# CS6124 Topics in Programming Languages.

edusevar enrollment key-6124

I Design a language

II Implement the language.

implementor must know the semantics of all constructs.

## Programming paradigms:

→ different ways of writing pgms. (Object oriented, functional... logical.

→ pattern of thought. for pgmmg.

⇒ We deal with functional pgmmg paradigms.

Lambda Calculus → model for computation

→ a small language with the core functionalities of functional pgmmg

## Type systems

→ Expressions take the type of value obtained from the expr. eval

## Language safety

→ pgms which are compiled correctly must give an o/p as expect

→ we use type system for language safety.

Develop an interpreter for func. language

Assignment 1 on Feb 4th

# Introduction to pgmng language:

## 1) Syntax:

→ rules to write code
→ grammar of a language.
→ lang. is a collection of sentences
→ syntax helps to make valid sentences.

→ C language → collection of all C programs.
(each line is a pgm)

→ Syntax gives rules to make valid sentences.

Eg:- → start symb.

$A \rightarrow aA \mid b$ $\quad a, b \Rightarrow$ terminal / alphabets.

$A \Rightarrow$ non terminal / variable.

$g1 = ab$
$aab$
$aaab$.
b

$L(G1) = \{b, ab, aab, aaab \ldots\}$

$G1 = (V, T, P, S)$
non. terminal prodn., Start symbols.

$a^*b \Rightarrow$ 0/more 'a' followed by b.
reg. expression

Eg:-

$E \rightarrow E + E \mid a$ $\quad$ terminals $= \{a, +\}$

$L(G) = \{a, a+a, a+a+a, \ldots\}$

BNF like notation for the same: Backus Naur form
$\langle expr \rangle ::= \langle expr \rangle + \langle expr \rangle \mid a$

Google: BNF

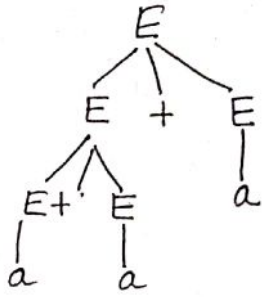
$E = E+E = E+(E+E)$

$= a+(E+E) = a+(E+E)$

$= a+a+a = a+a+E$

$= a+a+a$

②



$(a+a)+a$ ①

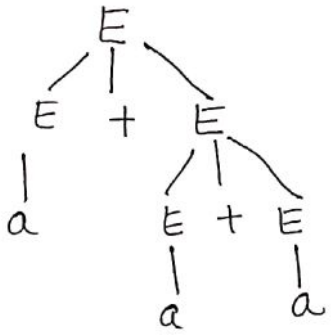


Yield of parse trees $= a+a+a$

$E \overset{*}{\Rightarrow} w$

w set of sentences derivable from E in 0/more steps ⇒ language

(concrete syntax trees)



$a+(a+a)$ ②

compiler designer will find it diff to implement it.

In ① & ② There is diff in order of evaluation steps in the m/c.

Operational meaning of the expression will be different.

(meaning understood by the m/c)

Program Semantics → o/p only matters (Denotational Semantics)

→ operational steps is the meaning. (Operational semantics)

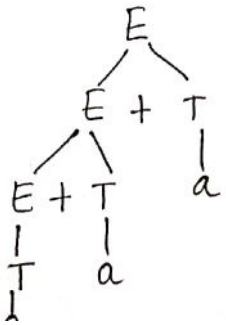
→ axiomatic semantics (proving pgm properties)

Grammar is ambiguous

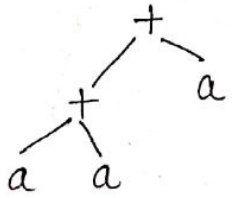
$E \to E+T \mid T$

$T \to a$

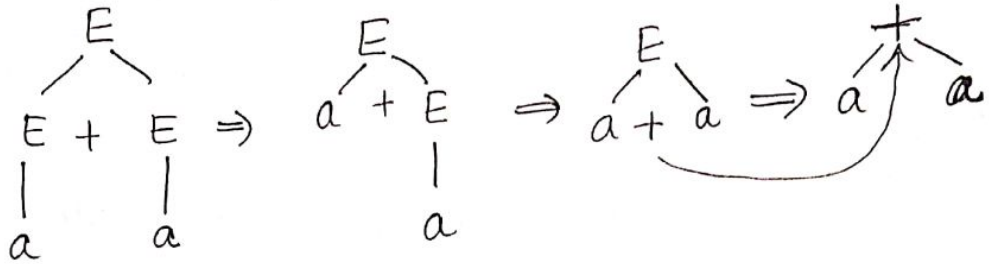
(left associative all expansions to the left.)



unambiguous.



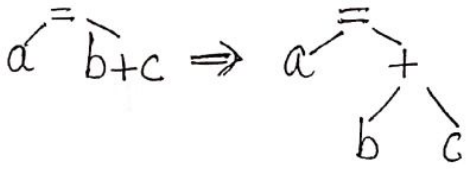
⇒ abstract syntax tree [AST]



push 'a' up.

Exercise Question

$a = b + c$



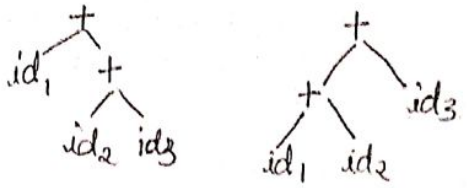
Grammar ⇒ meaning of constructs.

Backus Naur Form

It is a notational technique for CFG, often used to des
the syntax of languages used in computing

<expression> ::= <expression> + <term>
| <expression> − <term>
| <term>

01/20.




Panini
Chomsky (3)

Panini - Sanskrit philologist, grammarian & scholar "Father of linguistics"

Noam Chomsky - American linguist, philosopher, historian, social critic, & political activist. Still alive

① In denotational, its same meaning

② In operational, semantics is diff.

③ Axiomatic semantics.

The termination conditions will be specified.

Eg :-

(i) $x$ variable (abstraction of m/y loc)

(ii) $x = x+1$ LHS ⇒ m/y loc.

$x$ in RHS ⇒ value of $x$

Depending on L-value & R-value, the value of variables differ.

## Type System

type of var. meaning of exp # loc.

→ classification of values of a particular kind

→ associated with every type of values, there are some valid operations.

$x+y$ (−, *, %...)

$x$ and $y$ (or, not ...)

⇒ $x + y$ ⇒ compiler generates type error
(bool) (int) since + can be used with int / float only.

Typing rule ⇒ how to assign type to each construct.

Each $E = E_1 + E_2$.

If $E_1, E_2$ are int, E is also int

Exercise:

$x = y+z$

x is int, $(y+z)$ is int, what is the type of assignment?

02/01/2020, Thursday

→ Variables has type – type of value it stores.

→ Expression has type – type of value it computes.

→ Assignment stmt is not computing anything, but it is changing the state of the program.

(set of var with values) [set of values/variables]

type of value computed by assignment ?? no retn type for assignment

No value is associated with an assignment stmt.

∴ Type of assignment stmt is void

$E \rightarrow E_1 + E_2$. "attribute grammar"

{ if $E_1$.type = int and $E_2$.type = int then E.type = int
else E.type = type_error }

$E \rightarrow id$ { E.type = id.type }

Type of the id can be obtained from

① Declaration
② from the operations. [type inference]
③ Run time [late binding]

An entity is associated with an ~~value~~ attribute like name, value, ty
etc – Binding

Static Binding – at compile time
Dynamic Binding – at run time.

Scope of a binding – life time of the binding/variable.

Scoping rules → Static scope rules
→ Dynamic scope rules.

$S \rightarrow id = E$ { if id.type == E.type
then S.type = void
else S.type = type_error }

This info is used by semantic analyzer for type checking.
Type checking means checking whether the type of the constant is according to typing rules.

$E \rightarrow id[E_1]$ (array) { if $E_1$.type = int and
id.type = array ($T_1$ ...) ← extra information
then E.type = T
else E.type = type_error

$E \rightarrow id(E_1)$ (function) { if id.type = $T_1 \rightarrow T_2$ and (parameter, return type)
$E_1$.type = $T_1$
then E.type = $T_2$
else E.type = type_error }

read(a)

03/01/2020, Friday

## Programming Paradigms

→ different pattern of thought
→ different types
- → Functional
- → Procedural / imperative
- → Object Oriented

## Functional v/s Imperative

Example: Factorial of a number (without using a specific language)

imperative approach

Input : An integer number 'n'
fact = 1
i = 1
Repeat if $i \leq n$
fact = fact x i
i = i+1
return fact

$n! = (n-1)! \times n$ , if $n > 1$

$1! = 1$ , otherwise

if $n == 1$ , return 1

~~else~~ , return $n * fact(n-1)$ ] Functional approach

In 1930s, computational model is designed to solve and represe mathematical solutions

Turing machine is the underlying model for imperative pgmg app

I/p tapes → m/y storage

states → change in the variable values using assgnmt stm

Lambda Calculus is the underlying model for functional pgmg approach

→ Expressing the computations using functions.

→ λ (lambda) is used for defining func.

→ Function application (calling function)

Church Turing Hypothesis – Func on the natural no can be calculated by an effec method iff its computable by a Turing m/c.

$f(x) + f(x) \neq 2 * f(x)$ where f is a func & x is the i/p.

