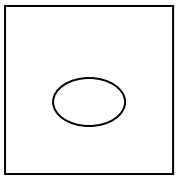
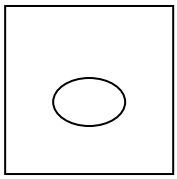
KEY for 2002 final exam

1. D

Side effect:
There is no formal definition of side-effect. Generally, a closed expression e has no side-effect if it evaluates to the same value anywhere and does not pollute the outside "environment".

There are four cases, side effects can occur:
1. eval(e) affects other parts of the program. (via state, control flow).
2. eval(some part of the program) affects eval(e) . (via state, control flow).
3. eval(e) affects the outside world/environment. (output: printing, writing to disk, controlling robot)
4. The outside world/environment affects eval(e). (input)





pure functional language has no side-effects

2. D

3. B

A<:B A is a subtype of B
{x:int; y:int } <: {x:int}

If functions *A* and *B* are equivalent except that *B* returns a subtype of what *A* returns, then *B* is a subtype of *A*.
A<B
A: t1→t1' B: t1→t2' t1'<t2'

However, if *A* and *B* are the same except that *B*'s argument is a subtype of *A*'s argument, then *A is a subtype of B*.
A<B
A: t1→t1' B: t2→t1' t2<t1

Function: if the return result contains more labels, the result is more flexible
Function: if the argument need more labels, it means more requirements, less flexible

4. B

5. A

6. A

There is some analogy between function and functor.
A function takes a value and returns a value
A functor takes a module and returns a module
f(…) -> …
f(a piece of code) -> (a piece of code)

values have types, we call them regular types
code also have types, we call them signature in ML, (or "interface" in other languages)

A function maps from a regular type to another regular type.
A functor maps from a signature to another signature.

7. B

8. A

9. B

10. C

11. B

12. C

type soundness:
If Gamma|-*e* : *Tau can be derived from the type system, then in the process of evaluating e, a "stuck state" is never reached.*

A → B But in "Remark 3", it states "B→A", so remark 3 is wrong.

A "stuck state" generally means you can not find any operational rules to do futher evaluation, e.g. (0 1)

e: (let f x = x x in f f)
this expression is not sound in type.(It does not type check in ocaml). But a "stuck state" is never reached in evaluationg e.

Whether a language is **deterministic** or not depends on the language's op sem.

And example of non-deterministic language:
Coin language
coin_expr = T | F | Flip of coin_expr

Flip (operation semantics) rule:

```
e=> T                                  e=> T
-------------------------------------  --------------------------
Flip(e) => T                           Flip(e) => F
```

Flip(Flip(T)) => ?? (not deterministic)

People usually uses deterministic languages for programming. Bool, D, OCaml, C/C++, Java are deterministic languages.

Turning complete. partial recursive functions.
Related topic:
There exists some functions which are not computable. So those functions are not partial recursive functions. And we cannot write a program in Turing maching/D/DSR/OCaml/C++/Java to compute these non-computable functions. One example of a non-computable function is HALT, which takes a program as its argument and return whether the program terminates or not.

Normallizing:

Bool language is normalizing. D, OCaml, C/C++, Java are not nomallizing.

---

13. B

14.

∀ C, C[x x x] => v, iff C[x x]=>v'

Find a counter example:

Let a = Function x -> 1 In
Let x = Function x -> x a In
(x x)
=> 1

Let a = Function x -> 1 In
Let x = Function x -> x a In
(x x x)
=> (1 x)
does not evaluate to any value

15.

1. Function a -> Function b -> {x=a.x; y=a.y; z=b.z}

2. There is no record operation to remove a field in DSR. The only possible approach is to create a new record with all fields, except "x". However for DSR record, you can access a field only if you know its label and then "Select", there is no way to access unknown labels. So it's impossible to remove a field in DSR/caml.

16.

Let x=3 In (Function x->x+1) 10

17.

let rec f x = f;;
1. f: 'a → 'b
2. x: 'a
3. f: 'b

The inference algorithm tries to unify 1 and 3, but a clever type inference algorithm can detects that this unification cannot be done in a finite time.
f: ('a → 'b)
f: ('a→('a→'b))
…

Since it can not unify 'b and ('a → 'b), the two are considered as different types. And a type error is reported.

19.

So this question is another way to test whether you really understand higher-order function and currying.

First-order functions		Higher-order functions
let f x = …	or	let f=fun x -> …
let f x y = …		let f=fun x -> (fun y -> (…)) // can returns a function
`module F (X:S1) =` `struct ⋯ end`	or	`module F=functor(X : S1) ->` `struct ⋯ end`
`module F (X:S1) (Y:S2) =` `struct ⋯ end`		`module F=functor(X : S1)->` `(functor(Y : S2)->struct ⋯ end)` **// functors can returns a functor** **// ocaml does not support this form**

functor application examples:

IntRect = Rect(Point(Int)),
FloatRect = Rect(Point(Float))
where Rect is a 2nd-order functor, Point a 1st-order functor and Int is a module

XRect = (Rect Point) The application "Rect Point" takes a functor as the argument
XRect(Int)
XRect(Float)

So with curried functor and functor application, we can build higher-order module systems. The argument of a functor can be a functor and a functor can return a functor.

