Programming Language.

Exams come all from Lecture.

What is PL ?

- A language for instructing a computer to perform computations.
- also used to express algorithms to other people.

What are necessary properties of a PL ?

- unambiguous
- implementable
- Turing Complete. (Express any computable functions) sufficiently powerful to

What are desirable properties of PLs ?

- High level (?)
- system independent (Not tied to any particular O.S or hardware)
- concise.
- Readable.
- Strong Mathematical fundation (programe verification)
- Reusability
- Efficiency (?)
- Exception handling
- Rich set of operators

"High level" vs "Low level" PLs

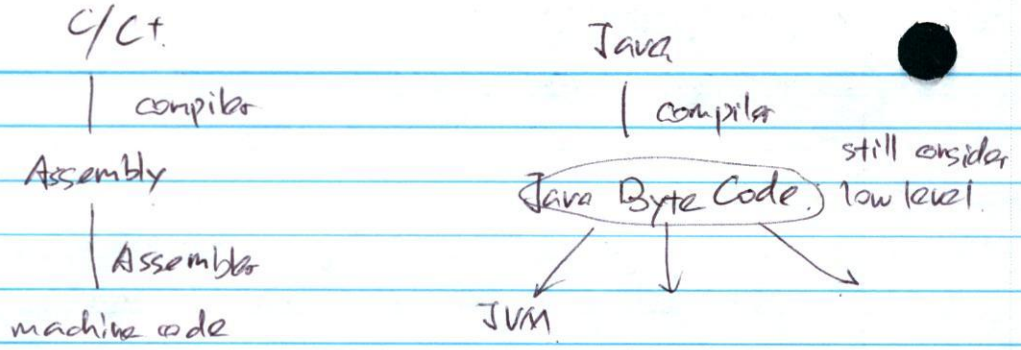
Low level PL - constructs of the language reflect the computer. hardware
(Assembly Language Programming)

- writing to memory locations
- testing bits and jumping
- perform arithmetic instructions

assembly code

```
move r1, $x
add r1, r2
jump too
```

machine code (binary code) 00101101
10010001



High level PL - not low level
- reflects a computational model other than the computer hardware
- mathematical functions
- logic languages - 1st order logic.
- set theory
⋮

About mathematical functions:

C = Imperative language
- variables are just memory locations
- assignment statements write data to memory locations
- loop + jumps for directing execution over assignment statement

```
int p=1
for (int i=1 ; i<=n; i++)
    p = p * i ;
```
} factorial.

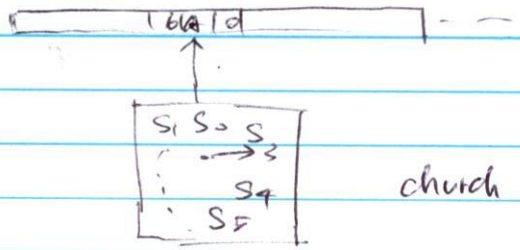
ML = functional language (declared)
- variables are not memory locations
- no assignment statements that overwrite variables
- no loops, only recursion

→ there's no loop

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise.} \end{cases}$$

```
fun fac 0 = 1
|   fac n = n * fac(n-1)
```

Turing Machine.



church thesis ?

A language is **Turing Complete** if it is equivalent in power to a **universal turing machine**.
↳ UTM is a Turing Machine that can emulate any othe TM.

How to define a PL.

- **Syntax** – rules governing the organization of symbols (letters, space, digits, etc) in a valid program.
  – defined by grammars
- **Semantics** – give meaning to valid sequence of symbols
  static / dynamic

Grammars for defining syntax

Regular Expression:
- used for defining the words (names, keywords, numberic literals) that can used

CFG – context free grammar
- used to define how words are composed to form expressions, expressions are composed to forms statements and soon.

1.26 **Recitation**

phase of compiler.
- lexer (Token, scanner ···)
- Parser (Take all lexer together and build a structure base on it)

→

Regular Expression.

ab*.

	concatenation	$R_1 R_2$	ab
R_1 is a RE	alternation	$R_1 \mid R_2$	a|b
R_2 is a RE.	Kleene star.	R_1^*	ϵ, a, aa, aaa

這裡a.b和前面不一樣

language "A set of. valid strings"

a(b|c)a

a b*.

a(b|c)*.

integer — digit digit*.

number — integer | real.

real — digit*. digit digit* in case of different Language.

[A-Z] [a-z]

String only contain 'a' and 'B'

~~ab~~ b(a|b)* b|b ?

Only ~~a~~ >b : a*ba* ba*.

drawback: nesting, palidrome.

CFG: Set of. Non-terminal symbols (do not appear in the final)

set of. terminals (do appear in the final)

start. symbol

productions.

RE ⇏ CFG

CFG ⇏ RE

eg: A → 0A | ε. recursive definition.

A → 0A. — 00A11

Use CFG to represent: strings contain only 0's and 1's

A → ε | 0A1A | 1A0A.

lecture. Syntax (continued)

Semantics

- static semantics : rules governing the use of. types and declarations of names
- Dynamic semantic: defines the execution behavior of. each valid. construct (what is +/loop mean?)

