Final Exam Solution

- Please read all instructions (including these) carefully.
- There are six questions on the exam, all with multiple parts. You have 3 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: ______________________________

SID or SS#: ______________________________

Circle the time of your section: 9:00 10:00 11:00 12:00

1:00 2:00 3:00 4:00

Problem	Max points	Points
1	15	
2	10	
3	10	
4	20	
5	20	
6	25	
TOTAL	100	

1. **Garbage Collection** (15 points)

   (a) In Cool suppose we want every object to support an additional method `hashValue` which returns an integer value such that

   - if we invoke `hashValue` multiple times on the same object, we always get the same integer result;
   - for two distinct objects that are live simultaneously, there is a very small chance that they have the same `hashValue`. Recall that a *live* object is one that is reachable from the set of roots.

   One implementation of `hashValue` is simply to return the address of the object invoking the `hashValue` method.

   For each of the following garbage collection strategies, state whether or not the proposed implementation of `hashValue` works, briefly explain why or why not, and propose an alternative implementation for any cases where the proposed implementation does not work.

   - **reference counting**

     ```
     Does work with reference counting.  With reference counting, an object
     is never moved during its lifetime, so its address does not change.
     Thus multiple invocations of hashValue on the same object always get
     the same integer result.  And since no two live objects can share the same
     address, the second property is also satisfied.
     ```

   - **stop & copy**

     ```
     Doesn't work since after a garbage collection, a live object is always
     copied to a different location.  Thus, multiple invocations of hashValue
     can return different results.

     An alternative is to store a timestamp when an object is created as an
     additional attribute in the object.
     ```

   - **mark & sweep**

     ```
     Does work with mark & sweep, fo the same reason as for reference counting.
     ```

(b) Consider the two tracing garbage collectors discussed in class: stop & copy and mark & sweep. For each of the following situations, please answer which garbage collection strategy is the fastest and briefly justify your answer. Ignore the cost of allocation and of tracing in your answers; focus on the cost of an actual garbage collection.

- In the situation where there is a survival rate (percentage of live data after a collection) of 8%, and it is 20 times more expensive to copy an object than to sweep an object.

```
Mark & sweep is faster.  Suppose the heap contains N objects and
sweeping an object costs M units of time.  The sweeping cost is N*M,
and the copy cost is N*8%*20M, which is 1.6NM.
```

- In the situation where the survival rate is 10% and it is five times more expensive to copy an object than to sweep an object.

```
Stop & copy is faster.  Again with a similar calculation, the sweeping
cost is N*M, and the copy cost is N*10%*5M, which is 0.5NM.
```

- In the situation where the size of live data is close to the size of the heap.

```
Stop & copy can't be used because only 1/2 of the heap can be used for
live data.

Mark & sweep can be used because the whole heap can be used and no
additional space is required to perform a garbage collection (with
pointer reversal).
```

2. **Dataflow Analysis and Optimization** (10 points)

Recall that one of Cool's runtime requirements is to detect dispatch to a `void` pointer. A naive implementation of this check requires a few additional instructions to be executed on every method call, which inevitably degrades the performance of Cool programs. In this problem we explore ways to eliminate the check.

(a) List two simple situations in Cool when the dispatch on void check can be safely eliminated. (No credit for proposing the optimization discussed below!)

```
The two most popular answers were (there are others):

(new A).foo()      // dispatch to a ``newed'' object
self.foo()         // dispatch to self
```

Consider the following excerpt from a Cool program:

```
let x : A <- foo() in {
  x.bar(); (* 1 *)
  ...
  x.baz(); (* 2 *)
}
```

We can eliminate the check for `void` in line `2` if it is known that the value of `x` has not changed since the last time we checked `x` for `void` in line `1`. To check this condition, we need to perform a data flow analysis.

