Exercise: Java programs

```
class C { int i;
          int seti (int j) { i=j; return i; }
          int geti () { return i; } }

class D { int f() { x = new C();
                    x.seti(10);
                    return x.geti(); // ?  10
        }
```

```
class E { int f() { x = new C();
                    y = x;
                    x.seti(10);
                    return y.geti(); // ?  10
        }
```

SOS rules for objects in MiniJava (v. 1)

- **Objects are just maps from variables to values, together with the name of the class:**

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
           | ObjectV object
and varname = string and binding = (varname * value)
and object = id * (binding list) and state = binding list
```

- **SOS rules for new expressions:**

(NEW) `new C()`, σ, $\pi \Downarrow$

Ignore

(THIS) `this`, σ, $\pi \Downarrow$

SOS rules for objects in MiniJava (v. 1), cont.

(VAR) $x, \sigma, \pi \Downarrow v$

(FIELD) $x, \sigma, \pi \Downarrow v$

Ignore

(VARASGN) $x = e, \sigma, \pi \Rightarrow$

(FIELDASGN) $x = e, \sigma, \pi \Rightarrow$

SOS rules for objects in MiniJava (v. 1), cont.

(METHOD-CALL)

$e_0 . f(e_1, \ldots, e_n), \sigma, \pi \Rightarrow$

Ignore

Which Java examples work?

SOS rules (v. 2) (cont.)

```
type stackvalue = IntV of int | StringV of string
                | BoolV of bool | NullV | Location of location
and location = int
type environment = (varname * stackvalue) list
type heapvalue = Object of classname * environment
type store = heapvalue list
type state = environment * store
```

- **Write an expression of type `state` for a state that contains variables `x` bound to 3 and `y` bound to an object of class `C`; `C` contains fields `a` and `b`, and in `y` these have integer values 4 and 5.**

```
( [ ("x", IntV 3); ("y", Location 0)],
  [ Object ("C", [("a", IntV 4); ("b", IntV 5)])] )
```

SOS rules (v. 2) (cont.)

- **Give the SOS rule for** `new`**:**

  See MP6

  (NEW**)** `new C()`, (ρ, η), $\pi \Downarrow$

- **New form of SOS rules reflected in new type of** `eval`**:**

```
let rec eval (e:exp) ((env,sto) as sigma:state) (prog:program)
        : stackvalue * store =
```

- **and the corresponding clause in** `eval` **(you can assume any auxiliary functions you think useful):**