↓ this may change the state of the program

→ In pure functional pgmg lang, there is no notion of m/y stat

In this case $f(x) + f(x) = 2 * f(x)$

This property is called referential transparency.

→ Causing a change in state can be called as side effect of the fun

→ If there is side effect in a pgmg language, then it cannot guarantee "Referential transparency"

Imperative constructs are also present in func. lang. other than pure func. lang.

eg:- i/p, o/p statements change the state but still it may be there in func lang

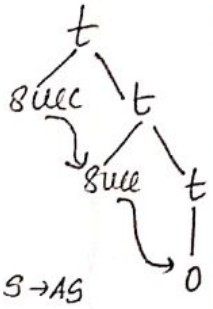
## Untyped Arithematic Expressions

### Syntax

Eg1:- t ::= // t is called term

0

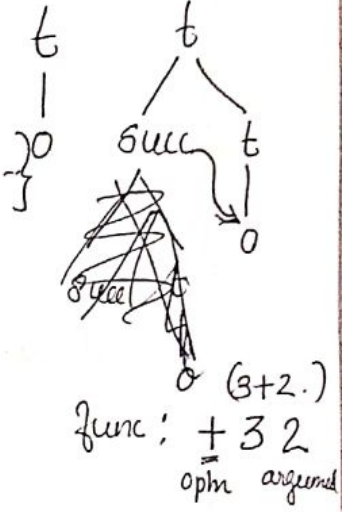
succ t // successor applied on t.
// succ is a func/operation.




S → AS

~~L = {0, 00, 000, ...}~~

L = {0, succ 0, succ succ 0, succ succ succ 0 ...}




⇒ succ applied multiple times onto 0.

Eg2: t ::=

0

succ t

pred t

L = {0, succ 0, pred 0, succ succ 0, succ pred 0, succ succ pred 0, succ pred pred 0, ...}

Eg3: t ::=

0

succ t

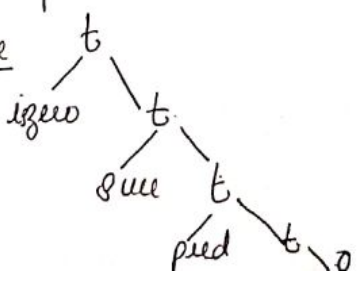
pred t

is zero t

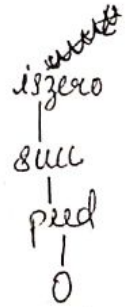
AST for t | 0 ⇒ 0 (concrete, abstract)

~~is~~ iszero succ pred 0

concrete tree




AST



Eg 4: t ::=
0
succ t
pred t
iszero t
true
false

Terms:
meta variable
meta lang.
object lang.

L = { 0, true, false.

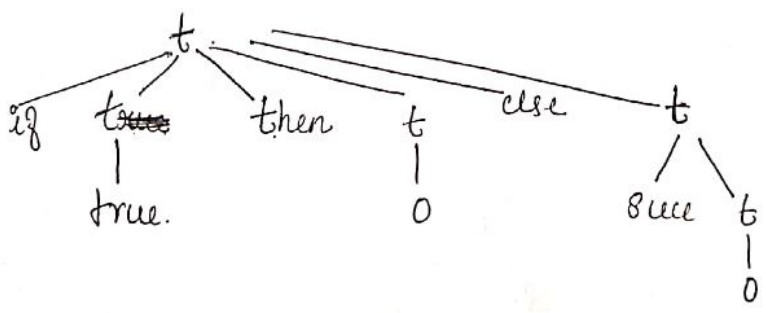
Eg 5: t ::= // meta variable "t" since we use it for describing language → object language.
0
true
false
if t then t else t // can be any term in the language //
succ t
pred t
iszero t

t ⇒ then not concrete

Samples:
if ~~true~~ iszero 0 then true else false.
if succ 0 then pred 0 else succ pred 0.
if true then 0 else false
if succ 0 then true else 0.
nested if.
if

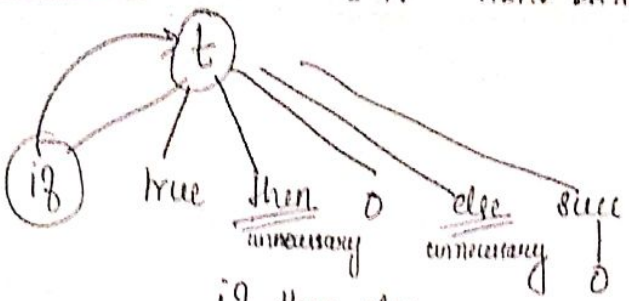
General term:
if $t_1$ then $t_2$ else $t_3$.
But concrete term will not contain $t_1, t_2, t_3 \ldots$.

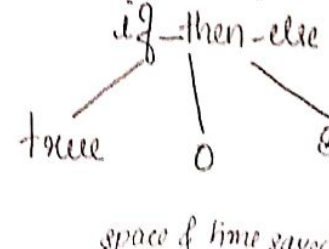
Draw parse tree: if true then 0 else succ 0.




Meta variable
→ metalinguistic varia
→ syntactical variable
⇒ it is a symbol / sym
which belongs to a meta language and stands for elts of a object language.
Let A, B ⇒ sentences of lang
Symbols A and B are of the metalanguage

Meta language
language used to describe another la
often called the object language.
The structure of sentences & phrase in a metalangu can be described b a metasynta




// AST : condensed form.

HW ⟹ Need for AST ??
- space & time saved
- we translate lang to intermed code - AST.
- independent of source & m/c lang. representation.
- can be reused - across platform using apt interpreters.

09/01/2020, Thursday

Front end translates source code → AST

Back end AST → m/c lang.

## Size of a term ( size (t) )

size(0) = 1

size(true) = 1

size (false) = 1

size (succ $t_1$) = size(t) + 1

size (succ succ 0) = 3

size (pred $t_1$) = size($t_1$) + 1

size (if $t_1$ then $t_1$ else $t_3$) = size($t_1$) + size($t_2$) + size($t_3$) + (1)

if-else-then is 1 opn

Notes:

In (succ $t_1$) ⇒ $t_1$ is called subterm.

If $t_1$ then $t_1$ else $t_2$ ⇒ 3 subterms ($t_1$, $t_2$, $t_3$)

× —— end of syntax —— ×

# Semantics

$t ::=$
- true
- false
- if $t$ then $t$ else $t$

Since there are no nos, we call it "Language of Booleans"

**Note:**
all terms are programs · eg true ⇒ program // trivial values.
false ⇒ "
if $t$ then $t$ else $t$ ⇒ pgm.

if (if true then false else true) then true else false.
// false //

Semantics gives the way in which pgm (term) is to be evaluated

operational ~~defn~~ semantics · steps of opn / evaluation of pgm

⇒ We require operational semantics.

[ evaluate $t_1$
if $t_1$ = true, evaluate $t_2$.
else evaluate $t_3$. ] meaning of prev. term.

---

Specification / Notation

Axioms:

(i) if true then $t_2$ else $t_3$. $\longrightarrow t_2$

["$\longrightarrow$" evaluates in one step]

(ii) if false then $t_2$ else $t_3$ $\longrightarrow t_3$

E-IF TRUE / E-IF FALSE

rules of how terms are evaluated "Evaluation Rules"

$t \longrightarrow t'$ : $t$ evaluates single step to

Inference rule:

(iii)
$$\frac{t_1 \longrightarrow t_1' \Rightarrow \text{precondition.} \quad \text{E-IF}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

Eg: if true then true else false $\longrightarrow$ true $\not\longrightarrow$
(evaluates to)
(final value of evaluation) can't be evaluated further.

| V ::= |
| --- |
| true |
| false |

Values of the Language

"V" ⇒ meta variable for value.

Notes: ⇒ Sometimes value itself may be recursive.

Guard then else.

Eg ① if true then false else true ⟶ false ↛ (E_IF TRUE; instance of the rule)

② if true then $t_2$ else $t_3$ ⟶ $t_2$

③ if (if true then false else true) then false ~~then~~ else true.

(since $t_1$ itself is a term, so evaluate $t_1$ first.)

~~if (if $t_1$ then~~ ↓ instance of E_IF

$$\frac{}{\text{if true then false else true} \longrightarrow \text{false}}\ \text{E\_IF TRUE}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}\ \text{E\_IF}$$

if (if true then false else true) then false else true

($t_1$ $t_2$ $t_3$)

⟶ if false then false else true.

($t_1'$ $t_2$ $t_3$)

E_IF FALSE ⟶ false ~~FALSE~~ ⟶ ⟶ ?! true.

Exercise:

diff semantics for the same terms.

(1) if true then $t_2$ else $t_3$ ⟶ $t_3$ E_IF TRUE

(2) if false then $t_2$ else $t_3$ ⟶ $t_2$. E_IF FALSE

(3) $$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}\ \text{E\_IF}$$

either change operational order can be changed

(or) ~~order~~ change meaning

15/01/2020, Wednesday

E_IF TRUE
E_IF
} evaluation rules ⇒ this lang. has only 2 rules.
E_IF FALSE is not specified.
Ok diff. semantics, but incomplete

(iv) $\frac{t_2 \to t_2'}{\text{if true then } t_2 \text{ else } t_3 \to \text{if true then } t_2' \text{ else } t_3}$ → diff semantics.
→ ambiguity. if given true we can use (i) or (iv)

---

In case of Lang of booleans.