(b) Describe in words: What information does this analysis need to compute for each program point? How is that information used to decide whether tests for dispatch to `void` can be omitted?

```
At each program point, for each variable, we maintain a boolean
property ``On all paths reaching this point, has this variable been
checked for void in a dispatch without a subsequent assignment to the
variable?''  When we encounter an assignment, the property is set to
``false'' for the assigned variable.  When we encounter a dispatch,
the property is set to ``true'' after the dispatch.  All other
statements propagate the information unchanged (at joins we are
conservative and take a logical ``and'' of the values of the
predecessors).

Another acceptable answer was to propose a variant of constant propagation,
where we treat each dispatch as assigning a fresh symbolic constant
to a variable and each real assignment as making the variable non-constant.

Yet another acceptable answer was to propose using liveness analysis
to figure out if between any two dispatches there was a killing assignment.
This is unnatural and tricky (you can't just check one program point,
but have to check every point in between the assignments).  Though
clumsy, it was worth full credit if correctly described.
```

(c) Is this a forward analysis or a backward analysis (circle one)?

```
forward is the expected answer

credit was given for ``backward'' if the previous answer was a correct
application of liveness analysis
```

(d) Can we apply the method from above to eliminate the check if `x` is an attribute rather than a local variable? Why or why not?

```
No.  A method call may modify an attribute, so we can't know after method
calls whether attributes are void or not.  (A yes answer is also acceptable,
if one points out that we either have to be very conservative at method
calls or do interprocedural analysis.)
```

3. **Optimization** (10 points)

For your reference, here is a list of the optimizations discussed in class. We will refer to this list in the following questions, and you can use this list to remind yourself of optimization names.

(a) Algebraic simplification
(b) Constant folding
(c) Common subexpression elimination
(d) Copy propagation
(e) Dead code elimination
(f) Peephole optimizations

We also introduce one new transformation *expression propagation*, where we allow an entire expression e in an assignment $a \leftarrow e$ to be substituted for later uses of a, assuming neither a nor any of the temporaries occuring in e are assigned to again before the use.

Consider the following section of a flow-graph for a bubble-sort algorithm in three-address code:

$$\boxed{\begin{array}{ll}
t_8 & \leftarrow j - 1 \\
t_9 & \leftarrow 4 * t_8 \\
temp & \leftarrow A[t_9] \\
t_{10} & \leftarrow j + 1 \\
t_{11} & \leftarrow t_{10} - 1 \\
t_{12} & \leftarrow 4 * t_{11} \\
t_{13} & \leftarrow A[t_{12}] \\
t_{14} & \leftarrow j - 1 \\
t_{15} & \leftarrow 4 * t_{14} \\
A[t_{15}] & \leftarrow t_{13} \\
t_{16} & \leftarrow j + 1 \\
t_{17} & \leftarrow t_{16} - 1 \\
t_{18} & \leftarrow 4 * t_{17} \\
A[t_{18}] & \leftarrow temp
\end{array}}$$

The following is the evolution of the basic block through several optimization stages:

$$\boxed{\begin{array}{ll} t_8 & \leftarrow j-1 \\ t_9 & \leftarrow 4*t_8 \\ \mathit{temp} & \leftarrow A[t_9] \\ t_{10} & \leftarrow j+1 \\ t_{11} & \leftarrow t_{10}-1 \\ t_{12} & \leftarrow 4*t_{11} \\ t_{13} & \leftarrow A[t_{12}] \\ t_{14} & \leftarrow j-1 \\ t_{15} & \leftarrow 4*t_{14} \\ A[t_{15}] & \leftarrow t_{13} \\ t_{16} & \leftarrow j+1 \\ t_{17} & \leftarrow t_{16}-1 \\ t_{18} & \leftarrow 4*t_{17} \\ A[t_{18}] & \leftarrow \mathit{temp} \end{array}} \overset{1}{\Longrightarrow} \boxed{\begin{array}{ll} t_8 & \leftarrow j-1 \\ t_9 & \leftarrow 4*t_8 \\ \mathit{temp} & \leftarrow A[t_9] \\ t_{10} & \leftarrow j+1 \\ t_{11} & \leftarrow j \\ t_{12} & \leftarrow 4*j \\ t_{13} & \leftarrow A[t_{12}] \\ t_{14} & \leftarrow j-1 \\ t_{15} & \leftarrow 4*t_{14} \\ A[t_{15}] & \leftarrow t_{13} \\ t_{16} & \leftarrow j+1 \\ t_{17} & \leftarrow j \\ t_{18} & \leftarrow 4*j \\ A[t_{18}] & \leftarrow \mathit{temp} \end{array}} \overset{2}{\Longrightarrow} \boxed{\begin{array}{ll} t_8 & \leftarrow j-1 \\ t_9 & \leftarrow 4*t_8 \\ \mathit{temp} & \leftarrow A[t_9] \\ t_{12} & \leftarrow 4*j \\ t_{13} & \leftarrow A[t_{12}] \\ t_{14} & \leftarrow j-1 \\ t_{15} & \leftarrow 4*t_{14} \\ A[t_{15}] & \leftarrow t_{13} \\ t_{18} & \leftarrow 4*j \\ A[t_{18}] & \leftarrow \mathit{temp} \end{array}} \overset{3}{\Longrightarrow} \boxed{\begin{array}{ll} t_8 & \leftarrow j-1 \\ t_9 & \leftarrow 4*t_8 \\ \mathit{temp} & \leftarrow A[t_9] \\ t_{12} & \leftarrow 4*j \\ t_{13} & \leftarrow A[t_{12}] \\ A[t_9] & \leftarrow t_{13} \\ A[t_{12}] & \leftarrow \mathit{temp} \end{array}}$$

For each optimization stage, identify all optimization(s) applied (there can be more than one!). You may refer to the list in the beginning of this problem.

Note: To answer this problem you only had to write down the optimiztaion names in no particular order. The answer below also explains what happened during each optimization.

- **Stage 1:**
  - Expression propagation: $t_{11} = t_{10} - 1$ becomes $t_{11} = j + 1 - 1$, similarly for t_{17}
  - Algebraic Simplification or Constant Folding: $t_{11} = j+1-1$ becomes $t_{11} = j$, similarly for t_{17}
  - Copy Propagation: $t_{12} = 4 * t_{11}$ becomes $t_{12} = 4 * j$, similarly for t_{18}

- **Stage 2:**
  - Dead code elimination: remove $t_{10} = j + 1$, $t_{11} = j$, $t_{16} = j - 1$, $t_{17} = j$

- **Stage 3:**
  - Common subexpression elimination: $t_{14} = j - 1$ becomes $t_{14} = t_8$, and $t_{18} = 4 * j$ becomes $t_{18} = t_{12}$
  - Copy propagation: $t_{15} = 4 * t_{14}$ becomes $t_{15} = 4 * j$
  - Common subexpression elimination: $t_{15} = 4 * j$ becomes $t_{15} = t_9$
  - Copy propagation: $A[t_{15}] = \ldots$ becomes $A[t_9] = \ldots$, and $A[t_{18}] = \ldots$ becomes $A[t_{12}] = \ldots$
  - Dead code elimination: remove $t_{14} = t_8$, $t_{15} = t_9$, and $t_{18} = t_{12}$

4. **Dynamic Scoping** (20 points)

In Cool, there are three kinds of variables: attributes of a class, method parameters, and local variables introduced by `let` and `case`. In particular, method parameters and local variables are only in scope within the body of their definition—this is lexical scoping. In this question, we will add a fourth kind of variable to Cool that supports *dynamic* scoping.

Like ordinary variables, dynamically-scoped variables can be used within the body of their definition, but in addition their scope extends through method calls. For example,

```
foo() : Int {
   let x : Int <- 6 in
      x + y
};
bar() : Int {
   dynamic-let y : Int <- 7 in
      foo()
};
```

In Cool, method `foo` is illegal because `y` is a free variable in its body. However, if we supported dynamic scoping, method `foo` would be fine as long so it was called in a context where a dynamically-scoped variable `y` was defined, such as in method `bar`.