```
    | NewId c ->
```

See MP6 solution

SOS rules (v. 2) (cont.)

(NOT) $!e, (\rho, \eta), \pi \Downarrow$

See MP 6

(INT-MULT) $e_1 * e_2, (\rho, \eta), \pi \Downarrow$

SOS rules (v. 2) (cont.)

(VAR) $x, (\rho, \eta), \pi \Downarrow$

See MP 6

(FIELD) $x, (\rho, \eta), \pi \Downarrow$

(METHOD-CALL)

$e_0 . f(e_1, \ldots, e_n), (\rho, \eta), \sigma, \pi \Downarrow v$

Some SOS rules from MP 5

(Plus-Int)

$e_1 + e_2,\ \sigma,\ \pi \Downarrow \text{IntV}\ (i_1 + i_2)$
$\quad e_1,\ \sigma,\ \pi \Downarrow \text{IntV}\ i_1$
$\quad e_2,\ \sigma,\ \pi \Downarrow \text{IntV}\ i_2$

(Var)

$x,\ \sigma,\ \pi \Downarrow \sigma(x) \quad \text{if } \sigma(x) \neq \perp$

(Method-Call)

$x.f(e_1, \ldots, e_n),\ \sigma,\ \pi \Downarrow v$
$\quad e_1,\ \sigma,\ \pi \Downarrow v_1$
$\quad \vdots$
$\quad e_1,\ \sigma,\ \pi \Downarrow v_1$
$\quad sl,\ \sigma_0,\ \pi \Rightarrow \sigma_0'$
$\quad e,\ \sigma_0',\ \pi \Downarrow v$

where $\sigma_0 = [(x_1,v_1);\ldots;(y_0,\ \text{NullV});\ldots]$
and f is defined in π as
Method(t, f, $[x_1;\ldots]$, $[y_1;\ldots]$, sl, e)

(Mult-Int)

$e_1 * e_2,\ \sigma,\ \pi \Downarrow \text{IntV}\ (i_1 * i_2)$
$\quad e_1,\ \sigma,\ \pi \Downarrow \text{IntV}\ i_1$
$\quad e_2,\ \sigma,\ \pi \Downarrow \text{IntV}\ i_2$

(Int-Const)

$i,\ \sigma,\ \pi \Downarrow \text{IntV}\ i$

(Stmt-List)

$s_1; \ldots; s_n,\ \sigma_1,\ \pi \Rightarrow \sigma_{n+1}$
$\quad s_1,\ \sigma_1,\ \pi \Rightarrow \sigma_2$
$\quad \vdots$
$\quad s_n,\ \sigma_n,\ \pi \Rightarrow \sigma_{n+1}$

(Stmt-List-0)

$[],\ \sigma_1,\ \pi \Rightarrow \sigma_1$

(Assignment)

x=$e,\ \sigma,\ \pi \Rightarrow \sigma[v/x], \quad \text{if } \sigma(x) \neq \perp$
$\quad e,\ \sigma,\ \pi \Downarrow v$

Proofs in SOS

- **Use SOS rules to obtain results of entire computations.**

3+(4*5), σ, π $\Downarrow$ **IntV 23**	(PLUS-INT)
3, σ, π $\Downarrow$ **IntV 3**	(INT-CONST)
4*5, σ, π $\Downarrow$ **IntV 20**	(MULT-INT)
4, σ, π $\Downarrow$ **IntV 4**	(INT-CONST)
5, σ, π $\Downarrow$ **IntV 5**	(INT-CONST)

- **Exercise:**

(3*4) + (5*2), σ, π $\Downarrow$ **IntV 22**	(INT-PLUS)
3 * 4, σ, π $\Downarrow$ 12	(Mult-Int)
3, σ, π $\Downarrow$ 3	(Int-Const)
4, σ, π $\Downarrow$ 4	(Int-Const)
5 * 2, σ, π $\Downarrow$ 10	(Mult-Int)
5, σ, π $\Downarrow$ 5	(Int-Const)
2, σ, π $\Downarrow$ 2	(Int-Const)

Proofs in SOS (cont.)

- **Suppose π contains class Main with method:** `int f (int a) { return a; }`.

x=5;y=null.f(x), σ_0, $\pi \Rightarrow \sigma_2$	(StmtList)
x=5, σ_0, $\pi \Rightarrow \sigma_1$	(Assignment)
5, σ_0, $\pi \Downarrow$ **IntV 5**	(Int-Const)
y=null.f(x), σ_1, $\pi \Rightarrow \sigma_2$	(Assignment)
null.f(x), σ_1, $\pi \Downarrow$ **IntV 5**	(Method-Call)
x, σ_1, $\pi \Downarrow$ **IntV 5**	(Var)
[], σ_3, $\pi \Rightarrow \sigma_3$	(StmtSeq-0)
a, σ_3, $\pi \Downarrow$ **IntV 5**	(Var)

where σ_0 **= [(x,IntV 0);(y,IntV 0)],** σ_1 **= [(x,IntV 5);(y,IntV 0)],** σ_2 **= [(x,IntV 5);(y,IntV 5)], and** σ_3 **= [(a,IntV 5)].**

Proofs in SOS (cont.)

- **Exercise: Same statements, but** `int f (int x) return a+1;`

x=5;y=null.f(x), σ_0, $\pi \Rightarrow \sigma_5$ **(**σ_5 **= [(x,IntV 5);(y,IntV 6)])**

(Stmt list)

$x=5, \sigma_0, \pi \Rightarrow \sigma_1 \quad (\sigma_1 = [(x, \text{IntV } 5)])$ (Asgn)

$5, \sigma_0, \pi \Downarrow 5$ (Int Const)

$y = \text{null}.f(x), \sigma_1, \pi \Rightarrow \sigma_5$ (Asgn)

$\text{null}.f(x), \sigma_1, \pi \Downarrow 6$ (Method call)

$x, \sigma_1, \pi \Downarrow 5$ (Var)

$[\,], \sigma_2, \pi \Rightarrow \sigma_2 \quad (\sigma_2 = [(a, 5)])$ (Stmt list 0)

$a+1, \sigma_2, \pi \Downarrow 6$ (Int Plus)

$a, \sigma_2, \pi \Downarrow 5$ (Var)

$1, \sigma_2, \pi \Downarrow 1$ (Int Const)

SOS rules (v. 2) (cont.)

(NOT) $!e, (\rho, \eta), \pi \Downarrow$

See MP6

(INT-MULT) $e_1 * e_2, (\rho, \eta), \pi \Downarrow$

SOS rules (v. 2) (cont.)

(VAR) $x, (\rho, \eta), \pi \Downarrow$

See MP6

(FIELD) $x, (\rho, \eta), \pi \Downarrow$

(METHOD-CALL)

$e_0 . f(e_1, \ldots, e_n), (\rho, \eta), \sigma, \pi \Downarrow v$

SOS rules (v. 2) (cont.)

- **Rules for statements actually don't change — they always passed the state along from one to the next — except for assignment.**

(VarAsgn) $x{=}e$, (ρ, η), $\pi \Rightarrow$

See MP6

(FieldAsgn) $x{=}e$, (ρ, η), $\pi \Rightarrow$

Inheritance in Java

```
// EXAMPLE 1
class B {                                class C extends B {
    string f() { return this.g(); }          string g() { return "C"; } }
    string g() { return "B"; } }

x = new B(); y = new C();
x.f(); // ?  "B"
y.f(); // ?  "C"

// EXAMPLE 2
class B { B aB;                          class C extends B {
    void r() { aB = this; }                  string g() { return "C"; } }
    string s() { return aB.g(); }
    string g() { return "B"; } }

x = new B(); y = new C(); x.r();  y.r();
x.s(); // ?  "B"
y.s(); // ?  "C"
```

Inheritance in Java (cont.)

```
// EXAMPLE 3
class B {
    B aB;
    void q(B x) { aB = x; }
    string s() { return aB.g(); }
    string g() { return "B"; } }
```

```
class C extends B {
    string g() { return "C"; } }
```

```
x = new B(); y = new C();
x.q(x); x.s(); // ? "B"
x.q(y); x.s(); // ? "C"
y.q(y); y.s(); // ? "C"
y.q(x); y.s(); // ? "B"
```

Inheritance in Java (cont.)