V ::= true
false

(specification of values) says what are the end values expected

(specification of evaluation rules)

---

→ We design the lang for an abstract machine
It is a state transition m/c. It reaches a final state ↔ final value
initial state ⇒ term to be evaluated.

if true then (if true then true else false) else false.

initial state → (i) $t_2 \to t_2'$

if true then true else false → if

(i) E_IF TRUE → if true then true else false.
↓ second

(ii) E_IF TRUE → true ↛
↑ final

Assume Evaluation rules: E_IF TRUE
E_IF → { ∴ programmer might have ignored the "if false" part }

the m/c will not yield final state for some terms.
it will halt but that may not be final state. term.
eg: segmentation fault, divide by 0. // run time errors

⇒ The lang. is syntactically correct but semantically incomplete

⇒ The m/c will reach a "stuck state"

The error due to stuck state ⇒ segmentation fault.

⇒ Such a language is not safe as we can't predict its behaviour.

e.g. $t \xrightarrow{*}$ if false then $t_2$ else $t_3 \not\rightarrow$

not defined — stuck state.

⇒ a term t is in Normal form ⇒ when we can't evaluate the term further.

Defn:

A term t is in normal form, if there is no $t'$ such that $t \rightarrow t'$.

Note: every value is in normal form. (1st Property of Language with E_IF, E_IF TRUE, E_IF FALSE)

every stuck state is also NF!!

NF without value = stuck state.

## Properties of Language of Booleans

1. Every value is in normal form.
2. If a term t is in NF then t is a value.
3. Determinacy of one step evaluation.
   (there is one rule that is applicable)

   If $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$

   (one way — alternate)
4. Corollary of (3) : Uniqueness of Normal forms.
   (if reach a final NF, that state will be unique.)

   If $t \xrightarrow{*} u$ and $t \xrightarrow{*} u'$ where u and u' are normal forms then $u = u'$

   (values are same.)
5. Evaluation of every term will terminate (Termination of evaluation)
   (i.e. no infinite evaluation)

$t ::=$
- true
- false
- or $t$ $t$

Term in the language: ~~or true false~~ ~~or $t_1$ false~~ | Concrete term doesn't have brackets

or (or true false) (or (or true true) false)

## Semantics

I$^{st}$ part values: $V ::=$ true | false

II$^{nd}$ part values. Evaluation rules:

$t_1 \to$ true  can be applied for or true $t_2 \to$ true

evaluate as per requirement

"Short circuit Evaluation

Semantics 2.

(i) or true true $\to$ true.
or true false $\to$ true
or false true $\to$ true
or false false $\to$ false.

"Complete evaluation Semantics"

if all subterms are evaluated

$$\frac{t_1 \to t_1'}{\text{or } t_1\ t_2 \to \text{or } t_1'\ t_2}$$

$$\frac{t_2 \to t_2'}{\text{or } v_1\ t_2 \to \text{or } v_1\ t_2'}$$

$$\frac{t_2 \to t_2'}{\text{or true } t_2 \to \text{or true } t_2'} \qquad \frac{t_2 \to t_2'}{\text{or false } t_2 \to \text{or false } t_2'}$$

Short circuit AND

(ii) Semantics 1 (or)

or true $t_2 \to$ true
or false $t_2 \to t_2$

$$\frac{t_1 \to t_1'}{\text{or } t_1\ t_2 \to \text{or } t_1'\ t_2}$$

$$\frac{t_2 \to t_2'}{\text{or } t_1\ t_2 \to \text{or } t_1\ t_2'}$$

$t_1$ can be another wrong in term order

HW

"Short Circuit Evaluation Scheme" for AND

$t ::=$ true
false
AND $t$ $t$

$v ::=$ true
false

Evaluation rules: AND true $t_2 \rightarrow t_2$ ——— (1)

AND false $t_2 \rightarrow$ false ——— (2)

$$\frac{t_1 \rightarrow t_1'}{\text{AND } t_1\ t_2 \rightarrow \text{AND } t_1'\ t_2} \quad \text{——— (3)}$$

17/01/2020, Friday

Step of evaluation

if (if true then true else true) then true else true

↓ E_IF // E_IF TRUE, E_IF FALSE can't be applied.

if true then true else true

↓ E_IF TRUE

true

↓

$$\frac{\text{if true then true else true} \rightarrow \text{true}}{\text{if (if true then true else true) then true else t} \rightarrow \text{if true then true else tru}}$$

Here step of evaluation E_IF has subterms that needs to be evaluated we assume that there is an abstract m/c to calculate subterms.

We can use derivation trees

no premise → ——— E_IF TRUE

axiom if true then true else true → true ——— E_IF

if (if true then true else true) then true else true → if true then true else true

E_IF TRUE

if true then true else true → true ↛

if (if (if true then true else true) then true else true) then true else h

Derivation tree for step #1

---------- E_IF TRUE
if true then true else true → true

---------- E_IF
if (if true then true else true) then true else true → if true then true else

---------- E-IF
if (if (if true then true else true) then true else true) then true else t

→ if (if true then true else true) then true else tr

Step #2.

if true then true else true → true

---------- E-IF
if (if true then true else true) then true else true → if true then tru

Step #3.


---------- E_IF TRUE
if true then true else true → true

In evaluation derivation tree the root is at the bottom.

$$\frac{\dfrac{}{t_2 \to t_2'}\ \text{leaf.}}{t_1 \to t_1''} \qquad \text{each level is a node.}$$

$$\frac{t_1 \to t_1''}{t \to t'}\ \text{root}$$

## Proof of Properties of Lang of Booleans.

I no need proof

II if $t$ is in NF, then $t$ is a value.

Contrapositive of the property:

if $t$ is not a value then $t$ is not a NF.

We will prove contrapositive.

→ if $t$ is not a value, then $t$ is of the form; if $t_1$ then $t_2$ else $t_3$

→ Case 1: if $t_1$ = true, E-IF TRUE can be applied.

$t \to t_2$ . $t$ is not NF as a rule could be applied

Case 2: if $t_1$ = false, E-IF FALSE can be appled.

$\therefore t$ is not NF.

Case 3: ~~If $t_1 \to t_1'$ \ E-IF~~

$t_1 \neq$ true, $t_1 \neq$ false or $t_1$ is not a value.

we use structural induction to prove this

$P(t)$ for a term $t$

For each assume $P(x)$ implies $P(t)$
subterm $x_i$

Hypothesis: For each subterm $t_i$ of $t$, if $t_i$ is not a value, then $t_i$ is not in NF (or $t_i \to t_i'$ for some $t_i'$)

20|01|2020, Monday

If $t_1$ is neither true/false, then $t_1$ is not a value. Then by induction hypothesis $t_1$ is not a normal form. so there is some $t_1'$ such that $t \to t_1'$. Then $t \to$ if $t_1'$ else $t_2$ then $t_3$. so $t$ is not a normal form.

③ Termination of evaluation

Every evaluation reduces the size of the term by one. So at some pt, the size reaches 1 (size of value) and evaluation terminates.

④ Determinacy of one step evaluation

→ depends on structure of derivation-tree.

Hypothesis: If each subterm $t_1$ satisfies the property "if $t_1 \to t_1'$ and $t_1 \to t_1''$ then $t_1' = t_1''$, the term t also satisfies $t' = t''$

Assume 2 derivation trees

Tree #1: $\dfrac{}{t \to t'}$

Tree #2: $\dfrac{}{t \to t''}$

Case 1: Assume the rule E-IF TRUE applied at the root of Tree #1. Then t is of the form 'if true then $t_2$ else $t_3$', then $t'$ will be $t_2$. The only rule that can be applied is E-IF TRUE $t'' = t_2$

Case 2: Then t is of the form 'if false then $t_2$ else $t_3$'. Then $t'$ will be

Assume the rule E-IF FALSE applied at the root Tree#1. is

The only rule that can be applied at the root of Tree #1 is E-IF FALSE. Hence $t'' = t_3 = t'$

Case 3 Assume the rule E-IF applied at the root of Tree #1

Then, t is of the form 'if $t_1$ then $t_2$ else $t_3$'

with $t_1 \to t_1'$

Then, $t'$ will be 'if $t_1'$ then $t_2$ else $t_3$'

Tree #2, $t_1 \to t_1''$. But as per hypothesis $t_1' = t_1''$.

Hence, $t'' =$ if $t_1'$ then $t_2$ else $t_3 = t'$.

## Semantics of untyped Arithematic expression

Numbers in the language can be represented using succ and pred

succ 0 = 1

succ succ 0 = 2

succ succ succ 0 = 3

pred 0 = -1
pred pred 0 = -2
} -ve nos are not considered as values in this language.

* It is not possible to include succ v in values, since it includes succ true, succ false etc. Since these are not values in the language categorize the values.

nv ::=
    0
    succ nv

V ::=
    true
    false
    nv

Evaluation rules of is_zero.

