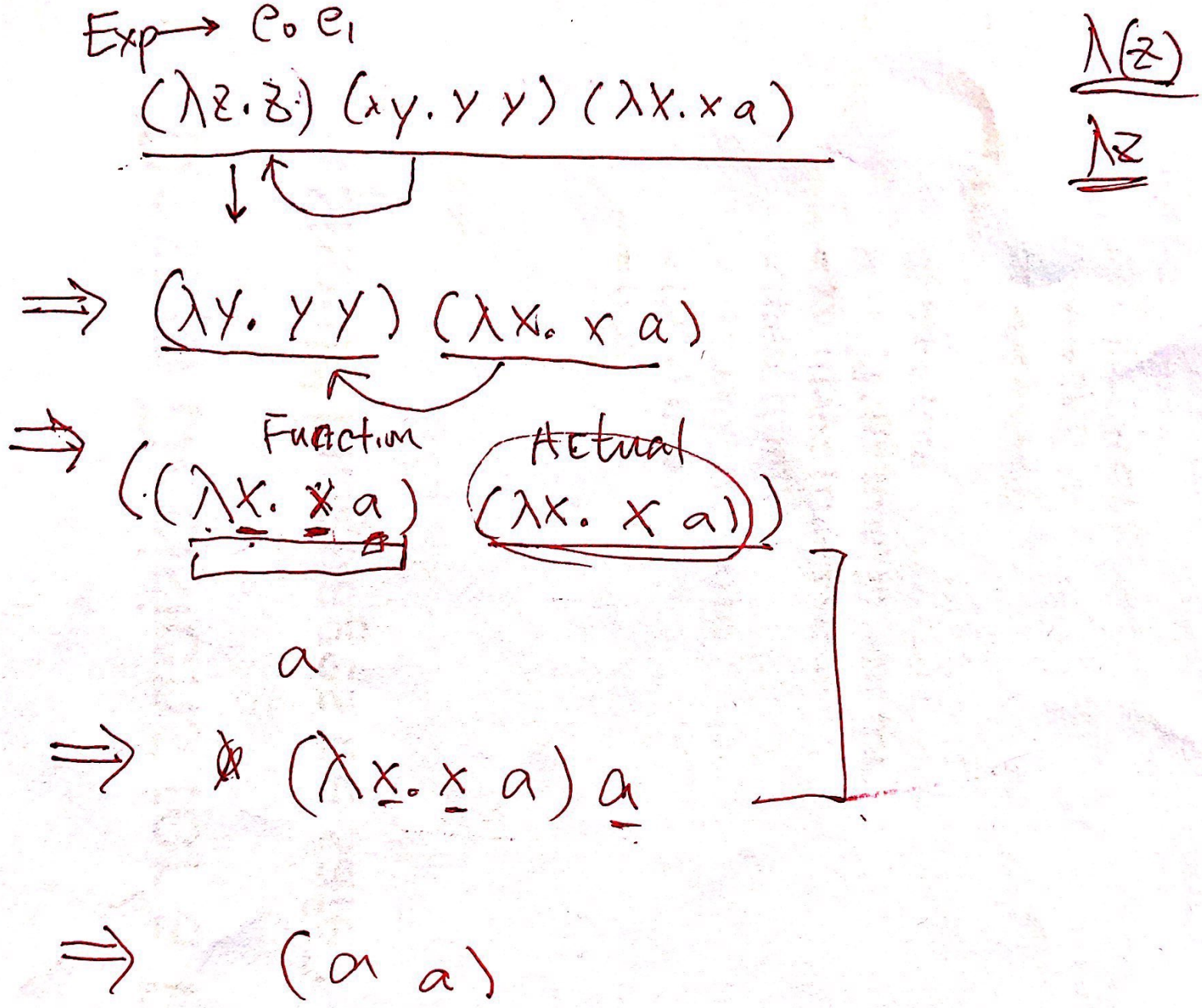
(a) (5pt)

$$\begin{aligned}
& \neg false && (22)\\
= \ & ((\lambda(x)(x\ false)true))false && (23)\\
= \ & (false\ false)true && (24)\\
= \ & ((\lambda(x)(\lambda(y)y))false)true && (25)\\
= \ & (\lambda(y)y)true && (26)\\
= \ & true && (27)
\end{aligned}$$

