

# Masters examination

Spring 2002

Department of Computer Science

The University of Arizona

This examination consists of nine problems, from which you will pick six. The questions are in three areas:

- (1) Theory (CSc 473, 573, and 545),
- (2) Systems (CSc 452, 552, and 576), and
- (3) Languages (CSc 453, 553, and 520).

You are to answer *two questions* from *each area*. If more than two questions are attempted in any area, the two highest scores will be used.

You have *two hours* to complete the examination. Books or notes are not permitted during the exam. At the end of the examination, submit just your solutions in the envelope provided. Do not include scrap paper.

Write your answers on the tablet paper provided separately. Start the answer to each question on a new sheet of paper. On *each page* that you hand in, write the *problem number* in the upper-left corner and the *last four digits* of your social security number in the upper-right corner. Results will be posted using the four-digit numbers.

Some problems will ask you to write an algorithm or a program. Unless directed otherwise, use informal pseudocode. You may invent constructs to help present your solution, provided the meaning of such constructs is clear and they do not make the problem trivial. For example, when describing code that iterates over the elements of a set, you might find it helpful to use a construct such as

**for**  $x \in S$  **do** statement

where  $S$  is a set.

## Solutions

## Theory 1      CSc 473 (Automata, Grammars, and Languages)

The following two questions concern regular languages.

- (a) (4 points) Let  $L$  be the language consisting of all strings  $x \in \{\mathbf{a}, \mathbf{b}\}^*$  that satisfy the following property. If  $\mathbf{aab}$  occurs as a substring of  $x$ , then it must eventually be followed by the substring  $\mathbf{ba}$ . Show  $L$  is regular by constructing a finite automaton that accepts  $L$ .
- (b) (6 points) For any language  $L$  over alphabet  $\Sigma$ , let  $\text{Pref}(L)$  be the language consisting of all prefixes of all strings in  $L$ . More formally,

$$\text{Pref}(L) := \left\{ x \in \Sigma^* : xy \in L \text{ for some } y \in \Sigma^* \right\}.$$

If  $L$  is regular, does that imply  $\text{Pref}(L)$  is regular? Prove your answer.

### Solution

- (a) Automaton to be provided later.
- (b) Yes, if  $L$  is regular, then  $\text{Pref}(L)$  is regular. We show this as follows. Since  $L$  is regular, there is a DFA  $M$  that accepts  $L$ . Given  $M$ , we construct a new DFA  $M'$  that accepts  $\text{Pref}(L)$ . The state set, transitions, and start state of  $M'$  are identical to  $M$ . The final states of  $M'$  are identified as follows. For every state  $s$  in  $M$  with the property that there is a path from  $s$  to an accepting state, make  $s$  a final state in  $M'$ . Clearly  $M'$  now accepts exactly those strings that are prefixes of strings accepted by  $M$ .

## Theory 2      CSc 573 (Theory of Computation)

The *Bipartition Problem* is the following decision problem, which is known to be NP-complete. The input is a collection of items numbered  $1, 2, \dots, n$ . Each item  $i$  has an associated size  $x_i \geq 0$ . The output is “yes” if and only if there is a subset  $S \subseteq \{1, \dots, n\}$  of the items such that

$$\sum_{i \in S} x_i = \sum_{j \notin S} x_j.$$

The *Knapsack Problem* is another decision problem. Here the input is a collection of items, each with an associated weight  $w_i \geq 0$  and value  $v_i \geq 0$ . In addition to the items, the input specifies a weight limit  $W$  and a value goal  $V$ . The output is “yes” if and only if there is a subset  $S$  of the items such that the weight of  $S$  does not exceed the limit,

$$\sum_{i \in S} w_i \leq W,$$

and the value of  $S$  meets the goal,

$$\sum_{i \in S} v_i \geq V.$$

- (a) (1 point) What are the steps in proving that problem  $X$  is NP-complete, given that problem  $Y$  is known to be NP-complete?
- (b) (9 points) Prove that the Knapsack Problem is NP-complete, given that the Bipartition Problem is known to be NP-complete.

### Solution

- (a) To prove that  $X$  is NP-complete given that  $Y$  is NP-complete, you need to (1) show that  $X$  is in NP, (2) give a mapping  $f$  from instances of  $Y$  to instances of  $X$  such that  $f(y)$  is a “yes”-instance iff  $y$  is a “yes”-instance, and (3) show that  $f$  can be computed in polynomial time.
- (b) Note that Knapsack is in NP, since after guessing an item subset  $S$  we can check whether  $S$  satisfies the weight limit and meets the value goal in polynomial time.