① iszero 0 → true (E_ISZEROZERO)

② iszero succ $nv_1$ → false (E_ISZEROSUCC)

③ $$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'}$$ (E_ISZERO)

eg :- evaluate iszero (if true then 0 else succ 0)

Step #1

$$\frac{\dfrac{}{\text{if true then 0 else succ 0} \rightarrow 0}\ \text{E\_IFTRUE}}{\text{iszero(if true then 0 else succ 0)} \rightarrow \text{iszero 0}}\ \text{E\_ISZERO}$$

Step #2

$$\frac{}{\text{iszero } 0 \rightarrow \text{true}} \text{ E\_ISZEROZERO}$$

Evaluation rules of succ and pred

④ $$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'}$$ E_SUCC

⑤ $$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'}$$ E_PRED

⑥ pred 0 → 0 E_PREDZERO

⑦ pred succ $nv_1 \rightarrow nv_1$ E_PRED SUCC

24/1/2020, Friday

Type Systems

→ iszero $t_1$ : If it is possible to say that $t_1$ will be evaluated to numeric value, then only iszero is applied.

→ If $t_1$ evaluated to numeric, then the type of $t_1$ is numeric and if $t_1$ evaluates to boolean, then the type of $t_2$ is bo

→ With the help of the type system, it is possible to approxima predict the type of the value, not the exact value.

# OCAML

(Objective Categorical Abstract Machine language)

→ supports functional, imperative and object oriented styles.

→ includes an interactive top level interpreter, a byte code compiler and an optimizing native code compiler.

refer: cs.cornell.edu

Datatypes:

int, bool, float, string, char.

→ Ocaml is statically typed and type safe pgmng lang.

C is not type safe.

22|1|2020, Wednesday

① Evaluate iszero (succ pred 0)

Step #1

$$\frac{}{\text{pred } 0 \to 0}\ \text{E\_PREDZERO}$$

$$\frac{\text{pred } 0 \to 0}{\text{succ pred } 0 \to \text{succ } 0}\ \text{E\_SUCC}$$

$$\frac{\text{succ pred } 0 \to \text{succ } 0}{\text{iszero (succ pred } 0) \to \text{iszero (succ } 0)}\ \text{E\_ISZERO}$$

Step #2

$$\frac{}{\text{iszero (succ } 0) \to \text{false}}\ \text{E\_ISZEROSUCC}$$

→ Some of the stuck terms in the language

⇒ succ true

⇒ iszero false

⇒ pred succ false.

→ Do we need pred 0 → 0?

This will cause 'pred 0' also as a stuck ~~term~~ term.

② Evaluate if(succ pred 0) then true else false

Step #1:

$$\frac{}{\text{pred } 0 \to 0}\ \text{E\_PREDZERO}$$

$$\frac{\text{pred } 0 \to 0}{\text{succ pred } 0 \to \text{succ } 0}\ \text{E\_SUCC}$$

$$\frac{\text{succ pred } 0 \to \text{succ } 0}{\text{if (succ pred 0) then true else false} \to \text{if (succ 0) then true else false.}}\ \text{E\_IF}$$

Step #2

if (succ 0) then true else false $\not\to$

The term results in a stuck ~~term~~.

→ It is also possible give some semantics for the stuck term.

for eg:- succ true → false

But the semantics doesn't have a logical meaning.

→ So usually such semantics are not included in the langu

→ Inorder to distinguish b/w valid terms and terms which donot have any meaning (eg: succ true), its better to a such semantics. But it can be evaluated to some error.

→ Need an additional check to predict the runtime behaviour the program. But it is not possible to predict the actual thing but an approximate behaviour.

For eg, the term may result in a stuck term. This is provided by the type system of the language.

Note: Language of untyped arithematic expression doesnt satisfy property 2. if a term t is in NF, then t is a value.

## Examples for failure of language properties.

① Assume the following rule is also included.

iszero (succ pred nv) → iszero nv.

eg:- ⓐ iszero (succ pred 0) → iszero 0 → true.

ⓑ iszero (succ pred 0) → iszero (succ 0) → false

ⓐ - using the new rule.

ⓑ - using the earlier rule.

This fails property ③ and ④ uniqueness of NF / Determinacy of 1 step era

② Assume the following rule.

succ true → succ true

The term size is not reducing in this rule, which violates property 5.

## Use of formal semantics

① Compiler / interpreter implementor can directly implement the semantics given.

② If the semantics are not complete [ie. semantics for some constructs are not specified], then implementor can take their own assumptions, which make one implementation different from another.

21/1/20 Friday

## Type systems

→ iszero $t_1$ : If it is possible to say that $t_1$ will be e
to numeric value, then only iszero is applied.

→ If $t_1$ is evaluated to numeric, then the type of $t_1$ is
and if $t_1$ evaluates to boolean, then the type of $t_2$ is

→ With the help of type system, it is possible to approxim
predict the type of the value, not the exact value.

→ method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

→ Static approximation to the runtime behaviour of the terms in a pgm.

→ Uses:
- detecting errors
- abstraction.
- Documentation.
- Efficiency.

Classify the values into different types.

T ::= Bool
      Nat
} types in the language.

→ Typing rules can be used to assign types to terms.

→ Typing rules for values.

① true : Bool (T_TRUE)

② false : Bool (T_FALSE)

③ 0 : Nat (T_ZERO)

succ nv : Nat ⇒ This is not required. Bcoz it is redundant when compared to rule 6.

④ $\dfrac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}}$ (T_ISZERO)

eg:- iszero 0

$$\dfrac{\dfrac{}{0 : \text{Nat}}\ T\_ZERO}{\text{iszero } 0 : \text{Bool}}\ T\_ISZERO$$

} Typing derivation tree

**General Notation**

$t : T$

The term t "is typeable as" T

eg:- iszero true — "ill-typed" term

If a term is typable, then it is called "well-typed"

⑤ $\dfrac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$ (T_PRED)

~~eg:- iszero pred 0~~

Eg: iszero pred 0

$$\dfrac{\dfrac{\dfrac{}{0 : Nat}\ T\text{-}ZERO}{pred\ 0 : Nat}\ T\text{-}PRED}{iszero\ pred\ 0 : Bool}\ T\text{-}ISZERO$$

(6) $$\dfrac{t_1 : Nat}{succ\ t_1 : Nat}\ T\text{-}Succ$$

eg:-

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 : Nat}\ T\text{-}ZERO}{pred\ 0 : Nat}\ T\text{-}PRED}{succ\ pred\ 0 : Nat}\ T\text{-}Succ}{pred\ succ\ pred\ 0 : Nat}\ T\text{-}PRED}{succ\ pred\ succ\ pred\ 0 : Nat}\ T\text{-}Succ$$

(7) $$\dfrac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : T}\ T\text{-}IF$$

$t_2$ & $t_3$ have same type then only one can statically type it as T. else stuck terms can occur.

If the type of $t_1$ and $t_2$ are same, the type of the term can be statistically predicted.

eg:- If true then 0 else false

This term is classified as 'ill-typed', even though it can be evaluated.

Some stmts which are not typable may/may not get stuck.

eg: if true then 0 else false..

$t_1 \xrightarrow{*} v_1$ & $v_1 : T$.

but we dont know what exact value is $v_1$. known only after evaluation.

we can only say $t_1 : T$. without knowing actual value.

Q: Draw the typing derivation tree:

if (iszero succ pred 0) then true else false.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 : Nat}\,T\_ZERO}{pred\ 0 : Nat}\,T\_PRED}{succ\ pred\ 0 : Nat}\,T\_SUCC}{iszero\ succ\ pred\ 0 : Bool}\,T\_ISZERO \qquad \dfrac{}{true : Bool}\,T\_TRUE \qquad \dfrac{}{false : Bool}\,T\_FALSE}{if\ (iszero\ succ\ pred\ 0)\ then\ true\ else\ false : Bool}\,T\_IF}{if\ (iszero\ succ\ pred\ 0)\ then\ true\ else\ false}$$

There are 3 branches for the derivation tree.

## Properties of type system:

1) **Uniqueness of types**: ~~Every~~ if a term is typable, its type is unique.

Every term that is typable has a unique type.

Every term has atmost 1 type.

Note: ill typed will have no type.

2) **Type Safety**: [PROGRESS + PRESERVATION]

⇒ "Well typed terms are well-behaved"

↓ typable ↓ runtime behaviour.

⇒ If a term t is typable, then either t is a value or $t \to t'$ (one step evaluation)

⇓

PROGRESS

PRESERVATION: If $t : T$ and $t \to t'$, then $t' : T$ (need not be same type, just typable. means it doesn't get stuck.)

i.e. $t'$ is also typable

Preservation doesn't say anything about $t'$

In every step, typability is preserved.

06|02|2020, Thursday

## Proof for Type Safety.

1) PROGRESS

If $t : T$, then either t is a value or $t \to t'$

If T_IF is applied at the root:, it is of the form.

if $t_1$ then $t_2$ else $t_3$ .. when T_IF : T.

$t_1 : Bool \Rightarrow t_1 : T$

$t_2 : T \Rightarrow t_2 : T$

$t_3 : T \Rightarrow t_3 : T$

As per type, $t_1$ is either a value or $t_1 \to t_1'$

case a) $t_1$ is a value, $t_1$ = TRUE

$t_1$ is of the form if true then $t_2$ else $t_3$, E_IFTRUE applies.

$t \to t_2$.