Ocaml does not have such higher-ordered module systems.

20. 3, 5
1. uncaught exception
2. division by zero
3. negating a function ---------- negation is an operator applied to integers
4. a match expression in which none of the cases matches
5. selection of a record field which is not present ---------- wrong record type
6. access of an array element out of the array size bound.

21.
a) Red = {red=255; green=0; blue=0}

b) let obj = {
red=255;
green=0;
blue=0;
brightness = Function this -> Function _ -> …
}

c) There is no programmers who does not make any mistake. Having an explicity syntax, can let the compiler check whether the immutable property is violated or not.

Longer Answer
(15 points)

```
let rec lookup le_list `l =
        match le_list with [] -> ("",Int(0))
        | (`l', e')::xs -> if (`l'=`l) then (`l',e') else lookup le_list xs

eval expr =
….
| Enum(`l) → Enum(`l)
| Ematch(e, le_list) -> let ee = eval(e) in
        (match ee with Enum(`l) ->
                let (`l', e')=lookup le_list `l in if(`l' ne `l) then raise runtime_mismatch
                                                        else eval(e');
         | _ -> raise SyntaxError;
        )
```

23.

$$\frac{}{\Gamma \vdash Enum(Lab(l_i) : Enu([Lab(l_1), \dots, Lab(l_n)])} \text{ (Enum Rule)}$$

$$\frac{\Gamma \vdash e:Enu([Lab(l_1), \dots, Lab(l_n)]),\ \Gamma \vdash e_1:\iota, \dots, \Gamma \vdash e_n:\iota}{\Gamma \vdash EMatch(\ e, [Lab(l_1), \dots, Lab(l_n)])\ ) : \iota} \text{ (EMatch Rule)}$$

KEY for 2001 final exam

1.
Type inference: initially give all variables arbitrary types, `'a, 'b`, and then to *unify* or the types

x: ‘a
x: ‘b→’c
x: ‘c
‘c: int → d’

try to unify them together. But the type inference algorithm detects a inference recursion. If any recursion occur, the inference will never teminate.

‘c = (‘b→’c)

x: `b → ‘c
: `b → (`b → `c)
: …

let rec f x = f x;;	**let rec f x = f x + 1;;**
f: ‘a → ‘b	f: ‘a → ‘b
x: ‘a	x: ‘a
	‘b: int
unify all.	
f: ‘a →’b	f: ‘a -> int

2.
Exception, State. Object inherently has state.

3.
Cannot change the value of a variable after its definition. Immutable variable does not involve states, which is an important feature of functional programming. In functional programming, you don’t need to care about which expression is evaluated first. Can be easily analysed in mathimatics.

4.
if-then-else, function are expressions
(if f(x)=5 then 10 else 20) + g(10) is an expression
(function x -> g(5)+x) 10 is an expression

5.
xn2 is raised

6.

```
eval expr =
….
| [[]]→ [[]]
| a::::b -> eval(a)::::eval(b)
| Match(l,e,x,xs,e’)) ->
        let l’=eval(l) in
        match l’ with [[]] -> eval(e)
        | x::::xs -> eval(e’)
        | _ -> raise TypeMismatch
```

7.

Γ |- [[]]: ι list for arbitrary *Tau*

Γ |- x: ι list, Γ |- xs: ι list

Γ |- x::::xs : ι list

Γ |- l: ι list, Γ |- e: ι1 , Γ |- x: ι , Γ |- xs: ι list, Γ |- e': ι1

Γ |- Match(l, e,x,xs,e') : ι1

8.

Types are precise and descriptive comments on the functionality of the code. This makes the syntax checking more easier for compilers. The compiler can store data according to its type without maintaining its type information in runtime. The data layout can be optimized during compilation time. The compiler can statically detect many type errors, reduce runtime checking.

9.

a)

```
let c=colorPointClass() in
c <- isnull ()
```

b) we are not sure precisely what "*magnitude"* will be executed at runtime.

c)

to implement non-dynamic dispatch, change

```
super.magnitude self () =>
super.magnitude super ()
```

similar example:

```
Let rectClass = Function _ -> {
  getWidth = Function this -> Function _ -> 10;
  getLength = Function this -> Function _ -> 1;
  area = Function this -> Function _ ->
    mult ((this.getLength) this {}) ((this.getWidth) this {})
} In

Let squareClass = Function _ ->
  Let super = rectClass {} In {

  getLength = (super.getLength);

  (* We override width to be the same as length *)
  getWidth = (super.getLength);

  areaStatic = Function this -> Function _ ->
    (super.area) super {};
  areaDynamic = Function this -> Function _ ->
    (super.area) this {}
} In
```