(b) (5pt)

$$\begin{aligned}
& \neg(\neg true) && (28)\\
= \ & \neg((\lambda(x)((x\ false)true))true) && (29)\\
= \ & \neg((true\ false)true) && (30)\\
= \ & \neg(((\lambda(x)(\lambda(y)x))\ false)true) && (31)\\
= \ & \neg((\lambda(y)false)true) && (32)\\
= \ & \neg false && (33)\\
= \ & true && (34)
\end{aligned}$$

Lambda Calculus Exercise



Lambda Calculus Exercise

b. or false true = true

Given:

or = λx. λy. ((x true) y)

true = λx.λy.x

false = λx.λy.y

Proof:

or false true	
= λx. λy. ((x true) y) false true	// replacing or w/ encoding
= λy. ((false true) y) true	// β-reduction: x → false
= (false true) true	// β-reduction: y → true
= ((λx.λy.y) true) true	// replace 1st false w/ encoding
= (λy.y) true	// β-reduction: x → false
= true	// β-reduction: y → true
	// or false true = true

Lambda Calculus Exercise

or false true = true

or : $\lambda x.\lambda y.((x\ \text{true})\ y)$

false : $\lambda x.(\lambda y.y)$

true : $\lambda x.(\lambda y.x)$

$\lambda x.\lambda y.((x\ \text{true})\ y)\ \text{false}\ \text{true}$

$\lambda y.((\text{false}\ \text{true})\ y)\ \text{true}$

$(\text{false}\ \text{true})\ \text{true}$

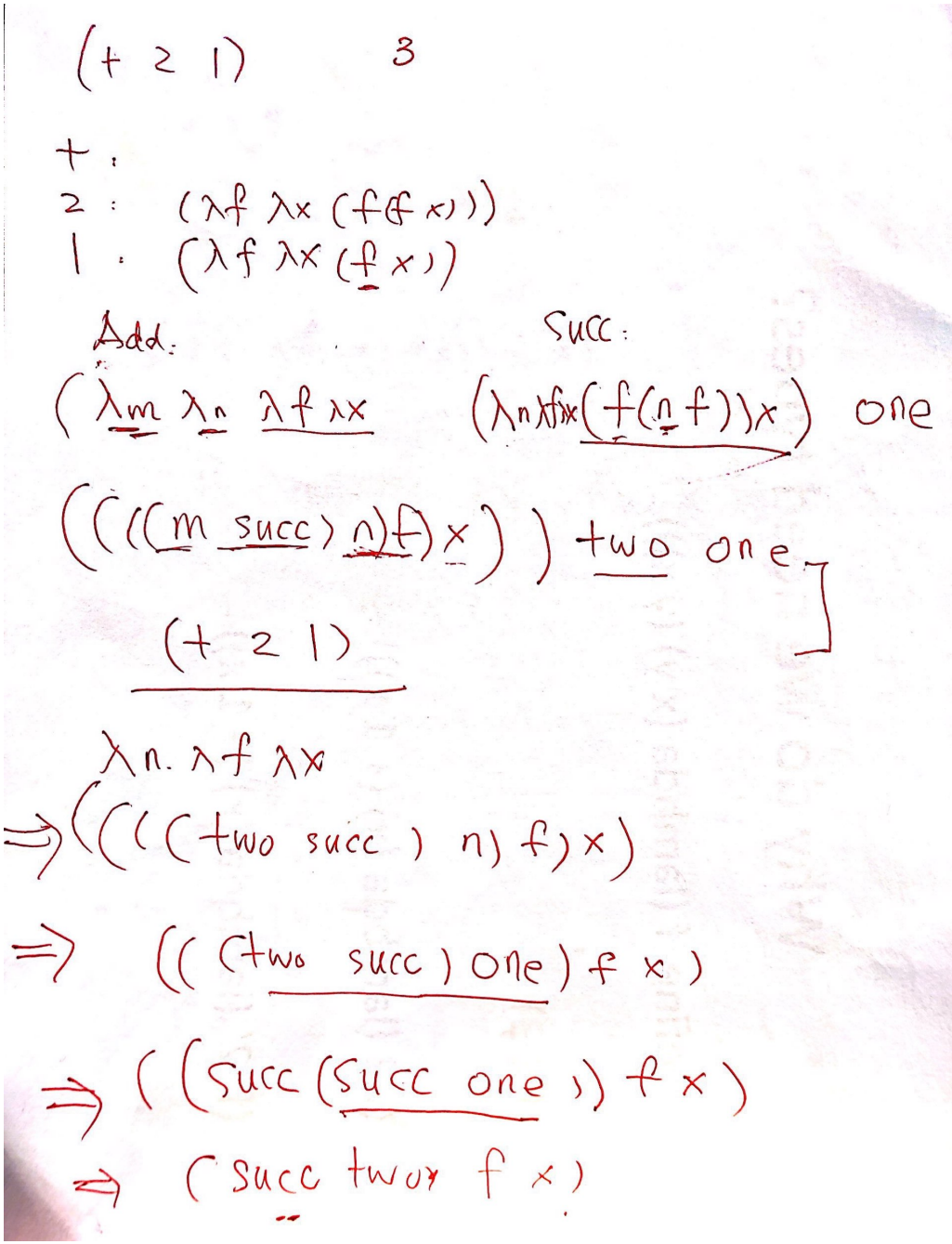
$(\lambda x.(\lambda y.y)\ \text{true})\ \text{true}$