Tutorial.

case (a) $t_1$ is a value, $t_1$ = FALSE

t is of the form: if false then $t_2$ else $t_3$, E_IFFALSE applies

ie. $t \rightarrow t_3$.

case (b)

$t_1 \rightarrow t_1'$

E_IF applies ie. $t_1 \rightarrow t_1'$

if $t_1$ then $t_2$ else $t_3 \rightarrow$

if $t_1'$ then $t_2$ else $t_3$.

Another possible term using T_PRED;

t is of the form (pred $t_1$)

pred $t_1$ : Nat; that is $t_1$ : Nat..

ie. $t_1$ is typable.

$t_1 \rightarrow t_1'$

E_PRED rule can be applied so pred $t_1 \rightarrow$ pred $t_1'$.

t is a value: E_PREDZERO & E_PREDSUCC can be applied.

PRESERVATION:

If t : T and $t \rightarrow t'$ then t' : T

term is of the form: if $t_1$ then $t_2$ else $t_3$.

↓E-IF

if $t_1'$ then $t_2$ else $t_3$.

according to precondition $t_2$ & $t_3$ are typable.

so the term (if $t_1'$ ....) is typable. takes the type of $t_2$ or $t_3$.

If we Remove E_PREDZERO from rules. What is its effect on type safety?

pred 0 → 0.

(T_PRED) E_PREDZERO.

term is typable but evaluation rule is missing, cnt be evaluated.

So it becomes a stuck term.

pred pred 0 ⟹ unsafe. (stuck term).

13/02/2020, Thursday

- same term can have multiple types

eg:- 0, 1 may be bool or int.

C1
|
C2 objects have same type.

- downcasting C1 base
↓
C2 derived.

when C1 is used in a place where $C_2$ was expected, the additional member of $C_2$ will be missing. Safety issue.

- upcasting is safe.

Ques: Extend Language of Typed arithematic s/m to include OR, AND, NOT.

① $$\frac{t_1 : bool \quad t_2 : bool}{OR\ t_1\ t_2 : bool}$$ T_OR.

② $$\frac{t_1 : bool \quad t_2 : bool}{AND\ t_1\ t_2 : bool}$$ T_AND.

③ $$\frac{t_1 : bool}{NOT\ t_1 : bool}$$ T_NOT

# Lambda Calculus

- a model for computation
- computation in terms of fns

## Notations used :

fn y . y * 2.

fn x . x + 1

⇓ replace fn using λ

λ y . y * 2.

λ x . x+1

Header Body

Function defn. without name with parameter x can be further evaluated its value.

header body

f(x) = x+1

⇓

f x = x+1

⇓

f x . x+1

header body

→ instead of .

- (λ x . x + 1) 2 // fn is applied onto 2. Func defn cnt be evaluated.

Function call | invocation application

Func call. can be evaluated.

- (λx . λy . x+y) 2 → λy . 2+y // returns fn as value of a fn. "higher order fn"

(λx . λy . x+y) 2 3 → (λy . 2+y) 3

→ // λy . 2+3 //

→ 5

## Syntax of "pure" Lambda Calculus

### Language of untyped λ calculus

$t ::=$

$x$ variable

$\lambda x . t$ abstraction header: $\lambda x$ body: $t$

$t\ t$ application

Write concrete terms in λ calculus.

1) $x$ ✓ — $x$
2) $\lambda x . t$ ✗ — $\lambda x . x$
3) $x\ x$ ✓ — $x x$
4) $\lambda x : \lambda x . t$ — $\lambda x\ x\ x$
5) $\lambda x . t . x$ — $(\lambda y . y)\ x$
6) $\lambda x . \lambda x . t\ t$ — $x x x x$
7) $\lambda x . \lambda x . x$
8)

— Value here is an abstraction.

— $(\lambda x . x)\ a \rightarrow a$ // a represents some abstraction not variable.
any

— $(\lambda x . x)(\lambda y . a) \longrightarrow \lambda y . a \not\longrightarrow$
// we replace $x$ with $\lambda y . a$ //

— $\lambda x . x$ } same func., diff variables.
$\lambda y . y$ } result is the argument itself.
↳ Identity function

— Constant function
$\lambda y . a$ // constant function ~~which~~ returns 'a' which can be evaluated further (another abstraction as well)

$(\lambda x.t_{12})\ t_{12}$

an abstraction followed by variable, it can be further evaluated.

form of redex.

$(\lambda x.t_{12})\ t_{12} \rightarrow [x \mapsto t_{12}]\ t_{11}$ // "Reduction"

// substitute $t_{12}$ (arg. in abstraction) in every occurence of $x$ in $t_{11}$

action done in reduction :- Substitution.

Reducible expression [Redex]

- can be further reduced.

-

Eg:- $(\lambda x.x)(\lambda y.y)$

$\rightarrow [x \mapsto \lambda y.y]\ x$

$= \lambda y.y \not\rightarrow$

② $(\lambda x.xx)(\lambda y.y)$

$\rightarrow$ ~~$\lambda x.x.xx \rightarrow \lambda y.y$~~

$\rightarrow [x \mapsto \lambda y.y]\ x\ x$

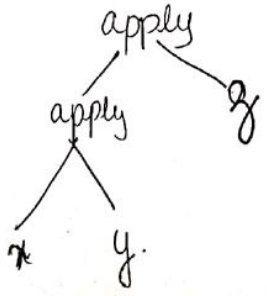
$= (\lambda y.y)(\lambda y.y)$

$\rightarrow [y \mapsto \lambda y.y]\ y$

$= \lambda y.y \not\rightarrow$

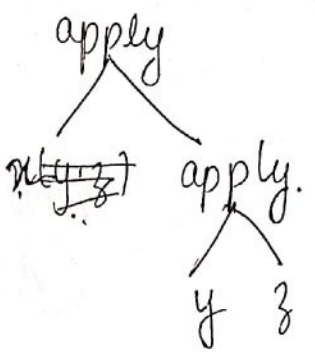
Associativity of applications usually "left associative!!"

$x\ y\ z$ means $(x\ y)\ z$.




AST of $x\ y\ z$

$x(y.z)$



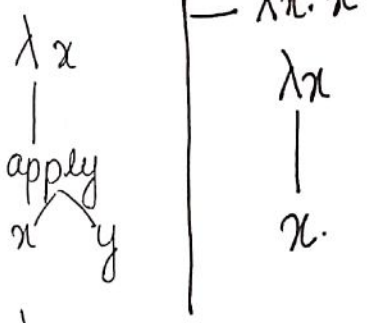

**Note:**

An application can be evaluated only if an abstraction is followed by an application.

$(\lambda x.\ t_{11})\ t_{12}$

$xy \Rightarrow$ is an appln but cant be evaluated.

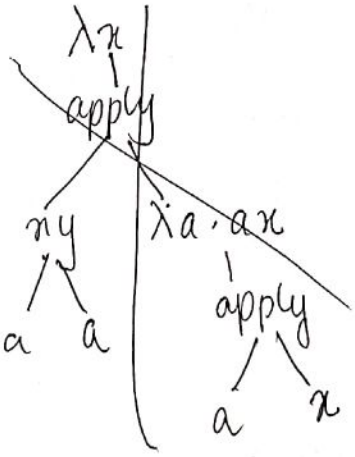
$\lambda x.xy$

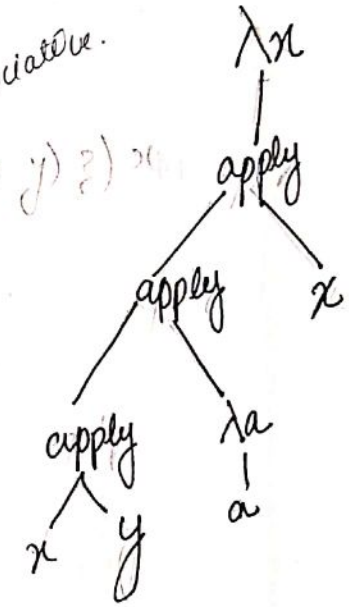



$\lambda a.a$


$\lambda x.xy\ \lambda a.a\ x \rightarrow$ abstraction. applications are left associative.

$(\lambda x.xy)(\lambda a.a)x \rightarrow$ application.


if parenthesis is introduced,

$(\lambda x.xy)(\lambda a.a)x \Rightarrow$ application.

$\lambda x . xy(\lambda a . a)\, x$ put parenthesis to avoid ambiguity

**Binding & Scope of binding**

$\lambda x\ x\ y\ x$ → free.

binder, bound variable.

Note: $(\lambda x . x)\, x$ → free; bound

→ body of the abstraction is the scope.

→ every occurence of binder variable (here $x$) are bound occurences.

Eg:- $\lambda x . xy\ \lambda y . y$

free occurence; bound occurence.

14/02/2020, Friday

abstraction

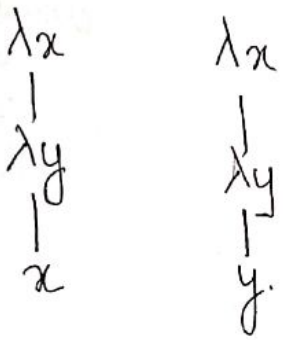
$\lambda x . \lambda y . x$ → $x$ is bound variable as it comes under $\lambda x$.

abstraction

$\lambda x . \lambda y . y$ → $y$ also

$\lambda x . x$

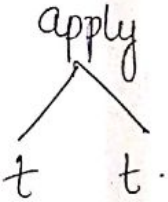
$\lambda x . a$ = constant




body of $\lambda x = \lambda y . x$ here $x$ is free. so it can be substituted

**abstraction**


**application**




$(\lambda x . \lambda y . x)\, u\ v$

$t_1$ $t_2$ $t_3$

evaluation will be in order $(t_1 t_2) t_3$

identify redex starting from left.