```
// EXAMPLE 4
class B {
    string f() { return this.g(); }  }
    string g() { return "B"; } }
```

```
class C extends B {
    B b;
    string g() { return "C"; }
    string f() { return b.g(); }
    void h(B y) { b = y; } }
```

```
x = new B(); y = new C();
y.h(y); y.f(); // ?
y.h(x); y.f(); // ?
```

"C"

"B"

Type-checking rules for MJ

- **Will give several kinds of judgments.** Γ **is a** *type environment*, **giving the types of variables.**

$\vdash \pi$	π **is a type-correct program**
$\pi \vdash \kappa$	κ **is a type-correct class in program** π
$\pi,\ \Gamma \vdash \mu$	μ **a type-correct method in a class whose fields are given in** Γ**.**
$\pi,\ \Gamma \vdash S$	S **is a type-correct statement, when** Γ **gives all variable declarations surrounding** S **(fields, parameters, locals)**
$\pi,\ \Gamma \vdash e : \tau$	e **is a type-correct expression of type** τ**.**

- **For now, ignore subclassing (i.e. inheritance)...**

Type-checking rules for programs and classes

$\vdash \pi$ where $\pi = cl_1 \ \ldots \ cl_n$

$\pi \vdash cl_1$

$\vdots$

$\pi \vdash cl_n$

(and $cl_1 \ldots cl_n$ have different names)

$\pi \vdash \texttt{class}\ c\ \{\ fld_1\ \ldots\ fld_k\ \mu_1\ \ldots\ \mu_m\ \}$

$\pi, \Gamma_c \vdash \mu_1$, where $\Gamma_c = \{fld_1, \ldots, fld_k\}$

$\vdots$

$\pi, \Gamma_c \vdash \mu_m$

(and all fields have distinct names and all methods have distinct names (ignoring overloading))

Type-checking rules for methods

$\pi, \Gamma \vdash \tau \; f \; \texttt{(args)} \; \{ \; S_1 \; \ldots \; S_n \; \texttt{return} \; e \; \}$

$\pi, \Gamma_{args} \vdash S_1$

$\vdots$

$\pi, \Gamma_{args} \vdash S_n$

$\pi, \Gamma_{args} \vdash e : \tau'$

where $\Gamma_{args} = \Gamma \cup \{ \text{args} \}$

and $\tau' = \tau$ (no subclasses)

or $\tau' < \tau$ (subclasses)

Type-checking rules for statements

$\pi, \Gamma \vdash$ `if` (e) S_1 `else` S_2

$\pi, \Gamma \vdash e : \text{bool}$

$\pi, \Gamma \vdash S_1$

$\pi, \Gamma \vdash S_2$

$\pi, \Gamma \vdash x$ `=` e

$\pi, \Gamma \vdash e : \tau$

where $\tau = \Gamma(x)$ (no subclassing)

or $\tau < \Gamma(x)$ (subclassing)

Type-checking rules for expressions

$\pi, \Gamma \vdash \mathbf{x} : \pi(x)$

$\pi, \Gamma \vdash \mathbf{not}\ e : \text{bool}$

$\pi, \Gamma \vdash e : \text{bool}$

$\pi, \Gamma \vdash e_1 * e_2 : \text{int}$

$\pi, \Gamma \vdash e_1 : \text{int}$

$\pi, \Gamma \vdash e_2 : \text{int}$

$\pi, \Gamma \vdash e_1 + e_2 : \text{int}$

$\pi, \Gamma \vdash e_1 : \text{int}$

$\pi, \Gamma \vdash e_2 : \text{int}$

Type-checking rules for expressions

$\pi, \Gamma \vdash e_1 + e_2 :$ string

$\pi, \Gamma \vdash e_1 :$ string

$\pi, \Gamma \vdash e_2 : \tau$ (any τ)

$\pi, \Gamma \vdash e_1 + e_2 :$ string

$\pi, \Gamma \vdash e_1 : \tau$ (any τ)

$\pi, \Gamma \vdash e_2 :$ string

$\pi, \Gamma \vdash e_0.f(e_1, \ldots, e_n) : \tau$

$\pi, \Gamma \vdash e_0 : C$

$\pi, \Gamma \vdash e_1 : \tau_1$

$\vdots$

$\pi, \Gamma \vdash e_n : \tau_n$

C defines $\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{\ldots\}$

(no subclassing)

or C inherits definition

$\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{\ldots\}$

Type-checking rules, with subclasses

$\pi, \Gamma \vdash x{=}e$

$\pi, \Gamma \vdash \tau \ f \ \texttt{(}\mathbf{\mathit{args}}\texttt{)} \ \{ \ S_1 \ \ldots \ S_n \ \texttt{return} \ e \ \}$

Already given

Type-checking rules, with subclasses (cont.)

$\pi, \Gamma \vdash e_0.f(e_1, \ldots, e_n) : \tau$

Already given

$\pi \vdash$ `class` c `extends` s { $fld_1 \ldots fld_k \ \mu_1 \ldots \mu_m$ }

$\pi, \Gamma_c \vdash \mu_1$

$\vdots$

$\pi, \Gamma_c \vdash \mu_n$

where $\Gamma_c = \{fld_1, \ldots, fld_k\}$ $\cup$ fields inherited from s and s's ancestors

Type-checking in MP7

- **Check types and add locations, constructing new AST of type** `programT`**:**

```
type programT = ProgramT of (class_declT list)

and class_declT = ClassT of id * id * ((var_kind * var_decl) list)
        * (method_declT list) * int (* number of fields *)

and method_declT = MethodT of exp_type * id * (var_decl list)
        * (var_decl list) * (statementT list) * annExpT * int (* size of stack frame *)

and statementT = BlockT of (statementT list)
    | IfT of annExpT * statementT * statementT
    | AssignVarT of id * annExpT * int
    | AssignFieldT of id * annExpT * int

and annExpT = expT * exp_type * int

and expT = OperationT of annExpT * binary_operation * annExpT | IntegerT of int
    | TrueT | FalseT | MethodCallT of annExpT * id * (annExpT list) | ThisT | NewIdT of id
    | VarT of id | FieldRef of int | NewIdAlloc of id * int | NotT of annExpT | NullT
    | StringT of string | CvtIntToStringT of annExpT | CvtBoolToStringT of annExpT
```

Example

- **Frame consists of four integer variables (`x`, `y`, `z`, and `w` (locations 1, 2, 3, 4); objects include field `s`, of type `string`, at offset 5. Write statementT for:**

location of x

