PL Midterm Solution

1 a) (1) O/×
(2) O
(3) O
(4) × It is detected by semantic analyzer.

b) (1) Von Neumann architecture
(2) memory cells
(3) lambda calculus
(4) constant values

c) (1) `(c 0)`
(2) `(call/cc (lambda (c) (mul xs c)))`

2 a) Orthogonality means a relatively small set of primitive constructs can be combined in a relatively small number of ways.
Too little orthogonality – a lot of exceptions to remember

b) (1) (B) static type binding
(2) (A) static storage binding

c) (1) Java
(2) None

d)

Javascript	`-66<"Snoopy"`	⇒	`-66<nan`	⇒ false
Perl	`-66<"Snoopy"`	⇒	`-66<0`	⇒ true
Scheme	`(< -66 "Snoopy")`			⇒ run-time type error
SML	`-66<"Snoopy"`			⇒ compile-time type error

e) C++ adopts copy semantics. Thus, each of **`a`** and **`b`** has its own list structure.
Scheme adopts sharing semantics. Thus, **`a`** and **`b`** share the same list structure.
Sharing is dangerous in the presence of side effects. C++ adopts copying, because it allows side effects; whereas, Scheme adopts sharing, since, at least in its functional part, it doesn't have side effects.

3 First of all, if the declaration isn't prohibited, the variable **x** will have a polytype

((int → 'a) → 'a) ref

With this type, the following SML program will pass compile-time type checking.

```
x := (fn x=>(x 2)+3);
```

∵ lhs type **((int → 'a) → 'a) ref**

rhs type **(int → int) → int**

∴ It follows that **'a=int**.

```
!x (fn x=>true);
```

∵ function type **(int → 'a) → 'a**

argument type **'b → bool**

∴ It follows that **'a=bool** and **'b=int**.

However, the evaluation of

```
!x (fn x=>true);
```

will cause a run-time error, since

```
!x (fn x=>true)
⇒  (fn x=>(x 2)+3)(fn x=>true)
⇒  ((fn x=>true) 2)+3
⇒  true+3
```

4 a) Pro

Type safe: Type errors are all detected at compile time

Efficiency: No type checking code at run time

Con

Inflexible: Variables have fixed types, e.g. **fn x => x x** is illegal in SML.

b) False

There are no type checking on assignment, parameter passing, and function value returning, because it is exactly these three operations that are used to change the type of the value stored in a variable.

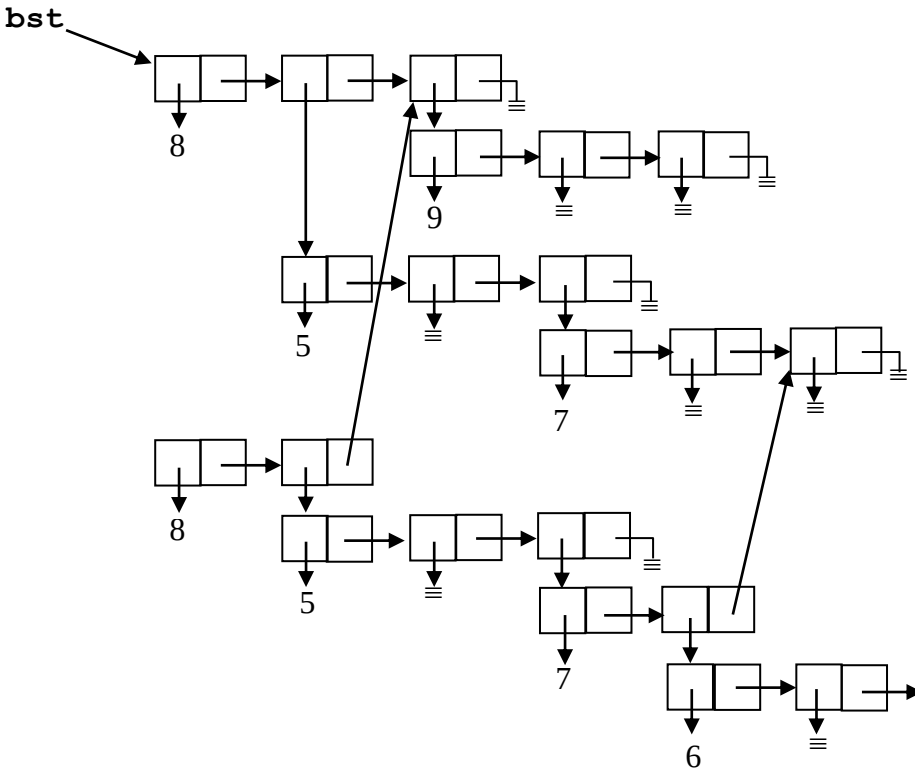
5 a) See lecture

b) See lecture

c) See lecture

6 a) `'(8 (5 () (7 () ())) (9 () ()))`

b)



c)

d) When a node is inserted as a child of node x in a binary search tree rooted at node r, the path from r to x must be searched and copied.
Thus, the insertion takes $O(n)$ time and space, where n is the length of the path from r to x.

7 a) Θ is a fixed-point combinator, because

$\Theta f = AAf$ (1)
$= f(AAf)$ (2)
$= f(\Theta f)$ by (1)

Moreover, it can only work with lazy evaluation, because in step (2) the argument AAf passed to f has to be delayed.

b) Define $\Theta = AA$ where $A = \lambda x.\lambda y.\, y(\lambda z.\, xxyz)$.
Then, Θ is a fixed-point combinator, because

$\Theta f = AAf$ (1)
$= f(\lambda z.\, AAfz)$ (2)
$= f(AAf)$ eta conversion
$= f(\Theta f)$ by (1)

Moreover, this combinator works with eager evaluation, ∵ in step (2) the λ-exp $\lambda z.\, AAfz$ passed to f in effect delays the argument AAf mentioned in part a).

c) $\Theta\ (\lambda f.\lambda a.\lambda b.\text{if } b = 0 \text{ then } a \text{ else } f\ b\ (a \bmod b))$

8 a) compile λx.((λy.x)2)

= abstract x (compile ((λy.x)2))

= abstract x ((compile λy.x) (compile 2))

= abstract x ((abstract y x) 2)

= abstract x (K x 2)

= S (abstract x (K x)) (abstract x 2)

= S (S (abstract x K) (abstract x x)) (abstract x 2)

= S (S (K K) I) (K 2)

b) S I (K 3) (S (S (K +) I) (K 4))

= I (S (S (K +) I) (K 4)) (K 3 (S (S (K +) I) (K 4)))

= S (S (K +) I) (K 4) (K 3 (S (S (K +) I) (K 4)))

= S (S (K +) I) (K 4) 3

= S (K +) I 3 (K 4 3)

= K + 3 (I 3) (K 4 3)

= + (I 3) (K 4 3)

= + 3 (K 4 3)

= + 3 4

= 7