$(\lambda x . \lambda y . x)\, u\ v$ // underline redex (body)

→ $([x \mapsto u]\ \lambda y . x)\, v$ // $x$ is replaced by $u$ in $\lambda y . x$ (body)

$x$ is free variable

→ $(\lambda y . u)\, v$ entire term is redex

→ $[y \mapsto v]u = u$ ↛

replacement is for free occurence of var.

$(\lambda x.\lambda y.y)\,u\ v$

$\rightarrow ([x \mapsto u]\lambda y.y)\,v$

$= (\lambda y.y)\,v$

reduced $\rightarrow [y \mapsto v]\,y$

$= v \not\rightarrow$

- Two abstractions : $\lambda x.\lambda y.x$ // always returns 1st argument
  $\lambda x.\lambda y.y$ // always returns 2nd argument.

  any variable can be used ; $\lambda t.\lambda f.t$, $\lambda t.\lambda f.f$ } same as above.

- $(\lambda t.\lambda f.t)\,u\ v \xrightarrow{*} u$
- $(\lambda t.\lambda f.f)\,u\ v \xrightarrow{*} v$

Encoding Languages using abstraction :
Reduce every argument and then substitute.

Encode language of Booleans [Encoding of Booleans] Church Booleans

t ::=
  true // abs
  false // abs
  if t then t else t. // abs

In λ calc., value is an abstraction.
so, true & false are abstractions.

Encoding for conditional : use term "test"

test = $\lambda l.\lambda m.\lambda n.\ l\ m\ n$ — 3 abstractions, body

test u v w $\xrightarrow{*}$ either v | w depending on u.

test tru v w $\xrightarrow{*}$ v

test fls v w $\xrightarrow{*}$ w

test u v w = $(\lambda l.\lambda m.\lambda n\ l\ m\ n)\ u\ v\ w$

$\rightarrow (\lambda m\ \lambda n\ u\ m\ n)\ v\ w$

$\rightarrow (\lambda n\ u\ v\ n)\ w$

$\rightarrow u\ v\ w \not\rightarrow$

test tru v w $\xrightarrow{*}$ tru v w $\xrightarrow{*}$ v

test fls v w $\xrightarrow{*}$ fls v w $\xrightarrow{*}$ w

test = $\lambda l.\lambda m.\lambda n.\ l\ m\ n$ → conditional

tru = $\lambda t.\lambda f.\ t$ → true

fls = $\lambda t.\lambda f.\ f$ → false.

Eg :- if true then false else true

Encoding in $\lambda$ ⇒ test tru fls tru.

$= (\lambda l.\lambda m.\lambda n\ l\ m\ n)$ tru fls tru.

$\rightarrow (\lambda m.\lambda n$ tru $m\ n)$ fls tru

$\rightarrow (\lambda n$ tru fls $n)$ tru

$\rightarrow$ tru fls tru..

$= (\lambda t.\lambda f.\ t)$ fls tru

$\rightarrow (\lambda f.$ fls $)$ tru

$\rightarrow$ fls $\not\rightarrow$

## Encoding for AND, OR, NOT

t ::=
- true
- false
- AND tt.

Abstraction with 2 arguments = $\lambda a.\ \lambda b.$ a b fls.

and tru tru
tru fls
fls fls
fls tru
} evaluation ??

and = $\lambda b.\ \lambda c.\ b\ c$ fls

gvn b and c (bool) returns c if b is tru, returns fls if b is fls.

and tru tru; = $\lambda t.\ \lambda f.\ t$

and tru fls; = $\lambda t.\ \lambda f.\ f$

OR : $\lambda a.\ \lambda b$ a tru b.

NOT : ~~$\lambda a.$ a fls tru~~ $\lambda a$ a fls tru

24/08/2020, Thursday @ 9.00am

Church Numerals

$C_2 = \lambda s.\lambda z.s(s\ z)$

apply s on z.

apply s on (s z)

i.e. Two application of s on z.

$C_3 = \lambda s.\lambda z.\ s(s(s\ z))$

$C_1 = \lambda s.\lambda z.s\ z$

$C_0 = \lambda s.\lambda z.z$

Similarly $C_{10}$ is 10 applications of s on z.

$C_{10} = \lambda s.\lambda z.s(s($

→ $C_n\ f\ g$ is an application of f, n times on g.

→ apply 1st arg 3 times on second $\lambda s\lambda z.s(s(s\ z))) \Longrightarrow C_3$.

$C_2\ f\ g = (\lambda s.\lambda z.s(s\ z))\ f\ g$

$= (\lambda z.\ f(f\ z))\ g$

$= f(f\ g)$

→ $(\lambda s.\lambda z.\ C_2\ f\ g)\ u\ v = \lambda s.\lambda z.(\lambda s.\lambda z\ s(s\ z))\ f\ g)\ u\ v$

new scope can't replace with u

~~$= \lambda s.\lambda z.((\lambda z.u\ u\ z)\ f\ g)\ v$~~

$= \lambda s.\lambda z.(f(f\ g))$

$= f(f\ g)$ [2 appln of f and g]

Another way.

$= (\lambda s.\lambda z.(\underbrace{\lambda s'.\lambda z'.s'(s'z')}_{\text{new scope}})\ f\ g)\ u\ v$

all bound variable must be replaced consistently

we can replace the variables as "α-conversion"

$= (\lambda s.\lambda z.(\lambda a.\lambda b.a(a\ b)\ f\ g)\ u\ v$

Renaming of variables : $\alpha$-conversion.



28/2/2020, Friday @ 10.15 am

id (id ($\lambda z$. id z))

redex.

$\lambda x \cdot x$ → identity fn.
$\lambda x \cdot a$ constant fn.

→ we can start with the innermost redex "id z"

→ Evaluation strategy of the above form is called β reduction.

id = $\lambda x \cdot x$
Use another name like tru, ffs etc.

## λ Calculus-Evaluation Strategies

1) Full β reduction – Choose any order for redex.

id (id ($\lambda z$. id z)) ⟶ id (id ($\lambda z \cdot z$))

innermost first ⟶ id ($\lambda z \cdot z$)

⟶ $\lambda z \cdot z \not\rightarrow$

id (id ($\lambda z$. id z)) ⟶ id ($\lambda z$ · id z)

⟶ $\lambda z$. id z

outermost first ⟶ id z

~~⟶~~ $\lambda z \cdot z \not\rightarrow$

2) Normal order → Choose outermost redex.
if there are two outermost, choose the leftmost
ie. leftmost outermost redex

3) Call-by-name → Order is as in normal form.
if redex is within abstraction, then it will not be done.

id (id ($\lambda z$. (id z))) ⟶ $\lambda z$. id z ⟶ reduced.
abstraction
appears inside abstraction, so it will not be

We use Call by value.

4) Call-by-value → before evaluating "apply" evaluate the arguments first.

id (id ($\lambda z$. id z)) ⟶ id ($\lambda z$ · id z)

⟶ $\lambda z$ · id z $\not\rightarrow$

Call by need : evaluated only when substitution need (lazy evaluation)

1) Defn.

2) Show typing rule trees.

3) Yes, terms are not typable. addition of these terms will not affect type safety.

4)

5) to do

6) $t_1\ t_2$.

abstraction.

$\lambda x\ \lambda y\ x\ y$

8) 3 trees.

| | | |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

$t_1 \rightarrow b$

XOR $t_1\ t_2 \rightarrow$ xor $t_1\ t_2$

V.

---

$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$

Evaluation doesn't terminate.

"Non terminating evaluation"

→ Evaluate using Call by name: $\lambda y.z((\lambda x.xx)(\lambda x.xx))$ — constant fn.

$\rightarrow z$

Call by value:

$\lambda y.z((\lambda x.xx)(\lambda x.xx)) \rightarrow$ non terminating argument.

$\rightarrow \lambda y.z((\lambda x.xx)(\lambda x.xx))$

Note: C by value we evaluate argument even if not reqd. by in call by name we substitute and then evaluate.

C. by value, we evaluate the argument all the time.

→ ① Consider. $C_2 = \lambda s.\lambda z.s(s\ z)$ → Church numeral.

$\xrightarrow{*} f\ f\ g.$

② $(\lambda s.\lambda z.C_2\ s\ z)\ f\ g \rightarrow \lambda s.\lambda z.(\lambda s.\lambda z.(s(s\ z)))\ f\ g.$

$\rightarrow \lambda s.\lambda z.$

Abstractions ① and ② are behaviourly equivalent to $C_2$.

$\rightarrow f\ f\ g. \not\rightarrow$

$\lambda s.\lambda z.\ s(C_2\ s\ z)\ f\ g \rightarrow \lambda s.\lambda z.\ s(f\ f\ g)$ (63)

$\rightarrow \lambda s.\lambda z.\ s(s\ s\ z)\ f\ g$ (three appln of s to z.)