We fix the problem with the example above by modifying the method declaration form. We add a second optional list of formals (with their types) that defines the types of all of the free variables in the method. We rewrite method `foo` as an example:

```
foo() (y:Int) : Int {
  let x : Int <- 6 in
     x + y
};
```

Here is the current type checking rule for method declarations. For simplicity, we have omitted SELF_TYPE, and you may ignore SELF_TYPE in your answers.

$$\frac{\begin{array}{l} M(C,f) = (T_1,\ldots,T_n,T_0) \\ O_C[T_1/x_1]\ldots[T_n/x_n],M,C \vdash e : T_0' \\ T_0' \leq T_0 \end{array}}{O_C,M,C \vdash f(x_1 : T_1,\ldots,x_n : T_n) : T_0 \ \{\ e\ \};} \quad \text{[Method]}$$

(a) Complete the partial rule below to incorporate our new method declaration form and make method `foo` above type check correctly. Note that we have modified the form of the method prototype (contained in the map M) to identify clearly the names and types of the free variables.

$$\frac{\begin{array}{l} M(C,f) = (T_1,\ldots,T_n; y_1 : T_1',\ldots,y_m : T_m'; T_0) \\ O_C[T_1/x_1],\ldots,[T_n/x_n][T_1'/y_1],\ldots,[T_m'/y_m],M,C \vdash e : T_0' \\ T_0' \leq T_0 \end{array}}{O_C,M,C \vdash f(x_1 : T_1,\ldots,x_n : T_n)(y_1 : T_1',\ldots,y_m : T_m') : T_0 \ \{\ e\ \};} \quad \text{[Method]}$$

Consider now another example with `foo` and `bar` written as follows:

```
foo() (y:Int) : Int {
  let x : Int <- 6 in x + y };
bar() : Int {
   dynamic-let y : Bool <- true in foo() };
```

The variable `y` is declared as a `Bool` in method `bar` and as an `Int` in method `foo`. In order to type check this program, we need a type rule for `dynamic-let`, and we need to modify our existing type rules. We add a new type environment called `D` (defined just like `O`), to hold the types of just the variables introduced by `dynamic-let`. Here is the new type rule for `dynamic-let` with initialization (the rule without initialization is similar):

$$\frac{\begin{array}{l} D, O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \\ D[T_0/x], O[T_0/x], M, C \vdash e_2 : T_2 \end{array}}{D, O, M, C \vdash \text{dynamic-let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad \text{[Dynamic-Let-Init]}$$

(b) Note that we add the typing for `x` to both `D` and `O`. For the following incorrect variation of the rule, give a simple example that fails to type check because we do not add the type assumption for `x` to the environment `O`.

$$\frac{\begin{array}{l} D, O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \\ D[T_0/x], O, M, C \vdash e_2 : T_2 \end{array}}{D, O, M, C \vdash \text{dynamic-let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad \text{[Dynamic-Let-Init]}$$

```
foo() : Int {
   dynamic-let y : Int <- 7 in
      y
};
```

```
If we only added y's type to D, we would not be able to look up y
in the body of the dynamic-let. Our unchanged rule for variable
references looks in O, not D.
```

Here is the current rule for method dispatch:

$$\frac{\begin{array}{l} O,M,C \vdash e_0 : T_0 \\ O,M,C \vdash e_1 : T_1 \\ \vdots \\ O,M,C \vdash e_n : T_n \\ M(T_0,f) = (T'_1,\ldots,T'_n,T'_{n+1}) \\ T_i \leq T'_i \quad 1 \leq i \leq n \end{array}}{O,M,C \vdash e_0.f(e_1,\ldots,e_n) : T'_{n+1}} \quad \text{[Dispatch]}$$

(c) Modify this rule to incorporate D. Make sure you use the new form of the M mapping.

