CS 421 Spring 2011 Final **Solution**

Friday, May 6, 2011

Name	
NetID	

- You have **180 minutes** to complete this exam
- This is a **closed book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 18 pages to the exam. Please verify that you have all 18 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	20	
2	20	
3	10	
4	10	
5	20	
6	10	
7	30	
8	10	
9	10 + 10 XC	
10	10	
11	10	
12	10	
Total	**170 + 10XC**	

1. (20 pts) (Recursive functions on lists)

   (a) Write a function `sum2` that turns a list of integer lists into a list of integers, by taking the sum of the first two elements of each of the contained lists.

   ```
   sum2 [[1;2;3;4]; [5;6]; [7;8;9]] ==> [3; 11; 15]
   ```

   Assume each contained list has at least two elements.

   i. Give the type of sum2

   ```
   int list list -> int list
   ```

   ii. Write this function directly using only recursion

   ```
   let rec sum2 l = match l with
       [] -> []
   | (x :: y :: _) :: zs -> (x + y) :: sum2 zs
   ```

   iii. Write this using `List.map`.

   (Note: the type of `List.map` is `('a -> 'b) -> 'a list -> 'b list`)

   ```
   let sum2 = List.map (fun (x :: y :: _) -> x + y)
   ```

(b) Write a different version of `sum2` in which the contained lists are not guaranteed to have two elements; if they have fewer, they are just ignored.

```
sum2b [[1;2;3;4]; [5]; [6;7]; []; [8;9;10]] ==> [3; 13; 17]
```

i. Give the type of `sum2b`

```
int list list -> int list
```

ii. Write this function directly using only recursion

```
let rec sum2b l = match l with
      [] -> []
    | (x :: y :: _) :: zs -> (x + y) :: sum2 zs
    | _ :: zs -> sum2 zs
```

iii. Write this using `List.fold_right` (it cannot be written using `List.map`).
(Note: the type of `List.fold_right` is `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`)

```
let sum2b l = List.fold_right (fun x a ->
                match x with
                    y :: z :: _ -> (y + z) :: a
                  | _ -> a
               ) l []
```

2. (20 pts) (Higher-order functions) You are given this type definition for binary trees with labelled leaves:

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

   (a) Write a function `labellengs: string tree -> int tree`. `labellengs t` is a tree with the same shape as `t`, whose leaf nodes contain the lengths of the corresponding labels in `t`. (Use function `String.length` to find the length of a string.)

```
let t = Node(Leaf "abc", Node(Leaf "a", Leaf "bc"))

labellengs t ==> Node(Leaf 3, Node(Leaf 1, Leaf 2))
```

```
let rec labellengs t = match t with
    Leaf s -> Leaf (String.length s)
  | Node(a, b) -> Node(labellengs a, labellengs b)
```

   (b) Write higher-order function `treemap` that take a function `f` and applies it to the value in each leaf node:

```
let it = treemap String.length t
==>  Node(Leaf 3, Node(Leaf 1, Leaf 2))

treemap (fun x -> x+1) it
==>  Node(Leaf 4, Node(Leaf 2, Leaf 3))
```

      i. Define `treemap`

```
let rec treemap f t = match t with
    Leaf s -> Leaf (f s)
  | Node(a,b) -> Node(treemap f a, treemap f b)
```

      ii. Give the polymorphic type of `treemap`

```
('a -> 'b) -> 'a tree -> 'b tree
```

(c) Write the function `height: 'a tree -> int`. The height of a tree is the length of the longest path from the root to a leaf node. (A tree consisting of a single Leaf node has height zero.) You can assume the function `let max x y = if x > y then x else y`

```
height t ==> 2
```

```
let rec height t = match t with
    Leaf _ -> 0
  | Node(a, b) -> 1 + max (height a) (height b)
```