Compiler - translates from one PL to another.

↑ does not execute.

Language translate from: source (java, /c) human written
to: target (Assembly, Machinecode, byte code)

Interpreter - executes the program
output is the result of running the program.

Phases of Compilation

lexical analysis ("lexer")
- forms words (tokens) from sequences of characters

for (⇒ "for" (keyword)
xyz ⇒ "xyz" (Identifier)
1.629 + ⇒ 1.629 (Num)

Parsing (aka 'syntactic analysis'. ("parser")
- forms sentences from sequences of tokens
- expressions statements declarations, functions, modules, programs

STS Type checking - ensures correct use of types and correct declaration and use of names

DS Code generation - (done by code generator)
- produces the target code.

DS Optimization - improves the generated code.

Grammars used to define the syntax of PL's

① Regular Expressions
- used to define the valid words in the PL

R.E ↳	Notation	Set
	a.	{a}
	R_1R_2 where R_1 & R_2 are REs	$\{s_1s_2 \mid s_1$ is a string from R_1
	$R_1 \mid R_2$ → $R_1 \cup R_2$	s_2 is a st ... $R_2\}$
	R^*. (0 or more concatenations) →	the set resulting from zero or more concatenations of strings from R

Alphabet {a, b}
Regular Operator: |, *, (,), concatation

eg
a	{a}
ab	{ab}
a|b|b	{ab, bb}
(a|b)*	{ε, a, b, ab, aa, bb, ba, aab, ... }

for | while | let | do
(0|1|2|3|4|5|6|7|8|9)
Shortcut [0-9]
[0-9][0-9]*
wrong [0-9]*. [0-9]* : ε. ε
[0-9][0-9]*. [0-9]*[0-9] you can have 0.0

② Context Free Grammar.
consists of:
Non-Terminal symbols
- one of which is designated the start non-terminal
Terminal symbols
Set of rules of the form
H → S
H: non-terminal symbol; S: string of symbols (Ter/or NonTerm) and

S → aSa Non-Terminal: S, B
S → B Terminal: a, b
B → bBb
B → ε ← empty string

A CFG defines the set of all strings of ~~non~~-terminals that can be derived from the start non-terminal
- Repeatly applying the rules in the CFG, replacing a non-terminal in a string by the right hand side of a corresponding rule.

derivation:

$S \Rightarrow asa \Rightarrow aaSaa \Rightarrow aaBaa \Rightarrow aabBbaa \Rightarrow aabbaa$

$a^N B^M a^N$

CFG defines arithmetic.

Exp → Exp + Exp

Exp → Exp * Exp

Exp → ID (all possible identifier and numbers)

Exp → NUM

- also written as

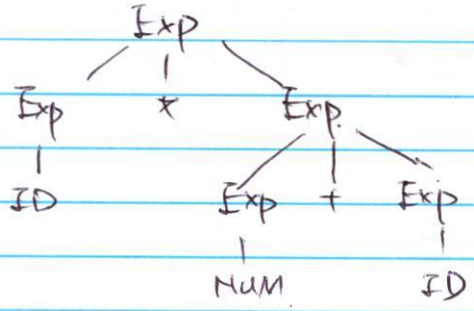
Exp → Exp + Exp | Exp * Exp | ID | NUM

Derivation:

Exp ⇒ Exp * Exp ⇒ ID * Exp ⇒ ID * Exp + Exp ⇒ ID * NUM + Exp

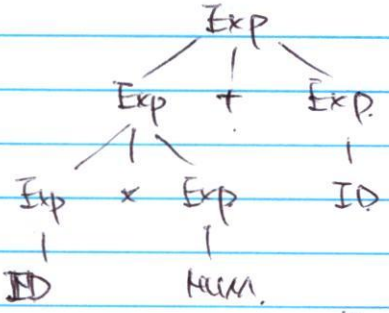
⇒ ID * NUM + ID ⇒ x * 3 + y

Parse Tree – graphical representation of a derivation.



Ambiguous Grammar

- allows two parser trees for the same string



How to fix

① Rewrite grammar → Exp → Term + Term | Term.

② Setting up rules → Term → Factor * Factor | Factor

Factor → ID | NUM | (Exp)

$a^N b^M c^N$ No way to write in CFG

Scoping of a name - portion of the program in which the name is visible.