To reduce Bipartition to Knapsack, we use the same set of items for Knapsack as are in Bipartition. We assign item weights and values by  $w_i = v_i = x_i$ . For the weight limit and value goal we take

$$W = V = \frac{1}{2} \sum_{1 \leq i \leq n} x_i.$$

Then an item subset  $S$  satisfies the weight limit and value goal requirements iff the total size of  $S$  is exactly half the total size of all the items. But this is equivalent to having the total size of the items in  $S$  be equal to the total size of the items not in  $S$ . So the constructed instance of Knapsack is “yes” iff the original instance of Bipartition is “yes”. Finally, note that the construction can be carried out in polynomial time.

### Theory 3      CSc 545 (Design and Analysis of Algorithms)

Let  $G = (V, E)$  be a directed graph in which *vertices* are weighted. Specifically, every vertex  $v_i \in V$  has an associated weight  $w_i \geq 0$ . We define the *cost* of a path  $P$  in  $G$  to be the sum of the weights of the vertices along  $P$ .

Given two vertices  $s$  and  $t$  in  $V$ , describe an algorithm that finds a path with minimum cost from  $s$  to  $t$  in  $O(m + n \log n)$  time, where  $n$  and  $m$  are respectively the number of vertices and edges in  $G$ .

Be sure to argue that your algorithm solves the problem and satisfies the required time bound.

### Solution

Given  $G$ , we construct a new graph  $G' = (V', E')$  whose *edges* are weighted. For each vertex  $v_i$  in  $V$ , we create two vertices  $x_i$  and  $y_i$  in  $V'$ , and we add an edge  $(x_i, y_i)$  in  $E'$  with weight  $w_i$ . For every edge  $(v_i, v_j)$  in  $E$  we also add an edge  $(y_i, x_j)$  to  $E'$  with weight 0.

With this construction, every path  $P$  in  $G$  from  $v_i$  to  $v_j$  corresponds to a path  $P'$  in  $G'$  from  $x_i$  to  $y_j$ . Moreover, the sum of the weights of the vertices in  $P$  equals the sum of weights of the edges in  $P'$ .

Thus to find a least cost path from  $s$  to  $t$  in  $G$  we can find a shortest path between the corresponding vertices in  $G'$ . All edge weights in  $G'$  are positive, so Dijkstra's algorithm applies. The running time of this algorithm is  $O(|E'| + |V'| \log |V'|)$ . Since  $|E'| = m + n$  and  $|V'| = 2n$ , this is  $O(m + n \log n)$ .

## Systems 1      CSc 452 (Principles of Operating Systems)

Give short answers to the following questions about page-replacement algorithms. Recall that the goal of a page-replacement algorithm is to minimize the number of page faults for a given string of page references and a given memory size.

- (a) (3 points) Describe the *optimal-replacement* algorithm. Why is it not used by operating systems?
- (b) (3 points) Describe the *least-recently-used* algorithm (LRU). Why is it not used by operating systems?
- (c) (4 points) Describe a scenario (specifically, page reference strings and the memory size) in which the *most-recently-used* algorithm (MRU) results in fewer page faults than LRU.

### Solution

- (a) The optimal replacement algorithm replaces the page whose next access is farthest in the future. This algorithm isn't used in practice because it requires knowing the future.
- (b) The LRU algorithm replaces the page that hasn't been accessed for the longest time. The assumption is that past behavior is indicative of future behavior, so that recently accessed pages are likely to be accessed before the least-recently-used page. This algorithm isn't used in practice because it has high overhead. An access time must be maintained for each page, and must be updated on each access to the page. This would be extremely slow to do in software, and expensive to do in hardware. Also, finding the LRU page will be slow and expensive. In practice, approximations of LRU such as the clock algorithm perform nearly as well with much less overhead.
- (c) Consider a memory that can hold  $n$  pages, and a program that cycles through  $n + 1$  pages in a loop.

With LRU, the first access to each page results in a page fault, because the new page brought in replaces the next page that will be accessed. Every loop iteration incurs  $n + 1$  page faults.

With MRU, there will only be one page fault per loop iteration, as the page that doesn't fit replaces the most-recently-used page (which is the previous page in the sequence). Thus the number of faults with MRU is a factor  $n + 1$  smaller than with LRU.

## Systems 2      CSc 552 (Advanced Operating Systems)

Client-side caches are commonly used to improve the performance of distributed file systems. A design issue that arises when implementing a client-side cache is the choice of an algorithm for propagating writes back to the server. Three commonly-used algorithms are:

- (1) write-through,
- (2) write-back (i.e. delayed write), and
- (3) write-back-on-close.

Define each of these, and discuss how each affects a distributed file system with respect to:

- (i) performance,
- (ii) reliability, and
- (iii) consistency.

### Solution

First the *definitions*:

- *Write-through* propagates a modification to the server synchronously with applying it to the cache.
- *Write-back* propagates a modification to the server by the cache-replacement policy.
- *Write-back-on-close* propagates all modifications to a file to the server when the file is closed.