```
z = x + 3
```

AssignVarT("z",
Operation((VarT "x", IntType, 1), Plus,
(IntegerT 3, IntType, 5)),
3)

location of z

location of temporary

```
s = s + 2
```

AssignFieldT("s",
Operation((FieldRef 5, IntType, 5), Plus
(IntegerT 2, IntType, 6)),
5)

locations of temporaries

Abstract machine exercises

Suppose the code sequence C has the following instructions at locations 10–12:

```
LOADIMM  6,3
ADD      5,1,6
MOV      3,5
```

If the current frame has values: [3,7,2,4,9,21,13,15], give the state after each instruction:

(10, C, [3,7,2,4,9,21,13,15], h, t, r)

LOADIMM 6,3 (11, C, [3,7,2,4,9,21,3,15], h, t, r)

ADD 5,1,6 (12, C, [3,7,2,4,9,10,3,15], h, t, r)

MOV 3,5 (13, C, [3,7,2,10,9,10,3,15], h, t, r)

Compilation of methods

MethodT(typ,f,args,vars,sl,ret,sz) $\rightsquigarrow$

See MP 7

Compilation of expressions

IntegerT i, loc $\rightsquigarrow$

StringT s, loc $\rightsquigarrow$

See MP7

TrueT, loc $\rightsquigarrow$

VarT id, loc $\rightsquigarrow$

NotT e, loc $\rightsquigarrow$

Compilation of expressions (cont.)

OperationT($e1$,Plus,$e2$), loc $\rightsquigarrow$

See MP7

CvtIntToString e, loc $\rightsquigarrow$

NewIdAlloc(c,sz), loc $\rightsquigarrow$

Compilation of statements

$x{=}e$, m $\rightsquigarrow$

See MP 7

$\{\ S_1, \ldots, S_n\ \}$, m $\rightsquigarrow$

if (e) S_1 else S_2, m $\rightsquigarrow$

Method calls

MethodCallT(e0,f,[e1,...,en])

See MP 7

Compiling while statements

While (e) S, $m \rightsquigarrow$

[JUMP m'] @ ils @ ile @ [CJUMP loc,$m+1$,m''],m''

S, $m+1 \rightsquigarrow ils$, m'

e, loc $\rightsquigarrow ile$

(where $m'' = m' + |ile| + 1$)

do S while (e), $m \rightsquigarrow$ ils @ ile @ [CJUMP loc, m, m'+|ile|+1]

S, m $\rightsquigarrow$ ils, m'

e, m' $\rightsquigarrow$ ile

Short-circuit evaluation of boolean expressions

- **The best way to compile boolean expressions is to avoid computing the value of the expression.**

$$e,\ m,\ t,\ f \rightsquigarrow_2 il,\ m'$$

- **Some expressions are compiled very simply:**

True, m, t, $f \rightsquigarrow_2$ **[JUMP** t**]**, $m+1$

False, m, t, $f \rightsquigarrow_2$ **[JUMP** f**]**, $m+1$

!e, m, t, $f \rightsquigarrow_2 il$, m'
e, m, f, $t \rightsquigarrow_2 il$, m'

Short-circuit evaluation of boolean expressions (cont.)

$e_1 \&\& e_2,\ m,\ t,\ f \rightsquigarrow_2$

$e_1 || e_2,\ m,\ t,\ f \rightsquigarrow_2$

See MP7
extra credit

If (e) S_1 else $S_2,\ m \rightsquigarrow$

Arrays in MJ

- **Arrays stored in the heap. Contents are integers — values or pointers to heap objects (including arrays).**
- **Have instruction (not used in MP7):**