$$\frac{\begin{array}{l} D,O,M,C \vdash e_0 : T_0 \\ D,O,M,C \vdash e_1 : T_1 \\ \vdots \\ D,O,M,C \vdash e_n : T_n \\ M(T_0,f) = (T'_1,\ldots,T'_n,y_1 : S'_1,\ldots,y_m : S'_m,T'_0) \\ T_i \leq T'_i \quad 1 \leq i \leq n \\ D(y_j) = S'_j \quad 1 \leq j \leq m \end{array}}{D,O,M,C \vdash e_0.f(e_1,\ldots,e_n) : T'_0} \quad \text{[Dispatch]}$$

```
Common mistakes:

Adding D to each line, but doing nothing else.
Changing the method invocation to explicitly pass the dynamic
variables.
Treating dynamic variables as expressions to recursively typecheck
instead of variable names to look up directly in D.

Note:

Here is a subtle point. The types of the dynamic variables declared in
dynamic-let (and added to D) must be invariant (equal to) the declared
dynamic types for the method. Since dynamic variables are
call-by-reference, contravariant subtyping (as for normal procedure
parameters) is not sound. We did not take off any points for students
who allowed contravariant subtyping of dynamic variable method
parameters.
```

(d) After adding these new type checking rules to our Cool compiler, we're satisfied that it will correctly type check programs that use `dynamic-let`. Now, it's time to generate code. In plain English, discuss one method for implementing `dynamic-let`. Be sure to describe where dynamic variables are stored, how they are looked up and referenced, and how they are managed across method invocations.

```
Here are several possible solutions:

1. Keep a runtime data structure like the SymbolTable (used in PA4 and
```

PA5) to store a mapping between the name of a dynamic variable and its value. For each dynamic-let, enter scope and insert the dynamic variable into the table. Whenever you look up the dynamic variable at runtime, find its name in your static data area and look up its value in the table. After the dynamic-let body is finished, exit scope.

2. A different version of the above idea: You can determine at compile-time what the names of all dynamic variables are in the entire program. Assign each of these a unique id. Instead of using a SymbolTable (mapping names to values), use an IntegerTable (mapping integer ids to values). It should be faster than a name lookup.

3. Another alternative: Don't use a SymbolTable or IntegerTable, but just a normal one-level hashtable. When you enter a dynamic-let, you save the old value of that variable (it's ok if it's void) in a temporary variable on your stack, and then write the new value into the table. When you exit a dynamic-let, read the value from the temporary slot in your stack and write it back into the table. This technique is known as shallow binding.

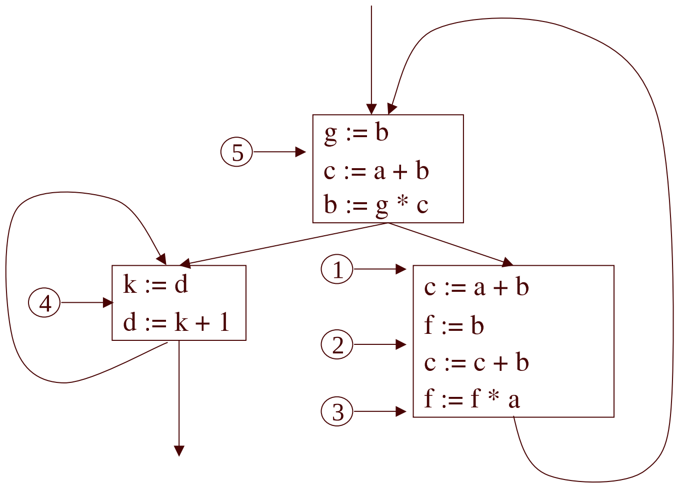
4. Put dynamic variables on the stack, in temporary slots, like let variables. On a method invocation, pass the formal arguments as before, but in addition, pass pointers to the temporary slots that the dynamic variables live in (this is call by reference). When you access the dynamic variable, you indirect through this pointer to access the value in its home activation record.