With respect to *performance*:

- *Write-through* has the worst performance as all writes generate network traffic and complete at the speed of the server.
- *Write-back-on-close* only writes a block to the server once, even if it was modified multiple times while the file was open. This results in fewer server writes than write-through and better performance.
- *Write-back* provides the highest performance as it allows the most data to be overwritten and deleted without being written to the server.

When a client crashes, the contents of its cache may be lost. With respect to *reliability*:

- *Write-through* is best, as the server is always up-to-date.
- *Write-back-on-close* has the drawback that a client crash while a file is open may cause the file to be corrupted.
- *Write-back* has the effect that a client loses any modified data it has cached. This is no bound on the age of the data lost, or to which files they may belong.

All three algorithms need a mechanism (such as tokens, leases, or callbacks) for ensuring that clients do not read stale data from their caches. With respect to *consistency*:

- *Write-through* keeps the server up-to-date, making it easy for the server to determine if a client has out-of-date data cached. Clients can simply poll to refresh their caches.
- *Write-back-on-close* has the property that while a file is open, the server may not have the most up-to-date data. This means that inconsistencies are possible, unless the server does

a callback to the writing client, invalidates all caches when a file is opened for writing, and so on. Client polling can lead to inconsistencies.

- *Write-back* has the drawback that the server may not have an up-to-date version of any file. Client polling will lead to many inconsistencies. Consistency can only be provided if the server keeps track of the complete cache state of all clients, and uses tokens or callbacks to ensure consistency. This increases the load on and complexity of the server.

## Systems 3      CSc 576 (Computer Architecture)

*Scoreboarding* and *Tomasulo's algorithm* are two commonly-used algorithms for performing dynamic instruction scheduling in processors that support out-of-order execution. Explain how the two algorithms differ in the following respects:

- (a) (3 points) handling of WAW and WAR hazards,
- (b) (4 points) delay between computation of a value by an instruction and start of execution of instructions waiting for the value, and
- (c) (3 points) implementation of load-store disambiguation.

### Solution

- (a) Through register renaming, Tomasulo's algorithm effectively eliminates these hazards. Each instruction is assigned a unique destination register to which it writes its result by the register renaming mechanism, thereby eliminating these hazards. In case of scoreboarding, the execution of an instruction is delayed until the hazards are resolved.
- (b) In Tomasulo's algorithm, the result is simultaneously broadcast to all waiting instructions at the same time that it is written to the register. Therefore, waiting instructions can begin execution right away. Scoreboarding requires that first the value be written to the register and then in the following cycle a waiting instruction can read it from the register before proceeding for execution. Thus, the register file acts as a bottleneck to performance in the scoreboarding algorithm while by using Tomasulo's algorithm this bottleneck is avoided.
- (c) Tomasulo's algorithm treats Load and Store instructions like other types of instructions (e.g., integer ALU or FP ALU) by having dedicated Load and Store queues (or units). These units can very easily implement load-store disambiguation by examining the referenced memory addresses. It is not possible to similarly implement load-store disambiguation with scoreboarding.

## Languages 1      CSc 453 (Compilers and System Software)

Consider the following context-free grammar for recognizing regular expressions. In the grammar,  $A$  is the start variable, '+' is the terminal corresponding to the alternation operator, and '\*' is the terminal corresponding to the Kleene closure operator.

$$\begin{aligned} A &\rightarrow B + A \\ A &\rightarrow B \\ B &\rightarrow C B \\ B &\rightarrow C \\ C &\rightarrow D * \\ C &\rightarrow D \\ D &\rightarrow ( A ) \\ D &\rightarrow \text{letter} \end{aligned}$$

For this grammar, describe the state of an SLR(1) parser immediately after shifting the last character of the string

$a*b($

Specifically, list the items in the current state and the contents of the stack.

### Solution

After the last shift, the state is,

$$\begin{aligned} D &\rightarrow ( \bullet A ) \\ A &\rightarrow \bullet B + A \\ A &\rightarrow \bullet B \\ B &\rightarrow \bullet C B \\ B &\rightarrow \bullet C \\ C &\rightarrow \bullet D * \\ C &\rightarrow \bullet D \\ D &\rightarrow \bullet ( A ) \\ D &\rightarrow \bullet \text{letter} \end{aligned}$$

and the stack is,

$CC($

## Languages 2      CSc 553 (Principles of Compilation)

To produce high-quality code, it is necessary to recognize special cases. For example, we would like to avoid generating code for an addition of zero to an operand. One way to do this is to perform a peephole optimization phase that looks for special cases and replaces them with improved code. The collection of special cases that define a peephole optimizer can be represented as a list of

**pattern** → **replacement**

pairs. If a code sequence matching the **pattern** is found, it is replaced with the **replacement** sequence.