```
ARRAYREF tgt,src,indx: (p, c, s, h, t, r)
                          -> (p+1, c, s[i/tgt], h, t, r)
```

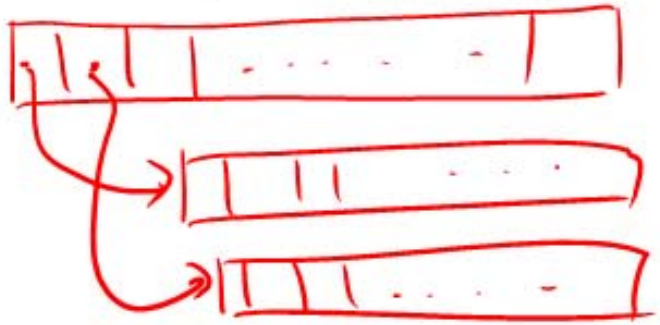
- **Array indexing:**

$a[e]$, **loc** $\rightsquigarrow$ il @ ARRAYREF loc, addr(a), loc1

e, loc1 $\rightsquigarrow$ il

Multi-dimensional arrays in MJ

- **A multi-dimensional array is an array that contains pointers to other arrays.**



- **Array indexing for multi-dimensional arrays:**

$e_1[e_2]$, **loc** $\rightsquigarrow$ il1 @ il2 @ ARRAYREF loc, loc1, loc2

e_1, loc1 $\rightsquigarrow$ il1

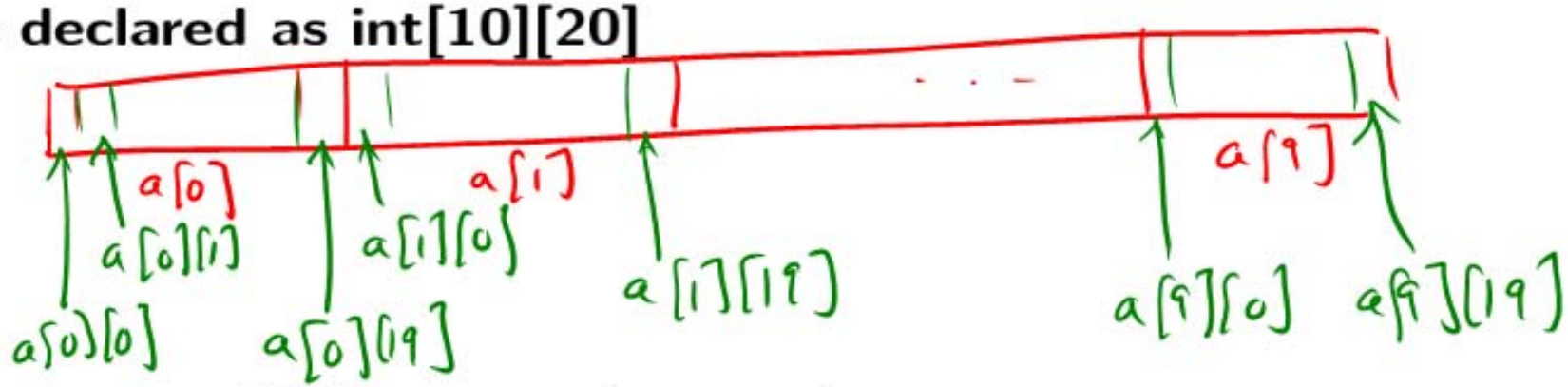
e_2, loc2 $\rightsquigarrow$ il2

Arrays in C

- **Arrays are addresses:** `a[i]` $\equiv$ `a + i` **(where** `i` **is multiplied by the size of a's elements)**

- **Multi-dimensional arrays always rectangular, and arranged in** *row-major order*:

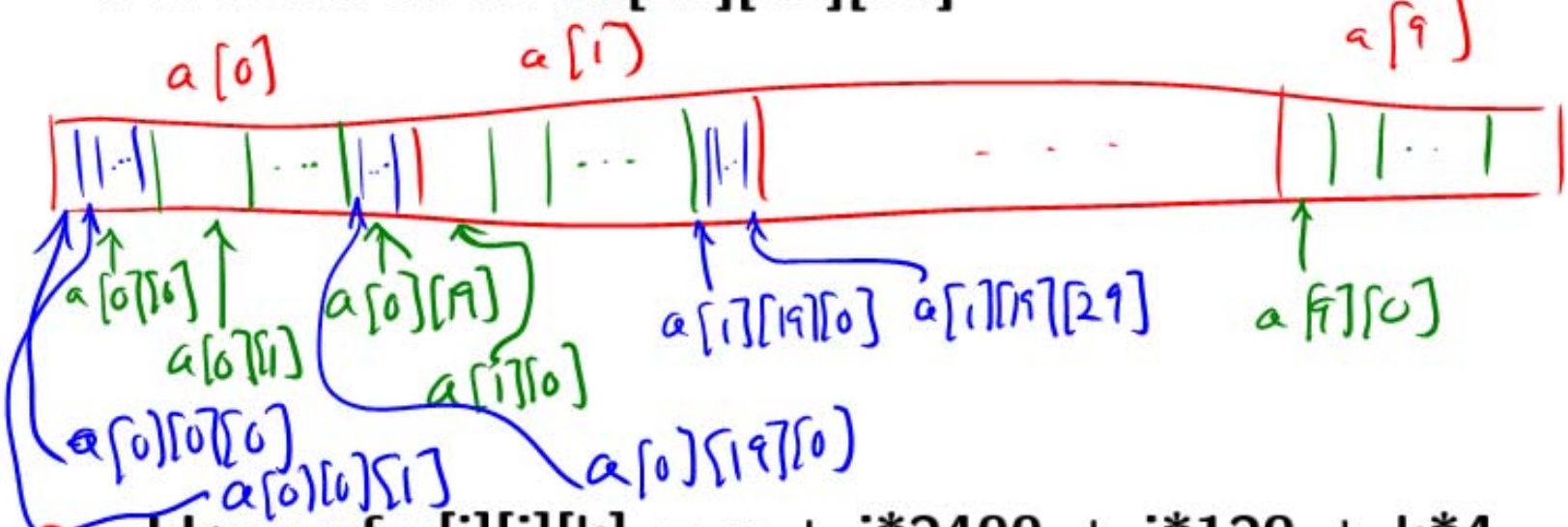
  - **a is declared as int[10][20]**



  - **address of a[i][j] = a + i*80 + j*4.**

Arrays in C (cont.)

- a is declared as int[10][20][30]



- address of a[i][j][k] = a + i*2400 + j*120 + k*4

- Rule is: address of $e_1[e_2]$ = address of e_1 + (e_2 * (size of elements of e_1))

V-tables (cont.)

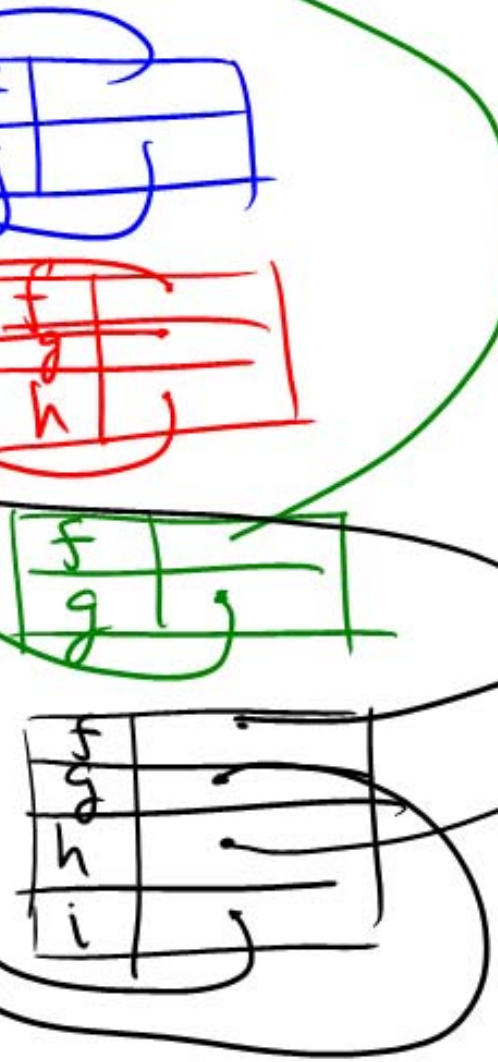
- Draw a table for each class, listing all the methods belonging to that class (including inherited ones). The order should be from top of the hierarchy to the bottom.