Block - syntatic construct that defines a scope.

```
{   int x
}
```

```
void f. (int y) {

}
```

Block Structure Language -

a PL in which blocks can be nested

- including nesting functions

In a block structure language, a variable reference is resolved to the corresponding declaration in the innermost surrounding block.

```
x : integer := 0;
procedure f();
begin
    x := x+1;
end
procedure g()
begin
    x: integer := 7;
begin
    f();
end.
```

Which x? Depends on language is in what kind of scoping.

static scoping

Dynamic scoping

Recitation:

Static / Dynamic Semantic Definition 考试会考背下来.

eg3 end in 'a' 的有 set of {a, b}

S → as | bs | a

eg4 even a's b set of {a, b}

S → asas | ε | bS

eg5 Give CFG

```
read x
read y
prod = x * y
write prod
```

program → start-list

start-list → state start-list

state → read id | write Ex | id = Expr

Ep → id

x * (- >) + 4

↑

2/8 lecture.

static scoping: The body of a function is evaluated in the environment in which the function is defined

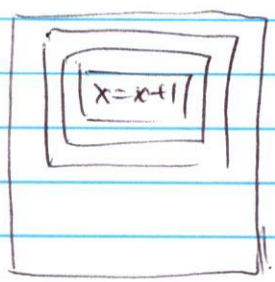
- environment: mapping of names to values or memory locations.

Dynamic scoping: ... called

* local declaration always precedes *

static scoping

compiler figures it out



compile time.

dynamic scoping.

call chain

f g h h k

← looking backwards

run time

dynamic scop
g → f → I

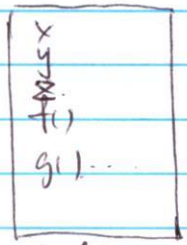
```
x : integer = 7.
procedure f. ( procedure h( ) )
begin
    h( );
end
procedure g( ).
    x: integer := 20;
    procedure I( ).
    begin.
        x := x+1;
    end.
begin
    f(I)
end
```

static scoping

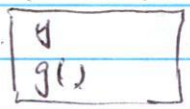
Q
If call g( )
starts.

Ada:
- modularity ("packages")
- concurrency

Module – a collection of code
– has a restricted interface to the rest of the program to code outside of the module.



module body } Ada: "package body"



interface

Ada: "package specification"

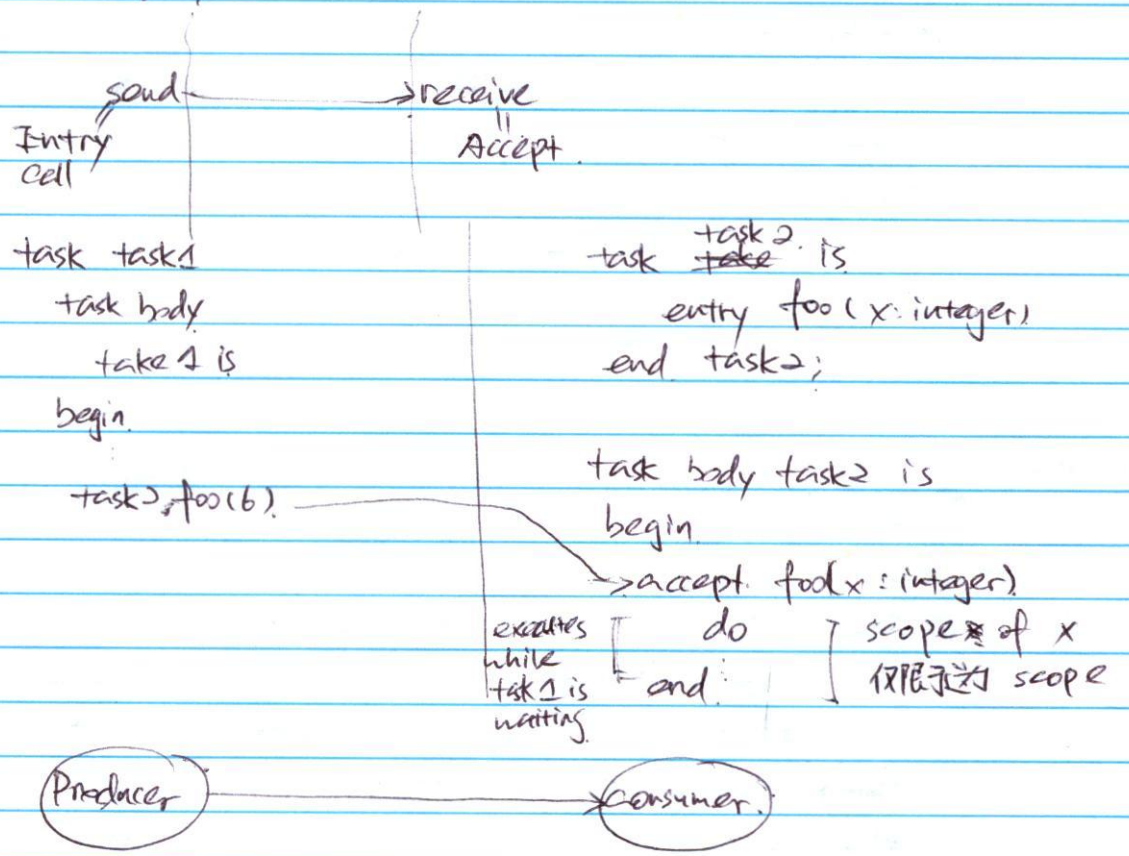
Concurrency – the order in which events may happen is unknown
– no assumption can be made about the order of events

Ada uses thread-based concurrency

- a thread is a sequence of operations (instructions).
- two threads are concurrent if no assumptions can be made about the order of operations in one thread relative to the other thread.