5. A different version of the above idea: Instead of passing pointer to the dynamic variables' temporary slots, pass the values of the dynamic variables. When you access the dynamic variables, you access them in the current activation record. When you return from a method, make sure to copy the dynamic variable values from the callee's activation record back into the caller's activation record. This is known as call-by-copy-in-copy-out.

6. A variant of the above that is difficult to get right is to use the old fp on the stack to gain access to the caller's activation record where the dynamic variables are stored. However, since dynamic-let may nest arbitrarily deep, it is not known at compile-time how many frames you need to traverse to find a given dynamic variable. Run-time bookkeeping for this solution is complicated. It involves keeping track of the number of frames to traverse for each dynamic variable separately and passing this information through every method invocation.

5. **Register Allocation** (20 points)

This problem is concerned with register allocation and instruction scheduling for the program whose control-flow graph is shown below.



(a) Write down the set of live temporaries at the points 1–5 in the program. Assume that there are no live variables on exit. (It is a good idea to compute the live variables at all points in the program, even though this part does not require it.)

Live at 1 = {a, b, d}

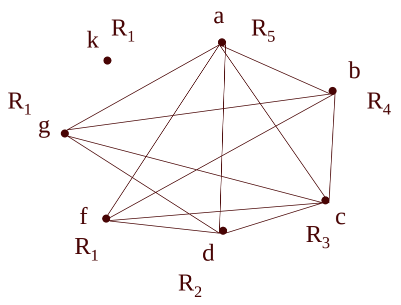
Live at 2 = {a, b, c, d, f}

Live at 3 = {a, b, d}

Live at 4 = {k}

Live at 5 = {a, b, d, g}

(b) Fill in the edges of the register interference graph:



Solution: See the graph.

(c) Show a coloring for the register interference graph using the minimal number of colors. Use color names such as R_1, R_2, ... Write the color next to each node in the graph.

Solution: See the graph.

(d) Consider now that we swap the instructions `f := b` and `c := c + b` so that the basic block containing them is as follows:

```
c := a + b
c := c + b
f := b
f := f * a
```

What is now the set of live registers at point 2 in the program (right before the instruction `f := b`? And what is the minimum number of colors necessary now to color the interference graph? Explain briefly why the number of colors changes.

Live at 2 = {a, b, d}

Colors = still 5. Even though there is no interference anymore between `c` and `f`, the set of nodes {a, b, c, d, g} continues to form a 5-clique. (The question was not intentionally misleading; we got it wrong the first time.)

(e) The swap from point 5d above is legal. But not all swaps are legal. Consider the sequence of two arithmetic instructions:

```
x := y op z
u := t op v
```

Explain when these two instructions can be swapped, according to the rules of instruction scheduling? Complete the partial answer below (where $x \not\equiv t$ means that x and t are not the same temporary).

Solution: The instructions can be swapped if and only if:

$x \not\equiv t$	read-after-write
$x \not\equiv v$	read-after-write
$x \not\equiv u$	write-after-write
$u \not\equiv y$	write-after-read
$u \not\equiv z$	write-after-read

(f) Consider again the original control-flow graph and assume that we have 3 available registers for allocation. We will have to spill some temporaries to memory. Explain why **it is not** useful to spill temporary `k` to memory.

Solution: Register `k` does not interfere with any other registers. Thus spilling it will not reduce the number of colors necessary to color the graph.

(g) While performing register allocation it is possible to eliminate some move instructions (instructions of the form `x := y`). If the register allocator assigns the same register to temporaries `x` and `y`, then the move instruction can be eliminated.

A simple way to force the register allocator to place two temporaries `x` and `y` in the same register is to combine the nodes in the register interference graph for `x` and `y` into one node (the combined node has all the edges of both original nodes). When is it legal to combine two nodes in a register interference graph?

Solution: The nodes can be merged if there is no edge between them in the graph.