(d) Higher-order function `treefold` is analogous to fold_right on lists. Specifically, `treefold f g t` returns a value obtained by applying `f` to the values in leaf nodes, and applying `g` to the pair of values returned from the children of an internal node.

```
let t1 = Node (Leaf 4, Node (Leaf 2, Leaf 3))
treefold string_of_int (fun s1 s2 -> s1 ^ "," ^ s2) t1
==> "4,2,3"
```

i. Use treefold to define `height`:

```
let height = treefold  (fun _ -> 0)   (fun x y -> 1 + max x y)
```

ii. Define `treefold`

```
let rec treefold f g t = match t with
    Leaf x -> f x
  | Node(a,b) -> g (treefold f g a) (treefold f g b)
```

iii. Give the polymorphic type of `treefold`

```
('a -> 'b) -> ('b -> 'b -> 'b) -> 'a tree -> 'b
```

3. (10 pts) (Computation as simplification) Use these simplification rules:

   - **(let)** let x = v in e ⇒ e[v/x] (v is a value)
   - **(β-value)** (fun x -> e) v ⇒ e[v/x] (v is a value)
   - **(delta)** Built-in operations: 1+1 ⇒ 2, 1>2 ⇒ false, hd [1;2;3] ⇒ 1, etc

   to simplify the following expression:

   ```
   let compose = fun f -> fun g -> fun x -> (let a = g x in f a a)
   in compose (+) hd [3;4]
   ```

   (*Hint*: Our solution has 7 steps.)

   ```
   let compose = fun f -> fun g -> fun x -> (let a = g x in f a a)
   in compose (+) hd [3;4]
   ==>(let) (fun f -> fun g -> fun x -> (let a = g x in f a a)) (+) hd [3
   ==>(beta) (fun g -> fun x -> (let a = g x in (+) a a)) hd [3;4]
   ==>(beta) (fun x -> (let a = hd x in (+) a a)) [3;4]
   ==>(beta) let a = hd [3;4] in (+) a a
   ==>(delta) let a = 3 in (+) a a
   ==>(let) (+) 3 3
   ==>(delta) 6
   ```

4. (10 pts) (Type-checking) Use the type rules of OCaml, given here:

   Variable: $\Gamma \vdash x : \tau,$ if $\Gamma(x) = \sigma$ and $\tau \leq \sigma$

   Application:

   $\Gamma \vdash e_1 e_2 : \tau$
   - $\Gamma \vdash e_1 : \tau' \to \tau$
   - $\Gamma \vdash e_2 : \tau'$

   Abstraction:

   $\Gamma \vdash$ fun x -> $e : \tau \to \tau'$
   - $\Gamma[x : \tau] \vdash e : \tau'$

   Tuple:

   $\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2$
   - $\Gamma \vdash e_1 : \tau_1$
   - $\Gamma \vdash e_2 : \tau_2$

   Let:

   $\Gamma \vdash$ let x=e in $e' : \tau'$
   - $\Gamma \vdash e : \tau$
   - $\Gamma[x : GEN_\Gamma(\tau)] \vdash e' : \tau'$

   to prove the following. (You may use names like Γ_1 to abbreviate type environments, to save some writing; be sure to say clearly what the abbreviations stand for.)
   Be sure to label each line with the rule you used to infer it.