Ada: A thread is called a TASK

like Msg synchronize in OS



Task Type

```
task type MyTask is
    entry Go;
end
task body MyTask is
    begin
        :
    end
```

```
T1: MyTask
T2: MyTask
A: array(1..∞) of MyTask
T1.Go
T2.Go
for i in 1..∞ loop
    A(i).Go;
end loop;
```

Parameter Passing

Math $f(x) = x^2 + 2$
let $y = 10$.
$f(y) = ? = 100 + 2 = 102$

} don't need to think about how y is "passed" to x

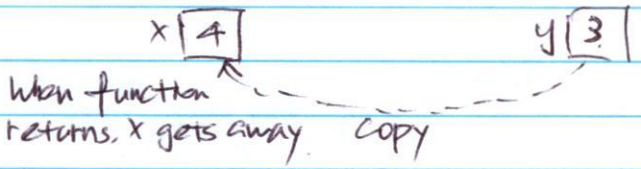
In imperative languages, variables are memory locations.

For parameter passing mechanisms:

① Pass by value
- the value of the actual parameter is copied to the ~~formal~~ ~~forward~~ parameter.

```
void f (int x)
{
  x = x+1;
}
```

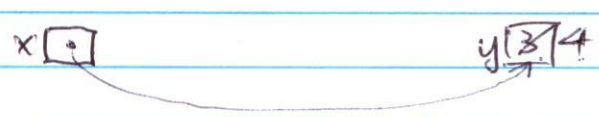
```
int y = 3
f(y)   <- actual parameter.
print(y)
```



② Pass by reference
- the address of the actual parameter is passed into the formal parameter.
- any reference to the formal parameter follows the address back to the actual parameter

```
void f (int x)
{
  x = x+1;
}
```

```
int y = 3
f(y)
print(y)
```



Quiz question.

```
class A {int x}
  A a = new A();
    a.x = 6;
    g(a);
    print(a.x); // 100
```

```
void g(A b)
{  b.x = 100;
}
```

☆ Java is pass by value ☆

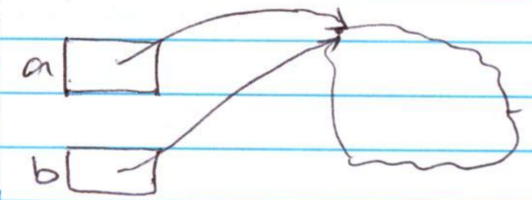
Object has pointer sermantics

---connecting to Array---

Java's objects have "pointer sermantics"

— the "value" of an object is its address

A a = new A();



A b = a

~~开始指向a所指后now create~~ If Jave is

新图 1&2是 pass by value 3是 pass by reference 先 b point to a 再 new A()

③ Pass by value result.

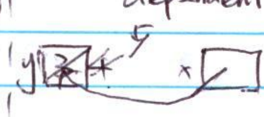
"copy in, copy out"

- upon procedure call, copy the values of the actual params to the corresponding formal parameters
- upon procedure return copy the values of the formal parameters back to the actual params

结果和 pass by reference 一样了?

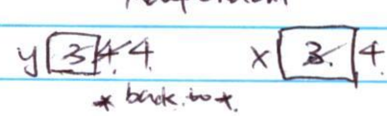
```
int y
void f (int x)
{
    y = y+1
    x = x+1
}
y=3
f(y)
print(y);
```

pass by reference — dependent.



print 5.

pass by result. — independent



print 4.

④ Pass by name (Not that efficient)

- The semantics of a function call is defined by textual substitution.
- "Algo copy Rule"

Step1: Replace a function call with the body of the function definition, renaming variables at the call site to avoid name conflicts.

Step2: Replace all occurences of the formal parameters in the function body with the actual parameters - textual replacement

- rename local variables in the function body to avoid name conflicts.

```
procedure f(x: integer, y: integer);
  begin
    y := x * w;
    y := y + 1;
  end;

z: integer := 15;
w: integer;
begin
    z := z + 1;
    f(z+3, w);
    print(w);
end;
```

step1: `y := x * 2` ; `y := y + 1`

step2: `w := (z+3) * 2` ; `w := w + 1`

⇒ 39

Prediction: ① (in Integer) [in] recommended to add

Ada: By default, (Integer) is taken as (in Integer.

② A := Procedure; X differ from other languages:
A := Function; Don't need ( )

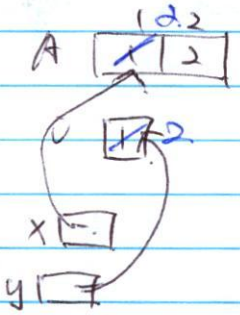
③ Not that Case Sensitive.

④ .ads – specification
.adb – body

3/22.

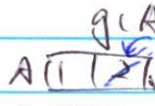
퀴즈 Q what's the difference between pass by reference and pass by name?