$(\lambda y.y)\ \text{true}$

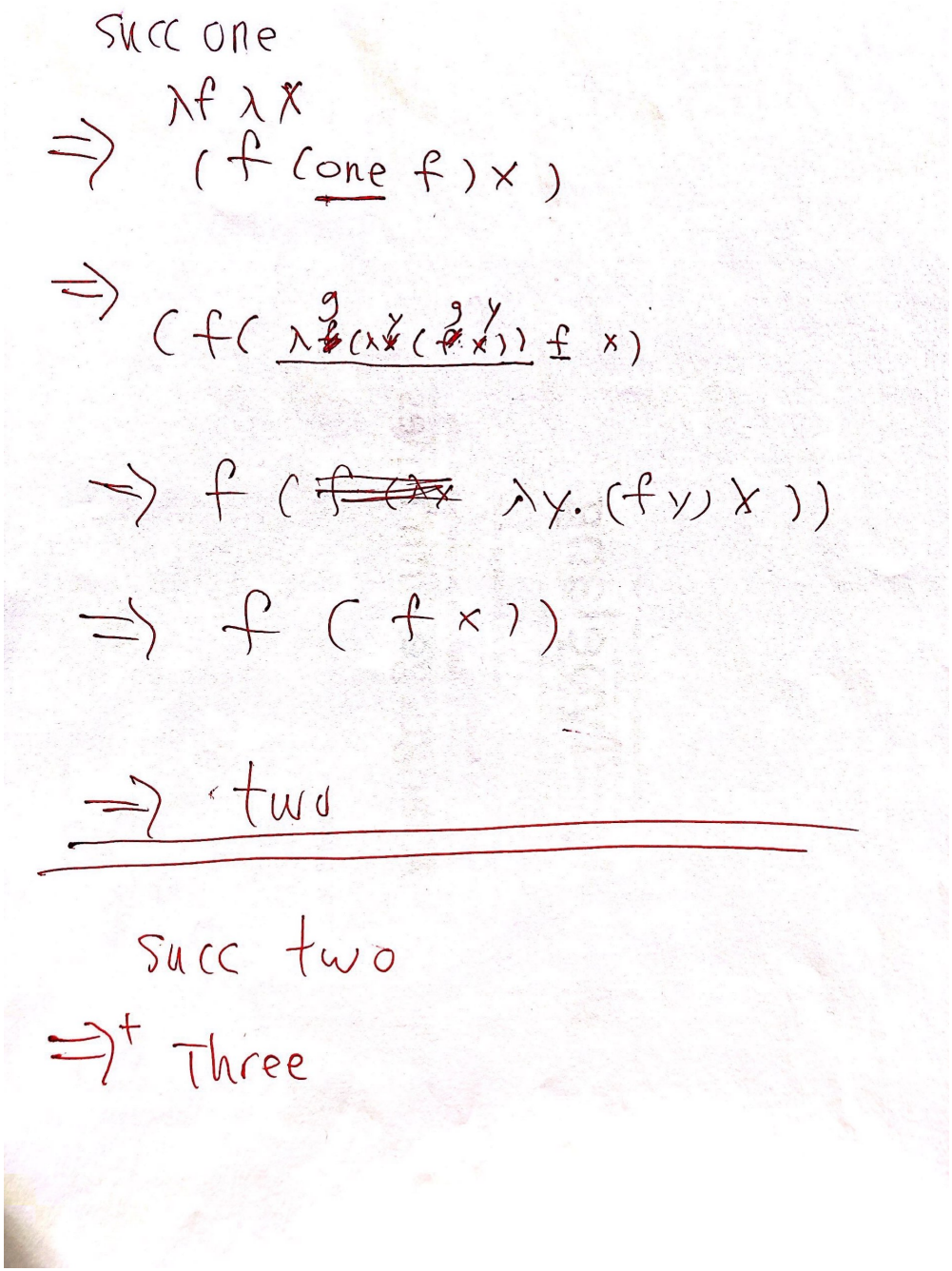
true

$((\lambda x.(\lambda z.zy))\ (\lambda z.z))$

Lambda Calculus Exercise



Lambda Calculus Exercise



RefLang

Side Effect

- Pure functional programs can be understood in terms of their input and output. Given the same input a functional program would produce the same output.
- Change the state of the program besides its output
- Examples:
  - Reading or writing memory locations
  - Printing on console, reading user input,
  - File read and file write,
  - Throwing exceptions,
  - Sending packets on network,
  - Acquiring mutual exclusion locks, etc...

Two Concepts

- **Heap**: an abstraction representing area in the memory reserved for dynamic memory allocation
- **References**: locations in the heap

Design Decisions – Heap

Heap size is finite, programming languages adopt strategies to remove unused portions of memory so that new memory can be allocated.

- manual memory management: the language provides a feature (e.g. free in C/C++) to deallocate memory and the programmer is responsible for inserting memory deallocation at appropriate locations in their programs.
- automatic memory management: the language does not provide explicit feature for deallocation. Rather, the language implementation is responsible for reclaiming ununsed memory (Java, C#).

How individual memory locations in the heap are treated:

- untyped heap: the type of value stored at a memory location is not fixed, can be changed during program execution
- typed heap: each memory location has an associated type and it can only contain values of that type, the type of value stored at a memory location doesn't change during the program's execution

Design Decisions – Reference (pointers)

1. Explicit references: references are program objects available to the programmer
2. Implicit references: references only available to implementation of the language
3. Reference arithmetic: references are integers and thus we can apply arithmetic operations
4. Deref and assignment only: get the value stored at that location in the heap, assignment can change the value stored at that location in the heap

Examples:

- C Programming language: manual memory management, explicit reference, untyped heap, reference arithmetic
- Java: automatic memory management, deref and assignment only, untyped heap, implicit reference
- Reflang: manual memory management, deref and assignment, untyped heap, explicit references

int a = 2

int * p = &a

*(p+20) = &(100)

malloc

free

* p = &i

p = p + q

*p = 2

i = *p

access object feature through a reference

s.tostring() = s.i

Deref S.i

S = new object()

Assignment S.i = 2

RefLang

- Expressions for allocating a memory location, dereferences a location reference, assign a new value to an existing memory location, free previously allocated memory location
- Examples:
  $(ref 1)
  loc: 0
  (ref 1)
- Value: the location at which memory was allocated (next available memory location)
- Side effect: assign value 1 to the allocated memory location
- Value and type are known from the expression