   $\Gamma_0 \vdash$ let f = fun x -> fun y -> (x,y) in f 3 "ab" : $int * string$

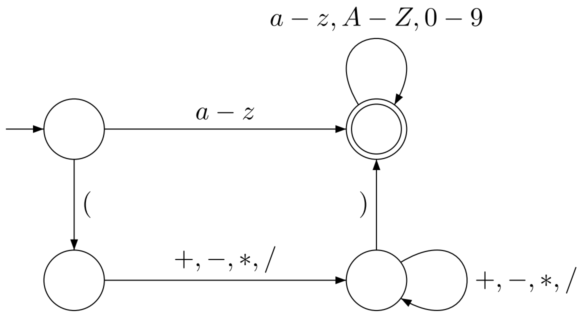
$\Gamma_0 \vdash$ let f = fun x -> fun y -> (x,y) in f 3 "ab" : $int * string$ (let)
- $\Gamma_0 \vdash$ fun x -> fun y -> (x,y) : $\alpha \to \beta \to \alpha * \beta$ (abs)
  - $\Gamma_0[x : \alpha] \vdash$ fun y -> (x,y) : $\beta \to \alpha * \beta$ (abs)
    - $\Gamma_0[x : \alpha][y : \beta] \vdash$ (x,y) : $\alpha * \beta$ (tuple)
      - $\Gamma_0[x : \alpha][y : \beta] \vdash$ x : α (var)
      - $\Gamma_0[x : \alpha][y : \beta] \vdash$ y : β (var)
- $\Gamma_0[f : \forall\alpha, \beta.\alpha \to \beta \to \alpha * \beta] \vdash$ f 3 "ab" : $int * string$ (app)
  - $\Gamma_0[f : \forall\alpha, \beta.\alpha \to \beta \to \alpha * \beta] \vdash$ f 3 : $string \to int * string$ (app)
    - $\Gamma_0[f : \forall\alpha, \beta.\alpha \to \beta \to \alpha * \beta] \vdash$ f : $int \to string \to int * string$ (var)
    - $\Gamma_0[f : \forall\alpha, \beta.\alpha \to \beta \to \alpha * \beta] \vdash$ 3 : int (var)
  - $\Gamma_0[f : \forall\alpha, \beta.\alpha \to \beta \to \alpha * \beta] \vdash$ "ab" : $string$ (var)

5. (20 pts) (Lexing) Give DFAs and regular expressions (in ocamllex notation) for the following sets of tokens; you do not need to label any of the nodes, but write the start node on the left.

   (a) An *identifier* in OCaml is either a lower-case letter followed by any number of letters or digits, or a non-empty sequence of the symbols +, -, *, or / (with no spaces) surrounded by parentheses.

```
Regex:   (['a'-'z']['a'-'z' 'A'-'Z' '0'-'9']*)
      | '(' ['+' '-' '*' '/']+ ')'
```

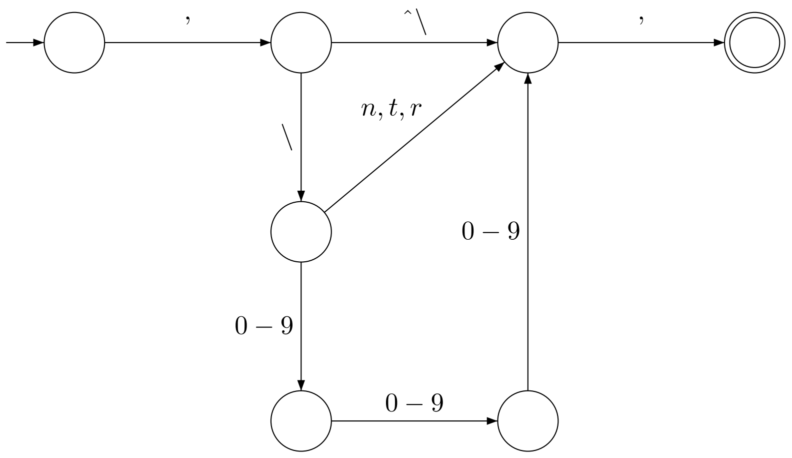
DFA:



   (b) A *character literal* is one of the following contained in single quotes: a single character other than \, or a \ followed by either one of the letters n, t, or r, or by exactly three digits.
   Note that in regular expressions single quote and backslash can be escaped using a backslash (\\ and \').

```
Regex: '\'' ( [^ '\\']
            | '\\' ( ['n' 't' 'r'] | ['0'-'9'] ['0'-'9'] ['0'-'9'])
       ) '\''
```

DFA:



6. (10 pts) (Grammars) A grammar for arithmetic expressions should have these properties, if possible:

   a. It should be unambiguous
   b. It should be LL(1)
   c. It should enforce left-associativity of + and *
   d. It should enforce precedence of * over +

   None of the following grammars satisfies all of these criteria. For each grammar, list all the properties that it fails to satisfy.