$\rightarrow f\ f\ f\ g$ //

→ Take 'n' as argument write a abstraction fn which returns behaviourly equivalent of $C_n$.

$\lambda n.\lambda s.\lambda z.\ n\ s\ z$

$\lambda s.\lambda z.\ \lambda\ \ n\ s\ z$

$\lambda n.\ n \rightarrow z$

$(\lambda n.\ \lambda s.\lambda z.\ n\ s\ z)\ C_n$

$\lambda s.\lambda z.\ C_n\ s\ z$

Abstraction given an Church numeral $C_n$ returns $C_{n+1}$

$\lambda n.\lambda s.\lambda z.\ s\ n\ s\ z$

$\rightarrow \lambda n.\lambda s.\lambda z.\ s(n\ s\ z))\ C_2 \xrightarrow{*} \lambda s.\lambda z.\ s(C_2\ s\ z)$

→ Succesor func: $scc = \lambda n.\lambda s.\lambda z.\ s(n\ s\ z)$

→ Write $C_{m+n}$ as $C_m$ and $C_n$.

$plus = \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z)$

ways encoded in λ.

$\lambda n.\ n(\lambda x.\ f\ s)\ n$

$f_s$ and $C_0$ are equivalent behaviourly. but they can be obtained by α-conversion.

HW
iszero

02|03|2020, Monday @ 8.00am

$C_2\,(\lambda x.x)\,tru \rightarrow C_2\; \dot{x}\; tru.$
$\rightarrow tru \not\rightarrow$
$C_2\,(\lambda x.x)\,fls \xrightarrow{*} fls \not\rightarrow$
} for any $C_n$.

f, g ⇒ all are functions in λ

$\rightarrow C_2\,(\lambda x.a)\,tru. \rightarrow a$ since $\lambda x.a$ ⇒ constant func.

$\rightarrow C_0\,(\lambda x.a)\,tru \rightarrow \lambda s.\lambda z.z\,(\lambda x.a)\,tru.$ } only $C_0$ gives tru, all others
$\xrightarrow{*} tru \not\rightarrow$

$\rightarrow C_2\,(\lambda x.fls)\,tru \xrightarrow{*} fls.$ —— ①
$C_0\,(\lambda x.fls)\,tru \xrightarrow{*} tru.$ —— ②

→ iszero. $\lambda n.n\,(\lambda x.fls)\,tru$ [This reduces to ① or ② depending on "n"

Pred func

Pair encoding in λ calculus

pair → constructor
pair → projections

| Projections |
| --- |
| fst, snd. (to pair |

(u, v) is a pair, its encoded as $\lambda b.b\,u\,v$ ← entity with 2 components in the body of abstraction.

→ ~~Constructor~~ pair = $\boxed{\lambda f\,\lambda s\,\lambda b.b\,f\,s.}$ [constructor of a pair].

pair v w $\xrightarrow{*}$ $\lambda b.b\,v\,w.$

→ Projections : fst and snd.

$fst = \lambda p.p\,tru$
$snd = \lambda p.p\,fls.$

Sample pair
$\lambda b.b\,C_1\,C_2.$

Evaluate : $fst\ (\lambda b.\ b\ c_1\ c_2)$

$= (\lambda p.\ p\ tru)(\lambda b.\ b\ c_1\ c_2)$

$\rightarrow (\lambda b.\ b\ c_1\ c_2)\ tru$

$\rightarrow tru\ c_1\ c_2 \rightarrow c_1$ ↛

~~$snd\ (\lambda p.\ p\ fls) = \lambda$~~

$snd\ (\lambda b.\ b\ c_1\ c_2) = (\lambda p.\ p\ fls)\ (\lambda b.\ b\ c_1\ c_2)$

$\rightarrow (\lambda b.\ b\ c_1\ c_2)\ fls$

$\rightarrow fls\ c_1\ c_2 \rightarrow c_2$ ↛

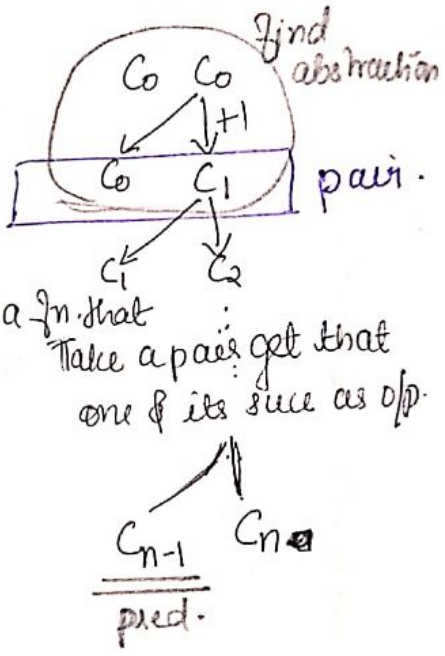
Consider,

→ $zz = pair\ c_0\ c_0$

~~pred~~ ss $= \lambda p.\ pair\ (snd\ p)\ (plus\ c_1\ (snd\ p))$

pair

generalized. ss ⇒ a fn. that Take a pair get that one & its succ as o/p.




→ $\lambda m.\ m\ ss\ zz$

m is a church numeral.

ss is applied m times on zz.

Result: $c_{m-1}\ c_m$.

$pred = \lambda m.\ fst\ (m\ ss\ zz)$

→ Subtraction : $m - n$

$\lambda m.\lambda n.\ n\ pred\ m$

Think : What if $m < n$ ??

→ $plus = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$

$plus\ c_2 = \lambda n.\ \lambda s.\ \lambda z.\ c_2\ s\ (n\ s\ z)$

$plus\ 2 = \lambda n.\ \lambda s.\ \lambda z.\ c_2\ s\ (n\ s\ z)$ // adds 2 to given ch. numeral

→ Higher order fns: takes fn as arg. & returns fn as result.

~~λ(m,n)~~

04|03|2020, Wednesday @ 9.00am

Formal Semantics

$t ::= x \mid \lambda x.t \mid tt$

$v ::= \lambda x.t$

Eg: $t_1 t_2$ "Left associative"

$t_1 \rightarrow$ apphn

$t_2 \rightarrow$ apphn.

Call by name.

Rule ①

Rule ② X

Rule ③ $(\lambda x.t_{12})t_2 \rightarrow [x_1 \rightarrow v_2]t_{12}$

1) $$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}\ E\_APP1$$

2) $$\frac{t_2 \rightarrow t_2'}{v_1 t_2 \rightarrow v_1 t_2'}\ E\_APP2$$

3) $(\lambda x.t_{12})v_2 \rightarrow [x.t \rightarrow v_2]t_{12}$. E_APPABS →

$(v_1 v_2)$ then we write it as $(\lambda x.t_{12})v_2$

$\lambda x$ is removed

all occurrence of $x$ becomes free and it replaced by $v_2$.

Issue in (3) some bound var may become free or some free → bound after substitution. This can be avoided by α-conversion.

$\lambda x.x\,x\,v_2$

$\lambda x.x\,y\ \lambda y.y$

Eg: ① $(\lambda y.\lambda w.y)w \rightarrow \lambda w.w \not\rightarrow$ identity fn.

2 α-conversions

② $(\lambda z.\lambda w.z)w \rightarrow \lambda w.w \not\rightarrow$ w is getting bound.

so replace $\lambda w \rightarrow \lambda x$

③ $(\lambda y.\lambda x.y)w \rightarrow \lambda x.w \not\rightarrow$ so that w is free.

Refer text : substitution semantics.

# TYPED LAMBDA CALCULUS

## Simply Typed Lambda Calculus.

There is only one type in pure λ calculus, so we have added true false, if t then t else t so that there some values to talk abt-

t ::=
- true
- false
- if t then t else t.
- x
- λx:T.t // type of x is specified as 'T' → argument.
- tt

V ::=
- true
- false
- λx:T.t

T ::=
- Bool
- T→T // → 'Type constructor

T→T→T... possible so cardinality '∞'

~~λx:T.t~~ Eg:- concrete term in language.
λx:Bool.x

In C.
x = f(y)
return f
arguments must be proper.
int→Bool.

Sample types
- Bool
- Bool→Bool
- (Bool→Bool)→Bool
- (Bool→Bool)→(Bool→Bool)

→ λx.Bool.true = Bool→Bool

→ if x then y else λz:Bool.z // another concrete term.

→ λx:Bool, λy:Bool, true.

Type
- Bool→(Bool→Bool) ✓ — In: takes Bool returns an abstraction.
- (Bool→Bool)→Bool ✗ — arg is abstraction returns a bool

→ In case of '→' : its "Right associative"

06/03/2020 Friday @ 10.15am

## Typing Rules

$\lambda x : Bool . true$ $\quad Bool \to Bool$

$\lambda x : Bool . x$ $\quad Bool \to Bool$ x is bound

$\lambda x : Bool . y$

y is external scope.

$(\lambda x : Bool . x)\, true$ // its an appln, "Bool"

$t_1\, t_2$

$t_1 : T_{11} \to T_{12}$

$t_2 : T_{11}$

$t_1 t_2 : T_{12}$

## Typing rule

① $$\frac{t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{t_1 t_2 \to T_{12}}$$

05/03/2020, Thursday @ 2.00pm

1) $\lambda x : Bool, x \; : Bool \to Bool$

we assume that x is bound by $\lambda x$, so its Bool

2) $\lambda x : Bool . true \; : Bool \to Bool$

3) $\lambda x : Bool . y \; : Bool \to$

we need a typing assumption regarding y.