```
class B {
   void f() {}
   void g() {}
}

class C1 extends B {
   void h() {}
}

class C2 extends B {
   void g() {}
}

class D extends C1 {
   void i() {}
   void g() {}
}
```



Exercises with anonymous functions

- **Define a list of three functions: the first adds 1 to any integer; the second doubles any integer; and the third triples any integer. Then apply the second function to 4:**

```
let funlis =
 [fun x -> x+1 ; fun x -> x*2 ; fun x -> x*3 ]
in ( hd (tl funlis) ) 4
```

- **Transform the following expression to the form `let rec length = ...` , and remove pattern-matching:**

```
let rec length lis = match lis with [] -> 0 | h::t -> 1 + length t
in length [1;2;3]
```

```
let rec length = fun lis -> if lis=[] then 0
                            else 1 + length (tl lis)
in length [1;2;3]
```

Abstract syntax of MiniOCaml

```
type exp =
     Operation of exp * binary_operation * exp
   | UnaryOperation of unary_operation * exp
   | Var of string | StrConst of string | IntConst of int
   | FloatConst of float | True | False
   | List of exp list | Tuple of exp list
   | If of exp * exp * exp | App of exp * exp
   | Let of string * exp * exp
   | Fun of string * exp
   | Rec of string * exp

and binary_operation = Semicolon | Comma | Equals | LessThan
   | GreaterThan | NotEquals | Assign | And | Or
   | IntPlus | IntMinus | IntDiv | IntMult
   | FloatPlus | FloatMinus | FloatDiv | FloatMult
   | StringAppend | ListAppend

and unary_operation = Not | Head | Tail | Fst | Snd
```

Exercise: define `closed`

```
(* closed e is true iff e has no free variables.
   closed1 e varlist is true iff any free variables in e are in varlist *)
let closed (e:exp) : bool =
   let rec closed1 (e:exp) (vars:string list) : bool =
       match e with
   Operation (e1, bop, e2) -> closed e1 vars && closed e2 vars
   | Var x -> isin x vars
   | Let (y, e1, e2) -> closed e1 vars
                        && closed e2 (y :: vars)

   | Fun (x, e) -> closed e (x :: vars)
       ⋮

  in closed1 e []
```

Rules for evaluation by substitution

`IntConst` $i \Downarrow$

See MP8 or lecture 19

e_1 + $e_n \Downarrow$

`List` [e_1, ..., e_n] $\Downarrow$

Rules for evaluation by substitution (cont.)

$\texttt{If}(e_1,\ e_2,\ e_3) \Downarrow$

See MP8 or lecture 19

$\texttt{Fun}(x,\ e) \Downarrow$

Rules for evaluation by substitution (cont.)

$\texttt{Let}(x,\ e_1,\ e_2) \Downarrow$

See MP8 or lecture 19

$e_1\ e_2 \Downarrow$

Examples of evaluation by substitution

`let x = 3 in x+1` $\Downarrow 4$ (let)

$3 \Downarrow 3$ (const)

$3+1 \Downarrow 4$ (δ)

$3 \Downarrow 3$ (const)

$1 \Downarrow 1$ (const)

`(fun x -> x+1) 3` $\Downarrow 4$ (app)

$\text{fun } x \to x+1 \Downarrow \text{fun } x \to x+1$ (fun)

$3 \Downarrow 3$ (const)

$3+1 \Downarrow 4$ (δ)

$3 \Downarrow 3$ (const)

$1 \Downarrow 1$ (const)

Examples of evaluation by substitution

```
((fun x -> fun y -> x+y) 3) 4
```

$\Downarrow 7$ (app)

(fun x -> fun y -> x+y) 3 $\Downarrow$ fun y -> 3+y (app)

fun x -> fun y -> x+y $\Downarrow$ fun x -> fun y -> x+y (fun)

3 $\Downarrow$ 3 (const)

fun y -> 3+y $\Downarrow$ fun y -> 3+y (fun)

4 $\Downarrow$ 4 (const)

3+4 $\Downarrow$ 7 (δ)