   (a) $E \rightarrow id \mid E + id \mid E \times id \mid (E)$

   Fails (b), (d)

   (b) $E \rightarrow id \mid id + E \mid id \times E \mid (E)$

   Fails (b), (c), (d)

   (c)
   $$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow P \times T \mid P \\ P &\rightarrow id \mid (E) \end{aligned}$$

   Fails (b), (c)

   (d)
   $$\begin{aligned} E &\rightarrow id\ F \mid (E) \\ F &\rightarrow \epsilon \mid + E \mid \times E \end{aligned}$$

   Fails (c), (d)

   (e)
   $$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times P \mid P \\ P &\rightarrow id \mid (E) \end{aligned}$$

   Fails (b)

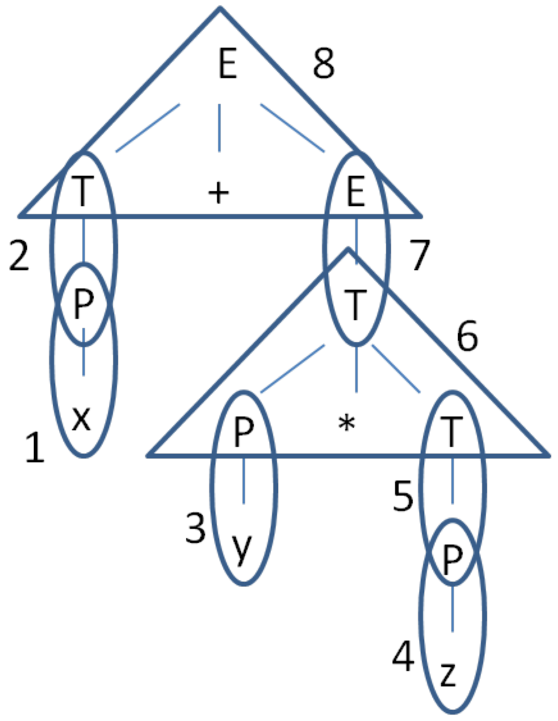
   (f)
   $$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow \epsilon \mid + E \\ T &\rightarrow P\ T' \\ T' &\rightarrow \epsilon \mid \times E \\ P &\rightarrow id \mid (E) \end{aligned}$$

   Fails (c)

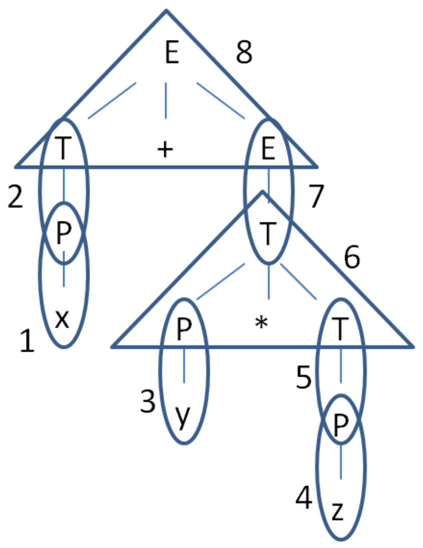
7. (30 pts) (Bottom-up parsing) This is a right-recursive stratified grammar:

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow P \mid P * T \\ P &\rightarrow id \end{aligned}$$

(a) Following is the parse tree for sentence x+y*z. Circle and number the handles of this tree.