6. **Lambda Calculus** (25 points)

Here is the grammar for the untyped λ calculus:

$$E ::= x \mid \lambda x.E \mid E\ E$$

(a) Show that this grammar is ambigious.

`The string x y z can be parsed as both (x y) z and x (y z).`

(b) On the next page is the DFA of LR(1) items for an LALR(1) parser for this grammar. The grammar has been augmented with a new start symbol S', and the states have been numbered. This DFA for recognizing viable prefixes has exactly four conflicts. For each of the conflicts, list what state it appears in, what kind of conflict it is (shift-reduce or reduce-reduce), and what lookahead token exhibits the conflict. Note that we use the symbol $\bullet$ instead of $\cdot$ as the marker in LR items.

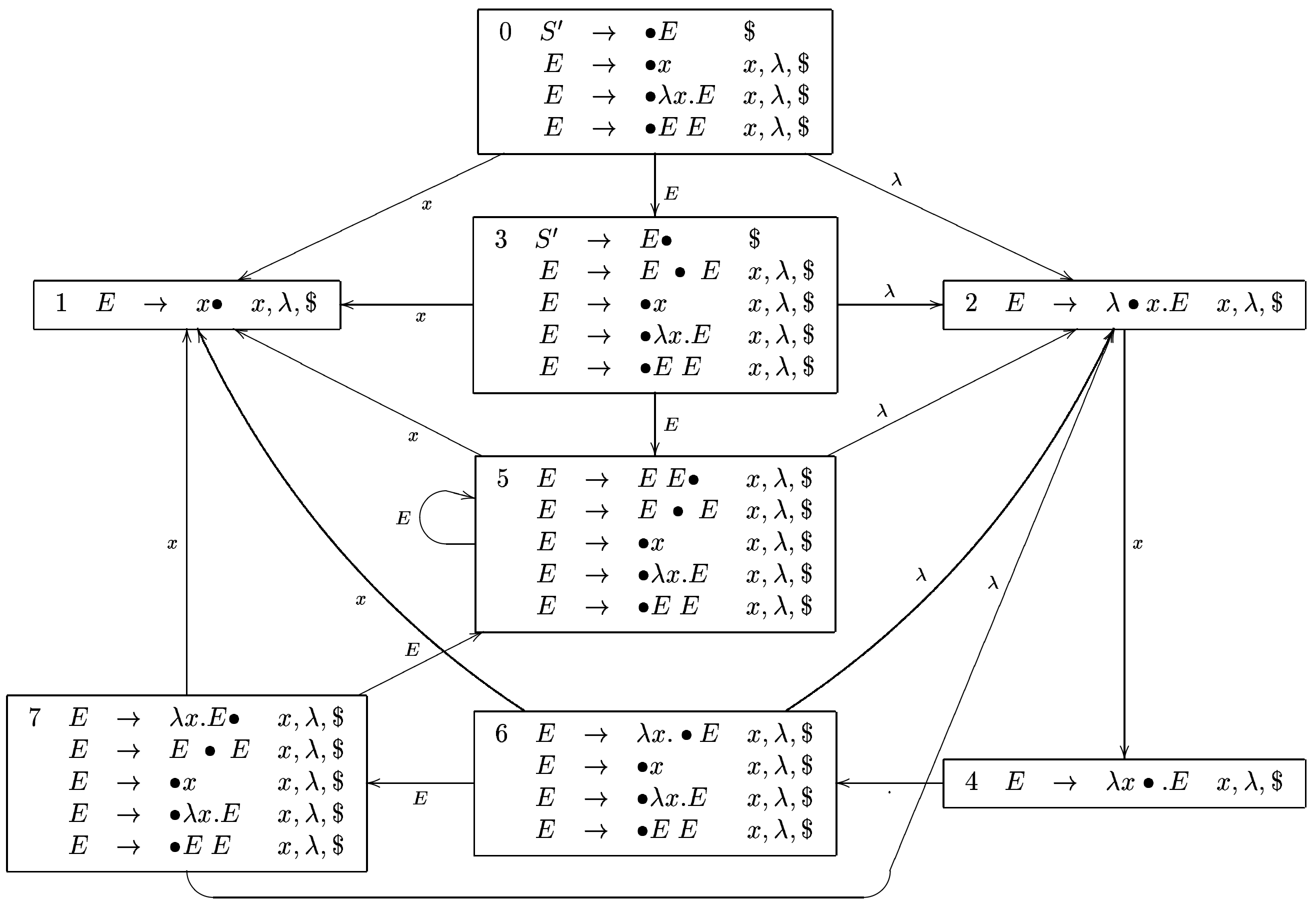
State 5 contains shift/reduce conflicts on x and λ. State 7 contains shift/reduce conflicts on x and λ.