```
let f = fun x -> fun y -> x+y
in let g = f 1
   in g 2
```

$\Downarrow 3$ (let)

3 $\Downarrow$ 3

4 $\Downarrow$ 4

fun x -> fun y -> x+y $\Downarrow$ fun x -> fun y -> x+y (fun)

let g = (fun x -> fun y -> x+y) 1 in g 2 $\Downarrow$ 3 (let)

(fun x -> fun y -> x+y) 1 $\Downarrow$ fun y -> 1+y (app)

fun x -> fun y -> x+y $\Downarrow$ fun x -> fun y -> x+y (fun)

1 $\Downarrow$ 1 (const)

fun y -> 1+y $\Downarrow$ fun y -> 1+y (fun)

(fun y -> 1+y) 2 $\Downarrow$ 3 (app)

⋮

Substitution model evaluation rules

(Const) Const x $\Downarrow$ Const x

(Fun) Fun(a,e) $\Downarrow$ Fun(a,e)

(Rec) Rec(f,Fun(a,e)) $\Downarrow$ Fun(a,e[Rec(f,Fun(a,e))/f]

(δ) $e\ op\ e' \Downarrow v\ OP\ v'$
$e \Downarrow v$
$e' \Downarrow v'$

(δ) $op\ e \Downarrow OP\ v$
$e \Downarrow v$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ True
$e_2 \Downarrow v$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ False
$e_3 \Downarrow v$

(List) $[e_1, \ldots, e_n] \Downarrow [v_1, \ldots, v_n]$
$e_1 \Downarrow v_1$
$\vdots$
$e_n \Downarrow v_n$

(App) $e\ e' \Downarrow v$
$e \Downarrow$ Fun(a, e'')
$e' \Downarrow v'$
$e''[v'/a] \Downarrow v$

(Let) Let(a,e,e') $\Downarrow v'$
$e \Downarrow v$
$e'[v/a] \Downarrow v'$

Ex. evaluation by substitution

```
let x = 3 in x+1
```

Done above

```
(fun x -> x+1) 3
```

Ex. evaluation by substitution

```
let f = fun x -> fun y -> x+y
in let g = f 1 in g 2
```

Done above

Ex. evaluation by substitution

Let $\mathcal{F}$ denote "`rec fac (fun x -> if x=0 then 1 else x * fac(x-1))`".

```
let fac = 𝓕 in fac 2 ⇓ 2
   𝓕 ⇓ fun x -> if x=0 then 1 else x * 𝓕(x-1)
   (fun x -> if x=0 then 1 else x * 𝓕(x-1)) 2 ⇓ 2
      fun x -> if x=0 then 1 else x*𝓕(x-1) ⇓ fun x -> if x=0 then 1 else x*𝓕(x-1)
      2 ⇓ 2
      if 2=0 then 1 else 2 * 𝓕(2-1) ⇓ 2
         2=0 ⇓ False
            2 ⇓ 2
            0 ⇓ 0
         2 * 𝓕(2-1) ⇓ 2
            2 ⇓ 2
            𝓕(2-1) ⇓ 1
               𝓕 ⇓ fun x -> if x=0 then 1 else x * 𝓕(x-1)
               2-1 ⇓ 1
                  2 ⇓ 2
                  1 ⇓ 1
               if 1=0 then 1 else 1 * 𝓕(1-1) ⇓ 1
                  1=0 ⇓ false
                     1 ⇓ 1
                     0 ⇓ 0
                  1 * 𝓕(1-1) ⇓ 1
                  ⋮
```

Environment model evaluation rules

(Const) Const c, $\rho \Downarrow$ Const c

(Var) a, $\rho \Downarrow \rho(a)$

(Fun) Fun(a,e), $\rho \Downarrow <$Fun(a,e), $\rho >$

(Rec) Rec(f,e), $\rho \Downarrow <$ Rec(f,e), $\rho >$

(δ) e op e', $\rho \Downarrow v$ OP v'
e, $\rho \Downarrow v$
e', $\rho \Downarrow v'$

(δ) op e, $\rho \Downarrow OP$ v
e, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ True
e_2, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ False
e_3, $\rho \Downarrow v$

(List) $[e_1, \ldots, e_n]$, $\rho \Downarrow [v_1, \ldots, v_n]$
e_1, $\rho \Downarrow v_1$
$\vdots$
e_n, $\rho \Downarrow v_n$

(Let) Let(a,e,e'), $\rho \Downarrow v'$
e, $\rho \Downarrow v$
e', $\rho[a \mapsto v] \Downarrow v'$

(App) e e', $\rho \Downarrow v$
e, $\rho \Downarrow <$Fun(a, e''), $\rho' >$
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v'] \Downarrow v$

(App) e e', $\rho \Downarrow v''$
e, $\rho \Downarrow v$,
where $v = <$Rec(f, Fun(a, e'')), $\rho' >$
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

Evaluation in environment model

- $\emptyset$ **denotes the empty environment. We may write** $\emptyset[x \mapsto v]$ **as** $\{x \mapsto v\}$**.**

`let x = 3 in x+1`, $\emptyset$ $\Downarrow$ 4

3, $\emptyset$ $\Downarrow$ 3

x+1, $\{x \mapsto 3\}$ $\Downarrow$ 4

x, $\{x \mapsto 3\}$ $\Downarrow$ 3

1, $\{x \mapsto 3\}$ $\Downarrow$ 1

`(fun x -> x+1) 3`, $\emptyset$ $\Downarrow$ 4

fun x -> x+1, $\emptyset$ $\Downarrow$ <fun x -> x+1, $\emptyset$>

3, $\emptyset$ $\Downarrow$ 3

x+1, $\{x \mapsto 3\}$ $\Downarrow$ 4

x, $\{x \mapsto 3\}$ $\Downarrow$ 3

1, $\{x \mapsto 3\}$ $\Downarrow$ 1

Evaluation in environment model