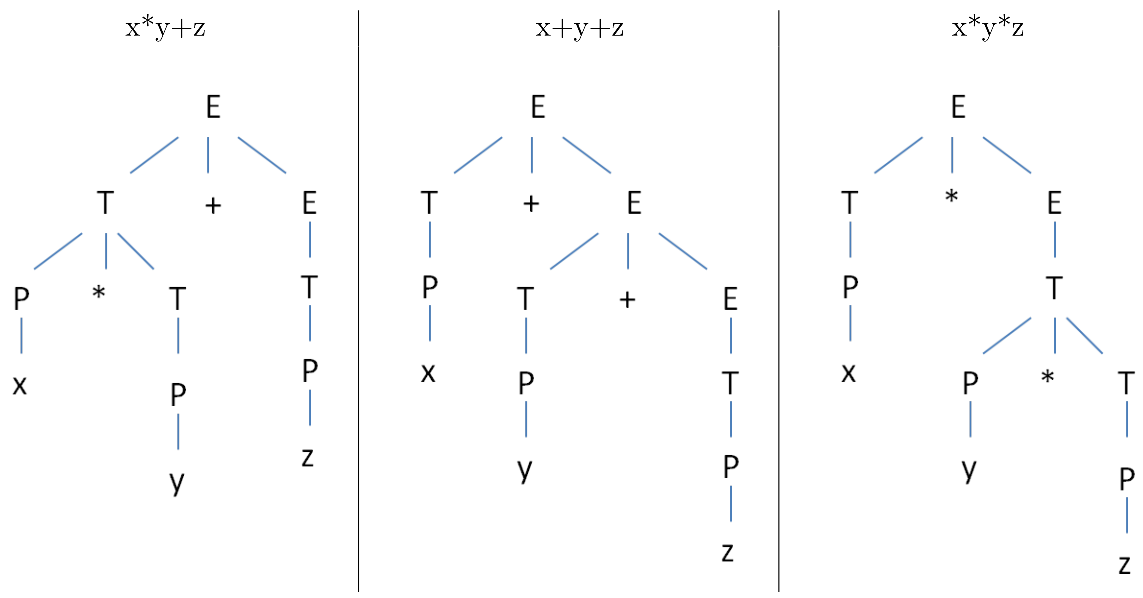


(b) Give the shift-reduce parse for that sentence:
(The parse tree is given here again for reference)



Action	Stack (top at right)	Input
Shift	empty	x+y*z
Reduce P→id	x	+y*z
Reduce T→P	**P**	**+y*z**
Shift	**T**	**+y*z**
Shift	**T +**	**y*z**
Reduce P→id	**T + y**	***z**
Shift	**T + P**	***z**
Shift	**T + P ***	**z**
Reduce P→id	**T + P * z**	**eof**
Reduce T→P	**T + P * P**	**eof**
Reduce T→P*T	**T + P * T**	**eof**
Reduce E→T	**T + T**	**eof**
Reduce E→T+E	**T + E**	**eof**
Accept	**E**	**eof**

(c) Write parse trees for these three sentences:



(d) By considering the parse trees above, give the correct action (Shift, Reduce by a given production, Reject, or Accept) under each of the following circumstances:

Stack (top at right)	Lookahead symbol	Action
T + z	*	**Red P→id**
x	*	**Red P→id**
P	*	**Shift**
E	eof	**Accept**
P	z	**Reject**
T + P * z	x	**Reject**

(e) The above grammar is not LL(1), but here is a left-factored version that is:

$$\begin{aligned} E &\rightarrow T\,E' \\ E' &\rightarrow \varepsilon \mid + E \\ T &\rightarrow P\,T' \\ T' &\rightarrow \varepsilon \mid *\ T \\ P &\rightarrow id \end{aligned}$$

Suppose the tokens are given by this type:

```
type token = Id of string | Plus | Star
```

A recursive descent recognizer for this grammar would consist of five functions, one for each non-terminal, each of type `token list` → `token list`. *Assume* you have functions `parseT`, `parseT'`, and `parseP`, and write `parseE`, `parseE'`:

```
let rec parseE  toks = parseE' (parseT toks)

and parseE' toks = match toks with
                        Plus :: r -> parseE r
                      | _ -> toks


and parseT  toks = parseT' (parseP toks)

and parseT' toks = match toks with
                        Star :: r -> parseT r
                      | _ -> toks

and parseP  toks = match toks with
                        Id _ :: r -> r
                      | _ -> failwith "parsing error"
```

8. (10 pts) (Function objects) The following sort function accepts a Comparator object that it uses for comparisons. Comparators are function objects, using the method name `less` instead of `apply`.

```
interface Comparator {
   public boolean less (int x, int y);
}

void sort (int[] A, Comparator c) {
   ... if (c.less(A[i], A[j])) ..
}
```

For example, for ordinary arithmetic comparisons, define this comparator:

```
class Less implements Comparator {
   public boolean less (int x, int y) { return x < y; }
}
```

and use it: "`int[] nums; ...  sort(nums, new Less()); ...`".