Reflang Expressions

ref: This expression evaluates its subexpression to a value, allocates a new memory location to hold this value, and returns a reference value that encapsulates information about the newly allocated memory location.

```
$ (define loc1 (ref 12))
// stores value 12 at some location in memory, creates a reference value
to encapsulate (and remember) that location, and stores that reference
value in variable loc1

$ (define loc2 (ref 45))

$ loc1 // check the reference value stored in variable loc1
loc:0

$ loc2
loc:1
```

Ref Exp

set! loc loc2

$ @- loc:10

loc:

9 : 1

loc 10

Reflang Expressions

deref: This expression evaluates its subexpression to a value. If that evaluation evaluates to a reference value, and that reference value encapsulates a location l, then it retrieves the value stored in Heap at location l.

```
$ (deref loc1) // gives the value stored at loc1
12

$ (deref loc2) // gives the value stored at loc2
45

$ (+ (deref loc1) (deref loc2)) //access both values and adds them
57
```

Reflang Expressions

assign: This expression is used to change the value stored on some location in Heap.

$ (set! loc1 23) //previous value 12 is overwritten by 23
23

$ (set! loc2 24) //previous value 45 is overwritten by 24
24

$ loc1 // loc1 still has address 0 but value has changed now
loc:0

$ loc2 // loc2 still has address 0 but value has changed now
loc:1

$ (+ (deref loc1) (deref loc2)) // different value different summation value
47

Reflang Expressions

free: This expression is used to deallocate the reference stored in Heap.

$ (free loc1) // deallocates the memory address 0

$ loc1 // variable loc1 still points to same location loc:0

$ (deref loc1) // dereference loc1
Error:null // invalid because memory location has been freed

$ (free loc2) // deallocates the memory address stored in loc2

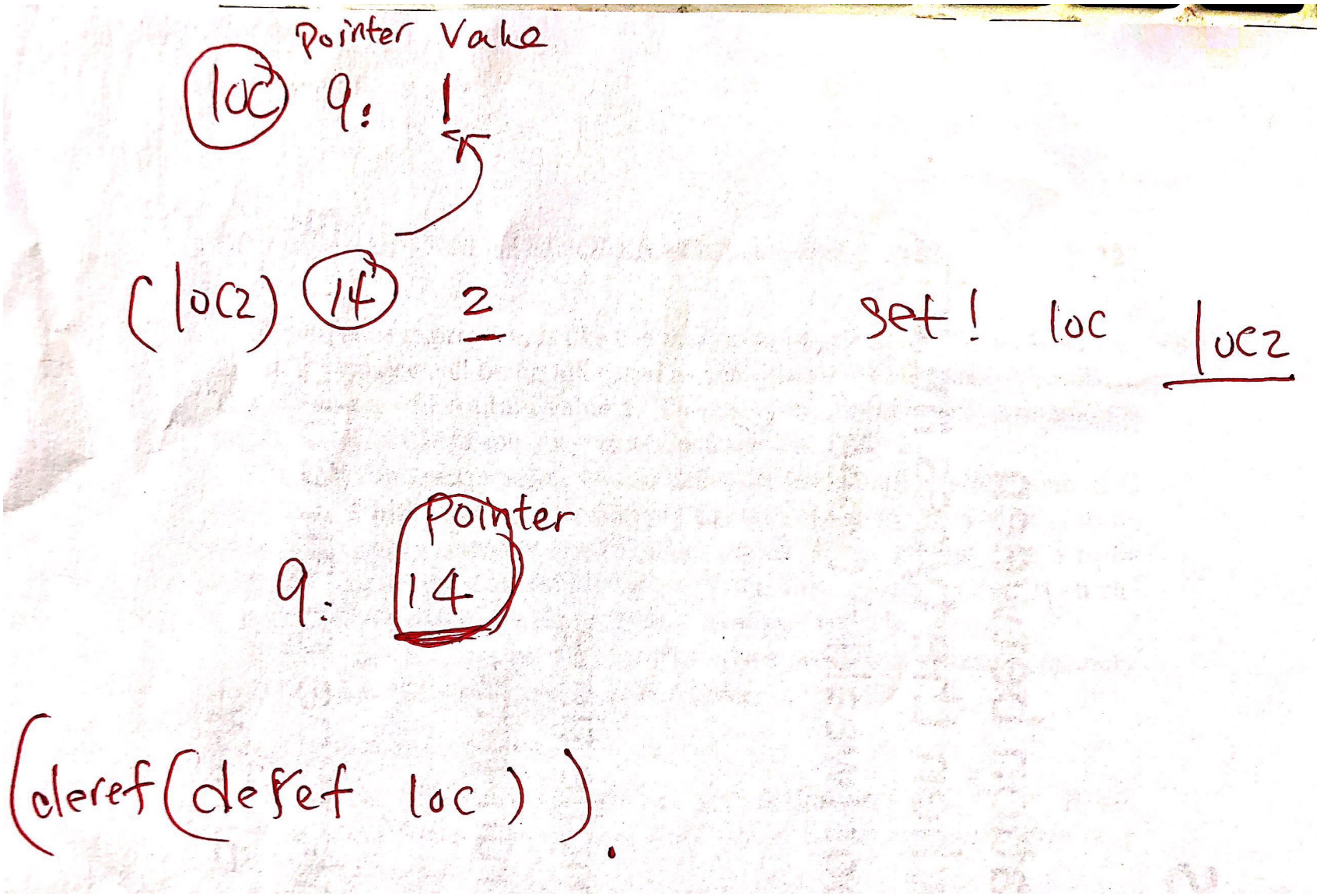
$ (deref loc2) // dereference loc2
Error:null // invalid because memory location has been freed