```
let f = fun x -> fun y -> x+y,   ∅
in let g = f 1 in g 2
```

$\Downarrow 3$

$\overbrace{\text{fun } x \to \text{fun } y \to x+y,\ \emptyset \Downarrow \langle \text{fun } x \to \text{fun } y \to x+y,\ \emptyset \rangle}^{F}$

$\text{let } g = f\ 1 \text{ in } g\ 2,\ \{f \mapsto F\} \Downarrow 3$ ρ_1

$f\ 1, \rho_1 \Downarrow \langle \text{fun } y \to x+y,\ \{x \mapsto 1\} \rangle$

$f, \rho_1 \Downarrow F$

$1, \rho_1 \Downarrow 1$

$\text{fun } y \to x+y,\ \emptyset[x \mapsto 1] \Downarrow \langle \text{fun } y \to x+y,\ \{x \mapsto 1\} \rangle$

$g\ 2,\ \{f \mapsto F, g \mapsto \langle \text{fun } y \to x+y,\ \{x \mapsto 1\} \rangle\} \Downarrow 3$ ρ_2

$g, \rho_2 \Downarrow \langle \text{fun } y \to x+y,\ \{x \mapsto 1\} \rangle$

$2, \rho_2 \Downarrow 2$

$x+y,\ \{x \mapsto 1, y \mapsto 2\} \Downarrow 3$ ρ_3

$x, \rho_3 \Downarrow 1$

$y, \rho_3 \Downarrow 2$

Evaluation in environment model

Let $\mathcal{F}$ denote "`rec fac (fun x -> if x=0 then 1 else x * fac(x-1))`".

`let fac =` $\mathcal{F}$ `in fac 2`, $\emptyset \Downarrow 2$

$\mathcal{F}, \emptyset \Downarrow < \mathcal{F}, \emptyset >$

`fac 2`, $\rho_1 \Downarrow 2$ $\quad (\rho_1 = \{fac \mapsto < \mathcal{F}, \emptyset >\})$

`fac`, $\rho_1 \Downarrow < \mathcal{F}, \emptyset >$

`2`, $\rho_1 \Downarrow 2$

`if x=0 then 1 else x *` $\mathcal{F}$`(x-1)`, $\rho_1[x \mapsto 2] \Downarrow 2$

`x=0`, $\rho_1[x \mapsto 2] \Downarrow$ False

`x`, $\rho_1[x \mapsto 2] \Downarrow 2$

`0`, $\rho_1[x \mapsto 2] \Downarrow 0$

`x *` $\mathcal{F}$`(x-1)`, $\rho_1[x \mapsto 2] \Downarrow 2$

map

- **The most famous of all higher-order functions:**

```
let rec map f lis = if lis=[] then []
                    else (f (hd lis)) :: map f (tl lis);;
```

- `map (fun x->x+1) [1;2;3]` [2;3;4]
- `let incrBy n lis = map (fun x -> x+n) lis`

  Increment every element of lis by 1
- `let incrBy n = map (fun x -> x+n)`

  same

- **Type of** `map`**?** $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list

map exercises

- **addpairs: (int * int) list → int list**

```
let addpairs lis
    = map (fun (x,y) -> x+y) lis
```

- **appendString: string → string list → string list concatenates the first argument to the end of every string in the second argument**

```
let appendString s slis
    = map (fun s' -> s' ^ s) slis
```

- **incrall: int list list → int list list increments every element of every list in its argument**

```
incrall lis = map (map (fun x -> x+1)) lis
```

fold_right

- **Usually called** `reduce`**, but called** `fold_right` **in OCaml:**

```
let rec fold_right (f:'a->'b->'b) (lis:'a list) (z:'b) : 'b
      = if lis=[] then z else f (hd lis) (fold_right f (tl lis) z)
```

- `fold_right (fun s s' -> s @ s') ["a"; "b"; "c"] ""`

  "abc"

- `fold_right (fun x y -> x+y) [3;4;5] 0`

  12

- `fold_right (fun x y -> x::y) [3;4;5] []`

  [3;4;5]

- `let h f lis = fold_right (fun x y -> (f x)::y) lis []`

  h = map

Currying (cont.)

- **Can define functions** `curry` **and** `uncurry`**:**

```
let curry (f:'a * 'b -> 'c) : ('a -> 'b -> 'c) =
```

fun x -> fun y -> f (x,y)

```
let uncurry (f:'a -> 'b -> 'c) : ('a * 'b -> 'c) =
```

fun (x,y) -> f x y

```
usecurried (curry f_uncurried);;
useuncurried (uncurry f_curried);;
```

More h-o function examples

- `reverse (f: 'a -> 'b -> 'c) : ('b -> 'a -> 'c)`

  `reverse (fun x y -> x-y) 3 5 = 2`

  let reverse f = fun x y -> f y x

- `twice (f: 'a -> 'a ) : ('a -> 'a)`

  `twice (fun x -> x+1) 3 = 5`

  let twice f = fun x -> f (f x)

Representing sets as functions

```
type intset = int -> bool

let emptyset : intset =
```

fun n -> false

```
let member (n:int) (s:intset) : bool =
```

s n

```
let add (n:int) (s:intset) : intset =
```

fun n' -> n'=n or s n'

Representing sets as functions (cont.)

```
let union (s1:intset) (s2:intset) : intset =
```

fun n -> s1 n or s2 n

```
let intersection (s1:intset) (s2:intset) : intset =
```

fun n -> s1 n && s2 n

```
let remove (n:int) (s:intset) : intset =
```

fun m -> s m && m <> n

```
let complement (s:intset) : intset =
```

fun n -> not (s n)

```
let intsAbove (n:int) : intset =
```

fun m -> m > n

Representing dictionaries as functions

- **Similarly, implementing an *environment* (a.k.a. *dictionary*), as in our mps, means defining a representation "`type environment =` *something*", and operations:**
  - `let emptyEnv : environment`
  - `fetch (id:id) (env:environment) : value`
  - `extend (id:id) (v:value) (env:environment) : environment`
- **These operations need to act in an "environment-like" way, e.g.**
  - `fetch "x" emptyEnv` **throws an exception**
  - `fetch "x" (extend "x" (IntConst 3) emptyEnv)` **returns** `IntConst 3`