답. pass by reference 2
2



Pass by name. 1

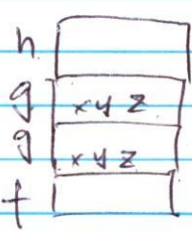
g(A(i), i) → i = i + 1; 4



A(i) = A(i) * 2

Allocating function calls on a stack

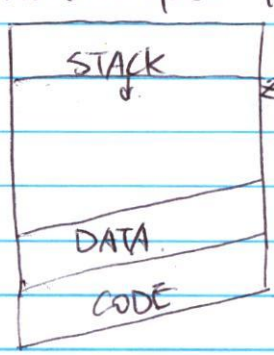
f ~ g ~ g ~ h



"stack frame"
aka "activation record"

Memory Organization on the x86

Address Space for a program



Why heap in between stack & Data ∵ S&D could grow.

return to Address points to Code
Dynamic Link points to stack

To access local variable in stack through FP. offset

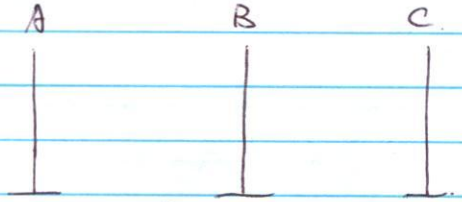
Functional Languages
- Variables denote value, not memory locations
- Recursion must be used instead of loops for repetition
- Functions are values
  "first class objects"

- functions that operate on other functions are called "higher order functions"

Recursive Thinking
① Solve the base case.
  - i.e. for the smallest possible problem size
② Assume that the function you are writing works correctly on size N-1.
③ Under assumption ②, construct the solution for size N

Example: Factorial
① Base case Fac(0) = 1
② Assumption Fac(n-1) = (n-1)!
③ Fac(n) = Fac(n-1) * n

Towers of Hanoi



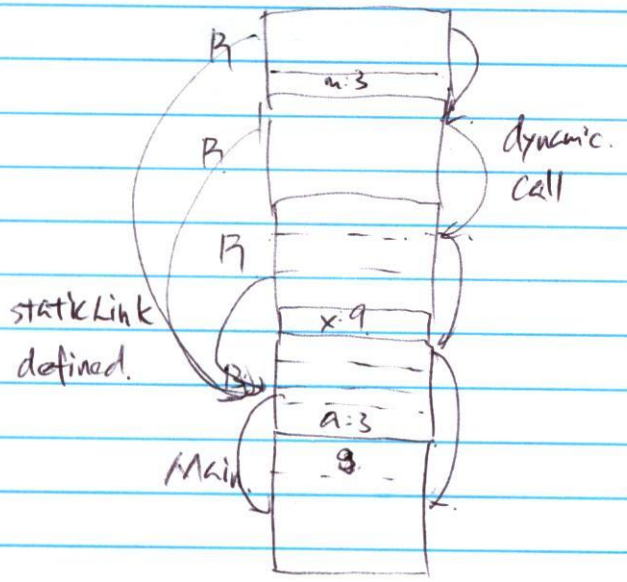
Move N disks from A to B
① Base case N=1
  - Move the disk from A to B
② Assumption:
  Can move N-1 disks from one peg to another, using the third peg as temporary
  move(N-1, FROM, TO, TEMP)
③ Define move(N, FROM, TO, TEMP) as

move (N-1, FROM, TEMP, TO)
move (disk from FROM to TO
move (N-1, TEMP, TO, FROM)

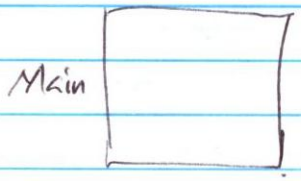
March 2nd - Midterm. (1 hour). How you learned in the class.
- LISP is dynamic scoping but scheme ~~skim~~ fix it.
Scheme - a dialect of Lisp.
- statically scoped
- dynamically typed

why call car & cdr?
'(1 2 3)
content of Address Register.
content of Decoder Register
NIL
L → [| | → [| | → [| | → NIL
1 2 3

(null? L) → testing for the empty list



Example



eg. length of list

① base case empty list. 0.

② Assumption. n-1 length has been found.

③ (+ 1 Len(cdr L))

```
(define len L
    (cond ((null? L) 0)
          (else (+ 1 (len (cdr L))))
    ))
```

3/8.

TM Complet.

Syntax Semantic

Scoping

Ada tasks package

The λ-calculus

- syntactic model of computation
- based on transformations applied to expressions

3/22

Syntax of the λ-calculus

E → C | x | E1 E2 | λx.E

Terminology

"Free" occurrence of a variable

- think of it as a "non-local" variable

x (x y)

↑free ↑

λx. +x |

↑not free - bound

(x (λx.x))

Rules for "free"-ness

x is free in x, but not in any other variable y.

or. any constant 6

* x is free in E1E2 iff x is free in E1 or in E2.
* x is free in (λy.E) iff x is free in E and y is not the same as x

λy.x

↑free.

Substitution

Notation: E[M/x]

- "E with all free occurrences of x replaced by M.
- example: (x y)[(λz.z)/y]

⇒ (x (λz.z))

Rules for substitution

$x[M/x] = M$

$C[M/x] = C$

$y[M/x] = y$, where y is different from x.