RefLang: More Examples

```
$ (free (ref 1)) // delocate the memory location where 1 is stored
$ (deref (ref 1)) // deref a memory location defined by ref 1
$ (let ((loc (ref 1))) (deref loc))
$ (let ((loc (ref 1))) (set! loc 2))
```

- ref, free, deref, set!

RefLang: More Examples



A reference expression is like the *malloc* statement in C, C++. It will result in a memory cell being allocated at the next available memory location. That location will contain value `1`. The value of the reference expression is the *location* at which memory was allocated, here `loc:0`.

The reference expression is also different from malloc statement in C and C++. Unlike malloc that accepts the size of the memory that is to be allocated, the argument of the reference expression is a value that is to be stored at the newly allocated location. From this concrete value, both the type of the value and the size required to store it can be derived.

In Reflang language we can explicitly free a previously allocated memory location using the *free expression* as follows.

```
$ (free (ref 1))
```

A free expression is like its namesake in languages like C, C++. It will result in a memory at the location that is the value of the expression `(ref 1)` to be deallocated.

We can also dereference a previously allocated memory location using the dereference expression

```
$ (deref (ref 1))
1
$ (let ((loc (ref 1))) (deref loc))
1
```

Dereferencing a memory location is a way to find out what is stored at that location. So a dereference expression takes a single expression, expression that evaluates to a memory location, and the value of the dereference expression is the value stored at the memory location.

We can also mutate the value stored at a memory location using the assignment expression

```
$ (let ((loc (ref 1))) (set! loc 2))
2
$ (let ((loc (ref 3))) (set! loc (deref loc)))
3
$
```

An assignment expression `set!` has two subexpressions, left hand side (LHS) expression that evaluates to a memory location, and right hand side (RHS) expression that evaluates to a value, and the value of RHS expres-

Reflang: Grammar

Program	::=	DefineDecl* Exp?	*Program*
DefineDecl	::=	(define Identifier Exp)	*Define*
Exp	::=		*Expressions*
		Number	*NumExp*
	|	(+ Exp Exp+)	*AddExp*
	|	(- Exp Exp+)	*SubExp*
	|	(* Exp Exp+)	*MultExp*
	|	(/ Exp Exp+)	*DivExp*
	|	Identifier	*VarExp*
	|	(let ((Identifier Exp)+) Exp)	*LetExp*
	|	(Exp Exp+)	*CallExp*
	|	(lambda (Identifier+) Exp)	*LambdaExp*
	|	(ref Exp)	***RefExp***
	|	(deref Exp)	***DerefExp***
	|	(set! Exp Exp)	***AssignExp***
	|	(free Exp)	***FreeExp***