Define these classes:

```
class LowerMag implements Comparator {
     //  less(x,y) if the absolute value of x is less than
     //  the absolute value of y.  Do not use any auxiliary functions.
    public boolean less (int x, int y) {
       return (x<0 ? -x : x) < (y<0 ? -y : y);
    }
}

class LowerLetter implements Comparator {
  // Assuming that x and y are (ASCII codes of) capital or small letters,
  // less(x, y) returns true if x is an earlier letter in the alphabet
  // than y, without regard to case
  // To convert code for a lower-case letter to upper-case: x - 'a' + 'A'
    public boolean less (int x, int y) {
       if ('a' <= x && x <= 'z') x = x-'a'+'A';
       if ('a' <= y && y <= 'z') y = y-'a'+'A';
       return x<y;
    }
}
```

```
class Or implements Comparator {
   // Combines two other comparators.
   // Possible use: sort(nums, new Or(new LowerLetter(), new LowerDigit()))
   // This means one integer is less than the other if it is either
   // a lower letter (in ASCII code) or a lower digit.

   Comparator c1, c2;

   public Or(Comparator c1, Comparator c2) {
      this.c1 = c1; this.c2 = c2;
   }
  public boolean less (int x, int y) {
     return c1.less(x, y) || c2.less(x, y);
  }
}
```

An integer function object is an object implementing this interface:

```
interface Intfun {
   int apply (int i);
}
```

For example:

```
class Absval implements IntFun {
   int apply (int i) { return i<0 ? -i : i; }
}
```

Complete this definition:

```
class ComposeComp implements Comparator {
   //  Applies a comparator after applying an integer function.
   //  Possible use: sort(nums, new ComposeComp(new Less(), new Absval()))

   Comparator c;  Intfun f;

   public ComposeComp(Comparator c, Intfun f) {
      this.c = c; this.f = f;
   }
  public boolean less (int x, int y) {
     return c.less(f.apply(x), f.apply(y));
  }
}
```

9. (10 pts + 10 XC) (Loop invariants) Given the loop invariants for the following loop; we have given the pre- and post-conditions:

   (a) Non-destructive reversal: This loop moves A to B in reverse order.
   Pre-condition: $i = 0$ & $n = |A|$ & $n = |B|$
   Post-condition: $\forall 0 \leq k < n . B[k] = A[n-1-k]$

   ```
   while (i<n) { B[n-i-1] = A[i]; i = i+1; }
   ```

   Invariant: $\forall 0 \leq j < i . B[n-j-1] = A[j]$ & $i \leq n$

   (b) Calculate average: Here, A is an array of doubles.
   Pre-condition: $m = 0.0$ & $i = 0$
   Post-condition: $m =$ mean of A

   ```
   while (i<A.length) { m = m + A[i]/A.length; i = i+1; }
   ```

   Invariant: $m =$ sum of $A[0..i-1]/|A|$ & $i \leq |A|$

   (c) **(10pts Extra Credit)** Non-destructive merge: This loops merges m elements from A with n elements from B into array C. A and B each have an additional sentinel element containing value "∞" which is larger than any other value contained in those arrays.
   Pre-condition: $|A| = m+1$ & $|B| = n+1$ & $|C| = m+n$ & $A[m] = \infty$ &
   & $B[n] = \infty$ & $a = b = c = 0$ & $\forall 0 \leq i < m-1 . A[i] \leq A[i+1]$ &
   & $\forall 0 \leq i < n-1 . A[i] \leq B[i+1]$

   Post-condition: $\forall 0 \leq i < m+n-1 . C[i] \leq C[i+1]$ &
   & C has the same elements as $A[0 \ldots m-1]$ and $B[0 \ldots n-1]$

   ```
   while (a < m || b < n) {
      if (A[a] < B[b]) {
         C[c] = A[a];
         a = a+1;
      }
      else {
         C[c] = B[b];
         b = b+1;
      }
      c = c+1;
   }
   ```

   Invariant: $\forall 0 \leq j < c-1 . C[j] \leq C[j+1]$ &
   & $C[0 \ldots c-1]$ has the same elements as $A[0 \ldots a-1]$
   and $B[0 \ldots b-1]$.