$(E_1\ E_2)[M/x] = (E_1[M/x]\ E_2[M/x])$

* $(\lambda x.E)[M/x] = \lambda x.E$ (by definition)

* $(\lambda y.E)[M/x] =$

$\begin{cases} (\lambda y.E[M/x]) & \text{if } y \text{ does not appear free in } M \text{ (no conflict)} \\ (\lambda z.E[z/y])[M/x] & \text{otherwise, where } z \text{ is a new variable.} \end{cases}$

now → $(\lambda y.+x\ 1)[y/x]$

$x \Rightarrow (\lambda z.+x\ 1)[y/x] \Rightarrow (\lambda z.+y\ 1)$

Conversion Rules

α conversion 不在y子情況 In Scheme

If y is not free in E, then (lambda (x) (+ x 3))

$(\lambda x.E) \Leftrightarrow (\lambda y.E[y/x]) \Leftrightarrow$ (lambda (y) (+ y 3))

* β conversion * (pass by name)

$(\lambda x.E)\ M \Leftrightarrow E[M/x]$

eg. $(\lambda x.+x\ x)\ (+2\ 3)$

$\overset{\beta}{\Leftrightarrow}$: $.+(+2\ 3)\ (+2\ 3)$

η conversion

$(\lambda x.E\ x) \Leftrightarrow E$

If x is not free in E.

church encoding

False $\lambda a.\lambda b.b$

True $\lambda a.\lambda b.a$

3/29

not free.

$(+\ x.\ ((\lambda x.\ x\ y)\ z))\ [(zz)/x]$

$\Rightarrow (+\ (zz).1\ ((\lambda x.\ x\ y)\ z))$

β- reduction is β conversion in ⇒ direction

$(\lambda x.E)\ M \Rightarrow E[M/x]$

- Numbers + operators can be encoded as expressions in the λ-calculus and use β-reduction to evaluate them.

But, we will take a short cut: δ-reduction - specifies the behavior of all operators.

$+\ 4\ 5 \overset{\delta}{\Rightarrow} 9$

if true $E_1\ E_2 \overset{\delta}{\Rightarrow} E_1$

if false $E_1\ E_2 \overset{\delta}{\Rightarrow} E_2$

Any expression that β- or δ reduction can be applied to is called a reducible expression "redex".

Any expression that cannot be reduced ~~to~~ is in "Normal Form"

b. $(\lambda x.\ x)$

Q: ~~Does~~ Can every expression be reduced to a normal form?

(No) $(\lambda x.\ \overbrace{x\ x}^{E})(\lambda \overbrace{x}^{M}.\ x\ x) \overset{\beta}{\Rightarrow}$

$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \Rightarrow \cdots$

Recursion using the Y combinator

- has a special property: the fixpoint of f.

$Y\ f \Leftrightarrow f\ (Y\ f)$

- for this reason, Y is also called a "fixed point" combinator

The fix point of a function f is the value z such that $f(z) = z$

e.g. the fixpoint of $f(x) = 2 * x$ is 0

$f(0) = 0$

use recursion $yf = f(yf)$

Factorial

$\underbrace{Y(\lambda f.\lambda x.\ if(=x\ 0)\ 1\ (*\ x\ (f(-x\ 1))))}_{FAC}$

FAC 3: $(Y(\lambda f.\lambda x.\cdots))\ 3$

$(\lambda f.\lambda x.\cdots)(Y(\lambda f.\lambda x.\cdots))\ 3$ $\quad yf = f(yf)$

$\Rightarrow_\beta$ $\lambda x.\ if(=x\ 0)\ 1\ (*\ x\ (Y(\lambda f.\lambda x.\cdots)(-x\ 1)))\ 3$

$\Rightarrow_\beta$ $if(=3\ 0)\ 1\ (*\ 3\ (Y(\lambda f.\lambda x.\cdots)(-3\ 1)))$

$\Rightarrow_\delta$ $\cdots \Rightarrow (*\ 3\ \underbrace{(Y(\lambda f.\lambda x.\cdots)\ 2)}_{FAC.})$

Y can be expressed in the λ-calculas as:

$(\lambda h.(\lambda x.h(x\ x))(\lambda x.h(x\ x)))$

$yf = (\lambda h.(\lambda x.h(x\ x))(\lambda x.h(x\ x)))\ f$

$\Rightarrow_\beta (\lambda x.f(x\ x))(\lambda x.f(x\ x))$

$\Rightarrow_\beta f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$

$\Rightarrow_\beta f(\lambda h.(\lambda x.h(x\ x))(\lambda x.h(x\ x))\ f)$ β conversion in the $\leftarrow$ direction

$= f(yf)$

Orders of Evaluation: The order in which redexes are chosen to reduce.

$\underbrace{(\lambda x.+\ x\ x)\overbrace{(+\ 3\ 4)}^{\delta\text{-reduction}}}_{\beta\text{-reduction}}$

Normal Order Reduction:

Always choose the leftmost outmost redex to reduce

- call the function before evaluating the arguments

Applicative Order Reduction:

Always choose the leftmost innermost redex to reduce

- Evaluate arguments before calling the function.