$\lambda y : Bool . (\lambda x : Bool . y)$ // bound by another abstraction.

gamma.

$\Gamma \vdash t : T$

→ $y : Bool \vdash \lambda x : Bool . y : Bool \to Bool$.

typing context - assumption regarding the free variable.

→ $y : Bool \to Bool \vdash \lambda x : Bool . y : Bool \to Bool \to Bool$

by default, right associative

$x\ y$ — How to make it typable

① Assumption $x : Bool \to Bool, y : Bool \vdash xy : Bool$

② Alternate $x : Bool \to Bool \to Bool, y : Bool \vdash xy : Bool \to Bool$

③ $x : \underbrace{(Bool \to Bool)}_{T_1} \to \underbrace{Bool}_{T_2}, y : \underbrace{Bool \to Bool}_{T_1} \vdash xy : Bool$

$T_1$ type of argument. → takes Bool→Bool returns Bool so Bool

=

Typing assumptions is represented as $\Gamma$.

① $\Gamma_1 \vdash xy : Bool$

② $\Gamma_2 \vdash xy : Bool \to Bool$

③ $\Gamma_3 \vdash xy : Bool$

→ $\Gamma ::= \phi \mid \Gamma, x : T$ → if there are no free variables in term ("closed term") Assumption can be $\phi$

→ We should always mention a typing context.

→ Typing rule with typing context

① $$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$ T_APP

under the typ. context $\Gamma$, the term $t_1 t_2$ is typed as T_APP

② $$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$ T_ABS

subderivation.

Eg

$$\frac{\Gamma, x:T_1 \vdash x : \text{Bool}}{\Gamma \vdash \lambda x:\text{Bool}.\, x :} \text{T\_ABS}$$

$\Gamma$ extended with $x:T_1$

③ $$\frac{x : T_1 \in \Gamma}{\Gamma \vdash x : T_1} \text{T\_VAR}$$

only when its present in typing context.

Eg:-

$\Gamma_1 = x : \text{Bool}, y : \text{Bool} \to \text{Bool}$

$\Gamma_2 = x : \text{Bool} \to \text{Bool}, z : \text{Bool}$

$$\frac{x : \text{Bool} \in \Gamma}{\Gamma_1 \vdash x : \text{Bool}} \qquad \frac{x : \text{Bool} \to \text{Bool} \in \Gamma_2}{\Gamma_2 \vdash x : \text{Bool} \to \text{Bool}}$$

Typing context / Typing Environment

is a set of assumptions regarding the type of free variables.

$x : T_1, y : T_2, z : T_3 \ldots\ldots$

In general a term in $\lambda$ calculus is typed as : $\Gamma \vdash t : T$

Eg:-

① Given $\Gamma_1 = x : \text{Bool} \to T_1, y : \text{Bool}$

$$\frac{\dfrac{x : \text{Bool} \to T_1 \in \Gamma_1}{\Gamma_1 \vdash x : \text{Bool} \to T_1}\text{T\_VAR} \quad \dfrac{y : \text{Bool} \in \Gamma_1}{\Gamma_1 \vdash y : \text{Bool}}\text{T\_VAR}}{\Gamma_1 \vdash xy : T_1}\text{T\_APP}$$

06/03/2020, Friday @ 10.15am

Typing rules for true, false, if t then t else t.

4) $\Gamma \vdash true : Bool$ [T_TRUE]

5) $$\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 :}\ [T\_IF]$$

6) $\Gamma \vdash false : Bool$ [T_FALSE]

Eg:- $(\lambda x : Bool . x)\ true$

It is a closed term (no free occurences) so $\phi$

$$\dfrac{\dfrac{\dfrac{x:Bool \in \phi, x:Bool}{\phi, x:Bool \vdash x : Bool}\ T\_VAR}{\phi \vdash \lambda x:Bool . x : Bool \to Bool}\ T\_ABS \qquad \dfrac{}{\phi \vdash true : Bool}\ T\_TRUE}{\phi \vdash (\lambda x:Bool . x)\ true : Bool}\ T\_APP$$

if $\phi$ is absent its emphy content.

$$\dfrac{\dfrac{\dfrac{x:Bool \in x:Bool}{x:Bool \vdash x : Bool}\ T\_VAR}{\vdash \lambda x:Bool . x : Bool \to Bool}\ T\_ABS \qquad \dfrac{}{\phi \vdash true : Bool}\ I\_TRUE}{\vdash (\lambda x:Bool . x)\ true : Bool}\ T\_APP$$

Eg:- Give a typing context for $(\lambda x : Bool . x)\, y$

Ans: $\Gamma = y : Bool$ [$\lambda x . x$ requires $y$ to be Bool]

If $g$, context = ___?

$\Gamma = f : \overset{T_{11} \to T_{12}}{\cancel{\lambda x : Bool}}$ , $g :- T_{11}$ ✗ give concrete types

$\Gamma_1 = f : Bool \to Bool , g : Bool.$

$\Gamma_2 = f : Bool \to Bool \to Bool , g : Bool$

$\Gamma_3 = f : (Bool \to Bool) \to Bool , g : Bool \to Bool$

(instance of T_APP rule.)

→ If x then λy: Bool. y else z.

understood from abstraction.

$T_1$ = x: Bool ~~y: Bool → Bool~~ z: Bool → Bool

abstraction

→ context for f g h

can't apply

$T_1$ = f: Bool → Bool g: Bool ~~h: Bool~~

(f g) h

abstraction

$T_2$ = f: Bool → Bool → Bool g: Bool h: Bool

## Type inference

x + y

↓ int addition

so x & y must be int. [algorithms are used for the same.

## Type safety

→ PROGRESS + PRESERVATION

→ PROGRESS

A closed well typed term is either a value or it takes a step of evaluation.

ie - lang doesn't say nyth about free variables.

- variables are discarded.

application $t_1 t_2$. is well typed only when $t_1$ if $t_2$ are typable under same T.

→ PRESERVATION

If $T \vdash t : T$, and $t \to t'$ then $T \vdash t' : T$

Proof :- Substitution preserves types

### Special case

1) x x

not typable

"ill-typed"

2) (λx: Bool. x)

(λx: Bool. x

of the form tt

" not typable

3) true true

↓ has to be abs.

so ill-typed

(λx. x x)(λx. x x)

"divergent terms"

ill typed.

(λ: Bool. x x)(λx: Bool. x x)

↳ ill typed

9/3/2020 Monday @8:00am.

# Programming constructs

## The let binding

t ::= .... // same as before.

let x = t in t

eg:- let $x = t_1$ in $t_2$. ——— (1)

↳ // name $t_1$ as x and use it in $t_2$.

Concrete term for (1) $\lambda y : Bool . y$

let $x = \lambda y : Bool . y$ in $x\,a$

another one (2): let x = true in $(\lambda y : Bool . y)\, x$

eg (3) → let $a = \lambda x : Bool . x$ in a true

→ let x = true in → x.

let y = false in → another let binding

if x then y else false.

→ another valid term : let $x = \underbrace{(\lambda y : Bool . y)}_{t_1}$ true in $\underbrace{x}_{t_2}$.

o/p ::= true.

→ To replace x, first evaluate x, then replace the occurence.

→ Call by value.

x, replace and then evaluate → call by name.

→ Evaluate: call by value.

Step 1: $$\frac{t_1 \longrightarrow t_1'}{\text{let } x = t_1 \text{ in } t_2 \longrightarrow \text{let } x = t_1' \text{ in } t_2} \text{ [E-APP1]}$$

// it will make sure that $t_1 \rightarrow t_1'$ value.

Step 2: let $x = v_1$ in $t_2 \longrightarrow [x \mapsto v_1]\, t_2$. [E-APPABS]

Typing rule:

~~$\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 :$~~

x will not be in typing context so we extend the $\Gamma$

extend $\Gamma$ with $x$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2.} \; [T\_LET]$$

→ Example for typing.

① let y = true in $(\lambda x : Bool . x)\, y$

→ $(\lambda x : Bool . x)$ true

→ true ↛

~~E_AP~~

let y = true in $(\lambda x : Bool . x)\, y \longrightarrow$ ~~let y~~ $= (y \mapsto \text{true})\,(\lambda x : Bool . x)$

$$\frac{\dfrac{\dfrac{}{\vdash \text{true} : Bool}\, T\_TRUE \qquad \dfrac{\dfrac{}{y : Bool \vdash \lambda x : Bool . x \to Bool \to Bool}\, T\_APP \qquad \dfrac{y : Bool \in y : Bool}{y : Bool \vdash y : Bool}\, T\_}{y : Bool \vdash (\lambda x : Bool . x)\, y :}}{\vdash \text{let } y = \text{true in } (\lambda x : Bool . x)\, y : Bool.}\, T\_LET.$$

→ let $x = t_1$ in $t_2$ as an application? Yes.

$(\lambda x : T_1 . t_2)\, t_1$ //

$x\, t_2.$
$t_2\, x$

→ let is a derived form.