Reflang: Extending Values

- RefVal $\neq$ NumVal
  - prevent from accessing arbitrary memory location
  - no arithmetics
  - extra meta data

RefLang: Heap abstraction

"Map"

Heap : RefVal $\rightarrow$ Value

```
1 public interface Heap {
2   Value ref (Value value) ;
3   Value deref (RefVal loc) ;
4   Value setref (RefVal loc, Value value) ;
5   Value free (RefVal value) ;
6 }
```

Reflang Expression Semantics

- Expression do not affect heap directly or indirectly:
1. Constant expression: value e env h = (NumVal n) h
   n is a Number, env is an environment, h is a heap
2. Variable expression – look up names for values: value (VarExp var) env h = get(env, var) h
- Indirectly affect heap through their subexpressions
- Directly affect heap

Stack

revisit all the expressions

Reflang Expression Semantics: Indirectly affect heap

- the order in which side efects from one subexpression are visible to the next subexpression has significant implications on the semantics of the defined programming language.
- Add expression: / Sub / Multiplication

$$\texttt{value (AddExp } e_0 \ldots e_n\texttt{) env h} = v_0 + \ldots + v_n, h_n$$

$$\texttt{if value } e_0 \texttt{ env h} = v_0\ h_0, \ldots, \texttt{value } e_n \texttt{ env } h_{n-1} = v_n\ h_n$$

$$\texttt{where } e_0, \ldots, e_n \in \texttt{Exp, env} \in \texttt{Env, h, } h_0, \ldots h_n \in \texttt{Heap}$$

a left-to-right order is used in the relation above for side-effect visibility

* An example of side effect in RefLang

inclass execise

Reflang Expression Semantics: Directly affect heap

- ref, set!, free
- deref: read from memory only

Reflang: RefExp

$$\text{value (RefExp e) env h} = l, h_2$$

$$\text{if value e env h} = v_0\ h_1$$

$$h_2 = h_1 \cup \{\ l \mapsto v_0\ \} \qquad l \notin \text{dom}(h_1)$$

where $e \in \text{Exp}$ $\quad \text{env} \in \text{Env}$ $\quad h, h_1, h_2 \in \text{Heap}$ $\quad l \in \text{RefVal}$

Heap is modified

Reflang: AssignExp

Set! Location.

value (AssignExp e_0 e_1) env h = v_0, h_3

if value e_1 env h = v_0 h_1 value e_0 env h_1 = l h_2

h_3 = { l ↦ v_0 } ∪ (h_2 \ { l ↦ _ }) l ∈ dom(h_2)

where e ∈ Exp env ∈ Env h,h_1,h_2,h_3 ∈ Heap l ∈ RefVal

range dom

f ref value

Location

V_0

↓

Reflang: FreeExp

$$\text{value (FreeExp e) env h} = \underline{\text{unit}},\ h_2$$

$$\text{if value e env h} = l\ h_1 \qquad l \in \text{dom}(h_1)$$

$$h_2 = h_1 \setminus \{\ l \mapsto \_\ \}$$

where $e \in \text{Exp}$ $\quad \text{env} \in \text{Env}$ $\quad h, h_1, h_2 \in \text{Heap}$ $\quad l \in \text{RefVal}$ $\quad \text{unit} \in \text{Unit}$

Reflang: DerefExp

$$\text{value (DerefExp e) env h} = v,\ h_1$$

$$\text{if value e env h} = l\ h_1 \qquad l \in \text{dom}(h_1)$$

$$\{\ l \mapsto v\ \} \subseteq h_1$$

where $e \in \text{Exp}$ $\text{env} \in \text{Env}$ $h, h_1 \in \text{Heap}$ $l \in \text{RefVal}$ $v \in \text{Value}$

Realizing Heap and Evaluators

See RefLang interpreter Code