$(\lambda y.3)((\lambda x.xx)(\lambda x.xx))$

Normal: 3 terminates

A: $(\lambda y.3)((\lambda x.xx)(\lambda x.xx))$ NOT terminates

Church-Rosser Theorem 1:

- All terminating reduction sequences for an expression will result in the same normal form.

Church-Rosser Theorem 2:

- If any reduction sequence terminates, then normal order reduction will terminate

4/12 ML cont.

$x+y$ $(op +)(x,y)$ infix op only takes 2 para tuples

fun f (op <) x y = x < y

x: any type, <: type, y: any type

don't have to be the same type

$(\alpha * \beta) \to \gamma$: $\alpha \to \beta \to \gamma$

$<$: x : y : result

(Define its own infix operator)

3 =*= 4

(op =*=)(3, 4)

infix ++;

在定义 infix 时, 不能写 fun (op++) a b = a+b;

✓ fun a++b = a+b;

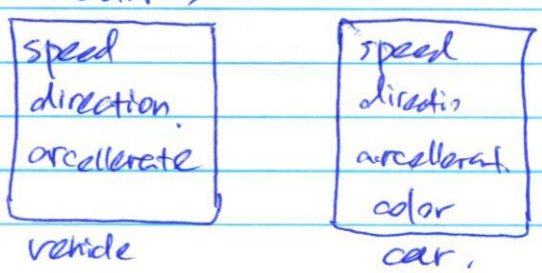
What features make a language "object oriented"?

① Encapsulation of data and code into a single structure.
- fields and methods w/in an object

② Inheritance. child. parent.
- define a new type based on an existing type, where the new type can reuse code defined in the existing type.
- the child type inherits the methods of the parent type as well as the fields

```
class vehicle {
    int speed.
    float direction;
    void accellerate(int delta) {--- }
}

class car extends vehicle {
    int color;
```



③ Subtyping with dynamic dispatch.
"dynamic overloading"
- subtyping: treating a type as if it were another type
  - anywhere a type T can be used, a subtype of T can be used.

"T_2 is a subtype of T_1"

T_1 x = new T_2( );

void f(T1 x) {---}

T2 y = new T2()

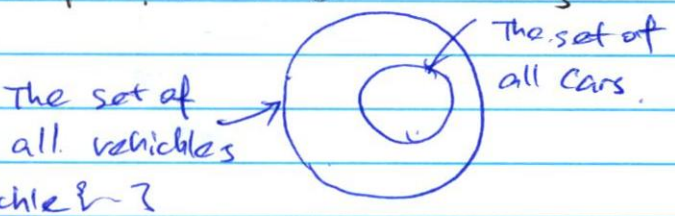
f(y);

- Think of subtyping as an "is a" relation.
  - an object of type T2 "is a" object of type T1.

Dynamic dispatch: determining at run time which method to call, based on the actual type of an object, not the declared type.

A type is a set of elements.
- there are operators defined on these elements

class vehicle { }



class car extends Vehicle {-}

In the subset interpretation of "subtyping" a subtype of a type T denotes a subset of the set denoted by T

what if there was subtyping between function types.
- there is, in Scala

(Using ML syntax, but ML does not have this!)

A fun f (g: A -> int) = g(new A())

|

B fun h (x: B) = x.foo() + 1

B -> int.

A is a B

B is A.

(Q1:) f(h) (not) ok!

You can not use a B -> int where an A -> int is expected.

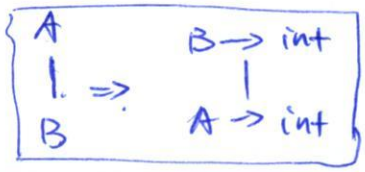
fun f (g: B -> int) = g(new B())

fun h (x: A) = x.foo() + 1

A -> int.

(Q2:) f(h) (is) ok!

You can use an A -> int whenever a B -> int is expected.



Function subtyping is contravariant on the input types

re C
|
car D

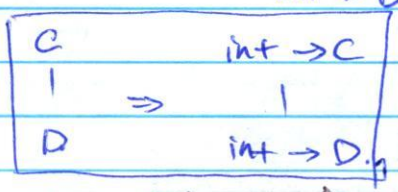
```
fun f (g: int -> C) = let y:C = g(6);
fun h (x: int). = new D();
                  int -> D.
```

f(h) is ok. D is C.

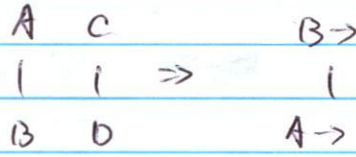
```
fun f (g: int -> D) = let y:D = g(6);
fun h (x: int) = new C();
                 int -> C.
```

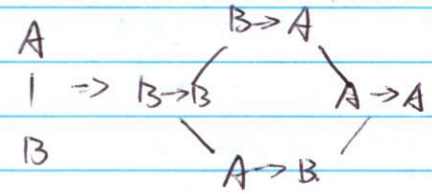
f(h) is not ok. C is not D

Anywhere a int → C can be used.
int → D can be used.