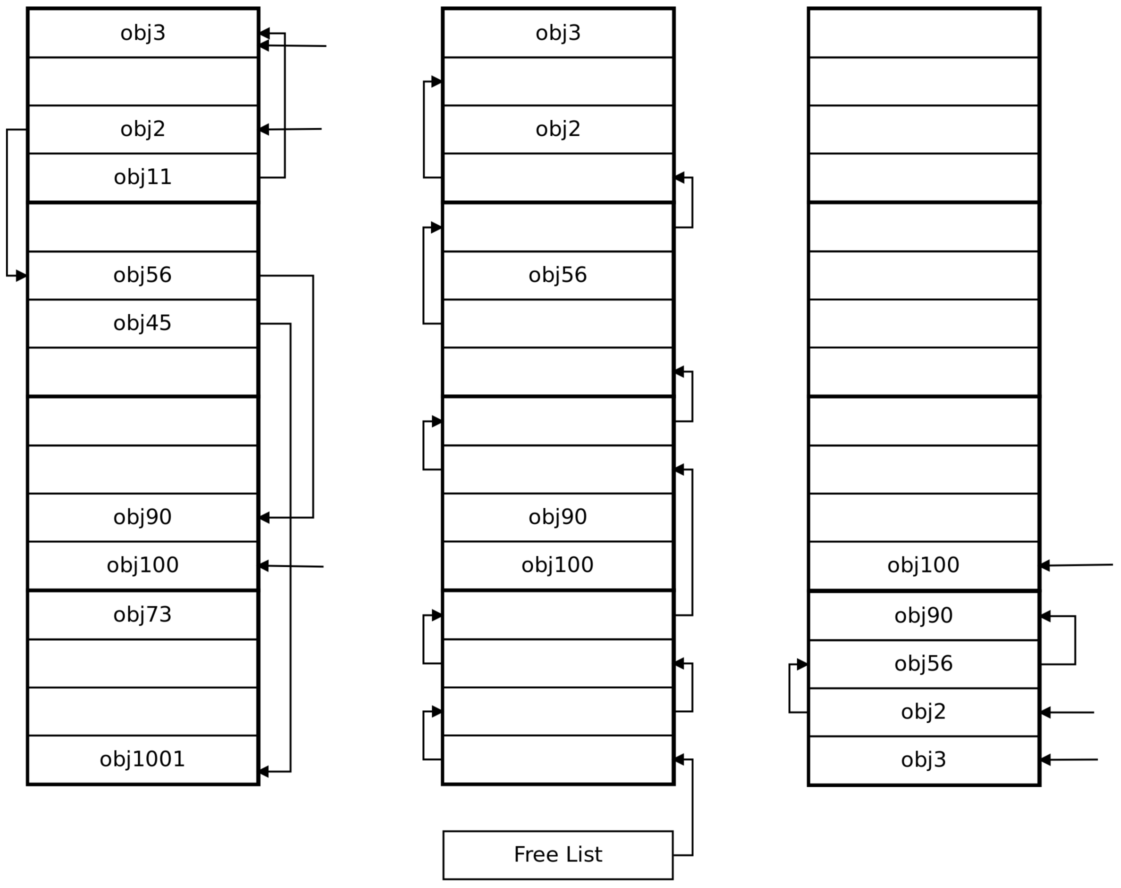
10. (10 pts) (Garbage Collection)

    The diagram below shows a heap with objects. The arrows indicate pointers from one object to another. Arrows with no source (those coming in from the right) indicate a pointer to the object from the stack. For simplicity we assume that every object is the same size, and each rectangle represents one object. If an object is empty, it just means we don't care what's in it.

    The blank diagrams are for you to fill in with the results of doing a garbage collection. Specifically, on the blank diagram in the middle, show the result of mark-and-sweep garbage collection. Make sure to create a free-list (order doesn't matter). Draw an arrow from the "Free List" box to the first free memory block, then from there draw arrows between free memory blocks to make the list.

    On the blank diagram on the right, show the result of compacting (aka stop-and-copy) garbage collection. The diagram represents the half of the heap that is not currently in use; the objects in the heap on the left will be copied into this space. The order in which the objects are moved is not prescribed. Make sure you also show the pointers coming from the stack.