(c) There are two rules for disambiguating λ calculus terms:

- The body of the function corresponding to "$\lambda x.$" extends from the "." as far to the right as possible. For example, $\lambda x.\lambda y.x\ y$ parses as $\lambda x.(\lambda y.(x\ y))$.
- Application associates to the left. For example, $x\ y\ z$ parses as $(x\ y)\ z$.

For each conflict in part (b), state how the conflict should be resolved to make the parser obey these rules. Phrase your answers as "the conflict in state X on input Y should be resolved by shifting/reducing by production W".

Both conflicts in state 7 should be resolved as shifts. Both conflicts in state 5 should be resolved as reduces by production $E \to E\ E$.



(d) In the λ-calculus, can evaluation of an expression using call-by-value differ from evaluation using call-by-name? Explain.

```
Yes.  An expression evaluated under call-by-value may not terminate
when evaluating under call-by-name does.  Note that it is not true
that call-by-name evaluation always terminates.
```

(e) Evaluate the following expression under call-by-name. Write each step of β-reduction.

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

Solution:

$$\begin{aligned}((\lambda x.\lambda y.x\ y)\ (\lambda x.\lambda y.x))\ (\lambda x.x) &\rightarrow_\beta (\lambda y.(\lambda x.\lambda y.x)\ y)\ (\lambda x.x)\\ &\rightarrow_\beta (\lambda x.\lambda y.x)\ (\lambda x.x)\\ &\rightarrow_\beta \lambda y.(\lambda x.x)\end{aligned}$$

(f) For each of the following λ expressions, circle its principal type:

i. $\lambda x.\lambda y.y\ x$

(a) $\alpha \to \beta \to \alpha$ (b) $\beta \to \alpha \to \alpha$

(c) $\alpha \to (\alpha \to \beta) \to \beta$ (d) $(\alpha \to \beta) \to \alpha \to \beta$

Solution: c

ii. $\lambda x.\lambda y.(x\ y)\ y$

(a) $\alpha \to (\alpha \to \beta) \to \alpha$ (b) $(\alpha \to \alpha \to \beta) \to \alpha \to \beta$

(c) $(\alpha \to \beta) \to ((\alpha \to \beta) \to \alpha) \to \beta$ (d) $\alpha \to (\alpha \to \beta) \to \beta$

Solution: b

iii. $\lambda x.\lambda y.x\ (y\ x)$

(a) $(\alpha \to \beta) \to ((\alpha \to \beta) \to \alpha) \to \beta$ (b) $\alpha \to (\alpha \to \alpha) \to (\alpha \to \alpha)$

(c) $(\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha$ (d) $\alpha \to (\alpha \to \beta) \to \beta$

Solution: a

iv. $\lambda x.\lambda y.y\ (y\ x)$

(a) $\alpha \to (\alpha \to \alpha) \to \alpha$ (b) $(\alpha \to \alpha) \to \alpha \to \alpha$

(c) $\alpha \to (\alpha \to \beta) \to \alpha$ (d) $\alpha \to (\alpha \to \alpha) \to (\alpha \to \alpha)$

Solution: a

(g) Write down a λ expression that has type

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

Solution: $\lambda x.\lambda y.\lambda z.x\ (y\ z)$