Function subtying is covariant on the output types





4/19

Function Subtyping :

- controvariant on the input type.

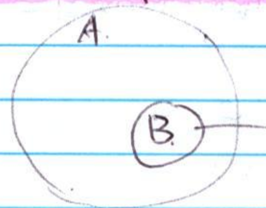
If B is a subtype of A, then A → Int is a subtype of B → int.

- covariant on the output type

If B is a subtype of ~~B~~ A then int → B is a subtype of int → A.

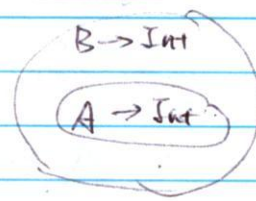
Subset Interpretation of subtyping.

A
|
B



though B might have more in own.

A → B → Int
| → |
B → A → Int

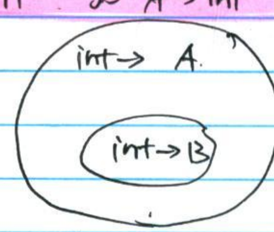


A → int denotes the set of all functions that can be applied to an A object and return an int

B → int denotes the set of all functions that can be applied to a B object and return an ~~a~~ int

Since any function expecting an A can be applied to B any function in A → int is also in B → int. So A → int is a subset of B → int

A → int → A
| → |
B → int → B



int → B denotes the set of all funtions that ~~can retur~~ taking a Int and returning a B.

int → A denotes the set of - - - - - - - - - - - - - - a A.

Since a B object is an A object. any function returning B also returns A. So any function in Int → B is also in Int → A, so Int → B is a subset of Int → A.

parameter type

interface I<T> Entry<T,T> twice(T value);

Java Generic does not implement Subtyping.

Q: Vehicle | ⇒ ? Car

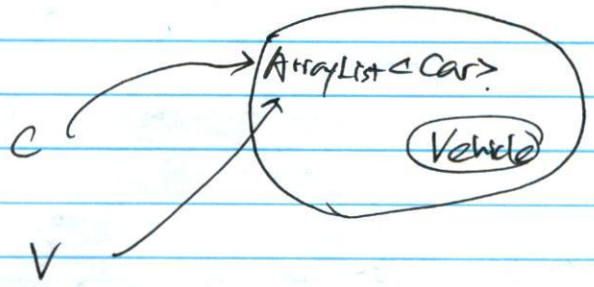
ArrayList<Vehicle> | ArrayList<Car>

NO!!!

Assumption: subtype is allowed.

```
ArrayList<Car> c = new ArrayList<Car>

ArrayList<Vehicle> V = c
V.add(new Vehicle);
Car x = C.get(0);
```




```
void AddVehicle(ArrayList<Vehicle> V){
    V.add(new Vehicle());

ArrayList<Car> c= new ArrayList<Car>()
AddVehicle(c);
```

⇒ So ArrayList<Car> is not a subtype of ArrayList<Vehicle>

Q: Is ArrayList<Vehicle> a subtype of ArrayList<Car>? NO!!!

```
ArrayList<Vehicle> V = new ArrayList<Vehicle>
ArrayList<Car> c = V;
    Car x = C.get();
```

★ No subtyping between instances of A generic ★

Defining a function that operates on different types of ArrayList.

```
void f ( ArrayList<Object> A ) {
    for ( Object e : A )?
        S.O.p (e)
    X A.add (new Object())
}
```

和前面一样=错误.

Solution: 用?

↑这里不行. 假设pass in Car. Object 和 ? 不match.

syntax matches any kind of list.

```
Vehicle
  |
Car
  |
Porshe
```

```
void f( ArrayList<? extends Vehicle> V )
{ for ( Vehicle e : V )
      print (e.speed);
}
```

Can't stick anything into List.

V 不为 C

V 不为 P.

```
void g ( ArrayList<? super Car > C )
{
    C.add (new Car());
}
```

super ↑ upper extends ↓ lower.

In bounded.java.

```
class C<T extends A> {
        T x
        ArrayList<T> f()
}
```

```
class C<T extends A>
    extends D<T>      D is parent.
    implements I<T> {----}
```

Scala. Notes

a+b ↔ a.+(b) a.f(b) ↔ a f b

```
def f(c: Collection[A]) {
    c.add(new A())
}
var cb: Collection[B]
f(cb)   Wrong :  C[A]
                  | x.
                C[B]

def f(c: Collection[B]) {
    c.add(new B())
}
var ca: Collection[A]
f(ca)  then works

def g(c: Collection[A]) {
    val a: A = c.get(0)
}
var cb: Collection[B] = ---
g(cb)
```

```
A     C[B]
|     |   contra case
B     C[A]
```

```
C[A]
|   covar case
C[B]
```

Back to Polymorphism

ML: parametric Polymorphism

OOP: bounded Polymorphism
- subtype
- interface