Solution:



11. (10 pts) (Compilation) In class, we gave the following translation schemes (among others) for translating source programs into an intermediate representation (IR).

    [e] : translate expression e to IR; returns pair (IR instruction list, location of value)

    [S] : translate statement S to IR

    $[\text{S}]_{\text{L}}$: translate statement S in context of a loop or switch statement, where L is the target of a break statement

    $[\text{e}]_{\text{Lt,Lf}}$: translate expression e to code that branches to Lt if e is true, or Lf otherwise (the short-circuit evaluation scheme)

    The instructions in our intermediate representation were: x = n; x = y; x = y + z (for any operation +); JUMP L; CJUMP x, L1, L2; and x = LOADIND y.

    Give the translation for the following constructs. (You may use functions `getloc()` and `getlabel()` to get fresh memory locations and fresh instruction labels, respectively.)

    (a) Python's while statement can have the form "`while (`e`)` S `else` S_1". S_1 is executed after the while loop is done (regardless of whether the loop body was executed even once). This is just like having statement S_1 following the while loop, except for one thing: if the loop is terminated by a `break` statement, S_1 is not executed and execution continues at the statement following the while loop. S_1 cannot contain a `break` statement.

        Translate this form of `while`:

        [`while` (e) S `else` S_1] =

Solution:

let L_1, L_2, L_3, L_4 = getloc() in
JUMP L_2
L_1: $[S]_{L4}$
L_2: $[e]_{L1,L3}$
L_3: $[S_1]$
L_4:

(b) Another variation on the while loop is one where the while statement can be followed by n statements: `while` (e) S `else` $S_1 \ldots S_n$. Inside the loop, `break` i immediately jumps to S_i and then to the very end; upon normal termination of the loop (with no break), all the S_i are skipped. There is no plain `break` statement (with no number). The statements S_i cannot contain `break` statements.

In such a while loop, the body (S) has to be compiled in a context in which all these break labels are given. We are asking you to translate a simpler version, in which there are exactly two statements following the loop; S needs to be compiled using a scheme of the form $[\mathrm{S}]_{\mathrm{L1,L2}}$. Give the following translation:

[`while` (e) S `else` S_1; S_2] =

Solution:

```
      let $L_1$, $L_2$, $L_3$, $L_4$, $L_5$ = getloc() in
      JUMP $L_2$
$L_1$: $[S]_{L3,L4}$
$L_2$: $[e]_{L1,L5}$
$L_3$: $[S_1]$
      JUMP $L_5$
$L_4$: $[S_2]$
$L_5$:
```

[`break 1`]$_{L1,L2}$ =

Solution: JUMP L_1

[`break 2`]$_{L1,L2}$ =

Solution: JUMP L_2

12. (10 pts) (Fill in the blanks)

   (a) In a dynamically-typed language, the expression "`False && (1/0)`" will not produce a run-time error, so long as boolean expressions are evaluated using **short-circuit** evaluation.

   (b) When the β rule is used to simplify expressions, rather than the β-value rule, it is called **lazy** evaluation.

   (c) In Java, C++, and some other languages (but not C), you can use the same function name for different functions as long as the number or type of the arguments is different. This is called **overloading**.

   (d) In a dynamically-typed languages, operations like `+` need to be able to check the types of their operands; to allow this, values must be **tagged**.

   (e) In C++, a checked downcast can be applied to an object only if that object's class has at least one **virtual** function.

   (f) In object-oriented languages, dynamic binding of method calls is implemented using a table of function pointers that is called a **v-table**.

   (g) For portability, languages like Java and C# are implemented by translating them to virtual machine code, which can be interpreted; to improve efficiency, the implementations usually translate this VM code to native machine code, in a process called **JIT-** compilation.

   (h) In implementations of functional languages, higher-order functions are represented by a pair containing a function pointer and an environment; this pair is called a **closure**.

   (i) A serious limitation of reference counting is that it cannot recover all cells if they are contained in structures that have **cycles**.

   (j) When a grammar presented to `ocamlyacc` is not LALR(1), `ocamlyacc` reports that the language has **conflicts**.