- (a) (5 points) Give at least three examples of special cases (i.e. a **pattern**) that a peephole optimizer might look for, and explain what the peephole optimizer would do to optimize the code (i.e. the **replacement**). You can choose if you like from the following list:

constant folding, strength reduction, null sequences, combining operations, algebraic laws, addressing mode operations.

- (b) (5 points) When there are a large number of **pattern** → **replacement** pairs, the peephole optimization phase may take too much time to optimize large-scale codes. What does a peephole optimizer do to speed up the optimization phase?

### Solution

- (a) Here are some types of pairs.

- Constant folding: evaluate constant expressions in advance.
- Strength reduction: replace slow operations with faster equivalents.
- Null sequences: delete useless operations.
- Combining operations: replace several operations with an equivalent one.
- Algebraic laws: use algebraic laws to simplify or reorder instructions.
- Addressing mode operations: use addressing modes to simplify the code.

- (b) To make pattern-matching faster, the operator-operand combinations in the instructions are hashed to identify applicable patterns. Also, the size of a peephole window is normally limited to two or three instructions.

## Languages 3      CSc 520 (Principles of Programming Languages)

Parameter-passing mechanisms in programming languages differ in regard to the time at which arguments (actuals) to subroutine (procedure or function) calls are evaluated. In languages using call-by-value, arguments are evaluated at the point of call, before executing any part of the subroutine body. In languages using call-by-name or various forms of lazy evaluation, arguments are not evaluated until the corresponding formal is referenced in the subroutine body.

Consider a language **Sloth** in which the time of evaluation of each argument is completely under the programmer's control. Assume **Sloth** has a syntax and semantics similar to **C++**. For simplicity, assume that variables can refer only to integers (there are no arrays or records, and no other primitive data types). Unlike **C++**, however, **Sloth** does not have call-by-value and the ability to pass references as values at call time: actual/formal association is controlled by the **in** and **out** commands described below. The operator **&** does not appear in **Sloth** parameter lists: only the types of each parameter are given in a subroutine definition.

As indicated, **Sloth** has two new statements (commands), described below. These commands can appear only in subroutine bodies. Below *X* refers to any identifier.

- **in X**: This command causes the argument (actual) corresponding to the parameter (formal) *X* to be evaluated *in the environment of the caller*. Any side-effects resulting from such an evaluation are incurred at this point. The value resulting from this evaluation becomes the *r*-value of *X*.
- **out X**: This command updates the location given by the *l*-value of the argument (actual) corresponding to the parameter (formal) *X*. The designated location is updated with the current *r*-value of the parameter (formal) *X*. Flow of control is not affected.

If a parameter *X* is referenced in a subroutine body before **in X** has been executed, it is a run-time error. If *X* is not a parameter, it is a syntax error.

- (a) (2 points) In **Sloth**, complete the definition of the following procedure, which when called will swap the contents of its two variable arguments:

```
void function swap( int X , int Y );
```

- (b) (8 points) Describe how to implement the parameter passing mechanism of **Sloth**. In your answer, address the following points:

1. For each argument, what is transmitted to the callee's activation at call time?
2. What code must be generated for each **in X**?
3. What code must be generated for each **out X**?
4. What code must be generated when parameter *X* is referenced in the subroutine body?
5. What code must be generated when parameter *X* is the target of an assignment in the subroutine body?

## Solution

(a) Sloth is flexible enough to simulate call-by-copy-in/copy-out:

```
void function swap ( int X , int Y )
{  int T;
   in X; in Y;
   T = X;  X = Y;  Y = T;
   out X; out Y;
}
```

(b) The key implementation technique is to use a *thunk* for each argument.

1. At the time of each call, a thunk is created for each actual; that thunk (or a pointer to it) is transmitted to the callee. The environment enclosed in the thunk will be the environment of the caller. When called, the thunk will evaluate the argument expression in the caller's environment, and return an *l*-value. If the actual argument is a simple identifier, the thunk will return its *l*-value (part of the caller's environment). If the actual argument is an expression, the thunk will return the *l*-value of a temporary location in the caller's activation that contains the appropriate *r*-value.
2. Each `in X` results in a call to the thunk associated with *X*. This call returns an *l*-value that is dereferenced to obtain an *r*-value. The resulting *r*-value is assigned to a memory cell in the callee's activation record associated with formal *X*.
3. Each `out X` results in a call to the thunk associated with *X*. This call returns an *l*-value, which is then updated with the contents of the local memory cell associated with *X*.
4. A reference to *X* in the body is simply a reference to the *r*-value of the local memory cell associated with *X*.
5. An assignment to *X* in the body is simply an assignment to the *l*-value of the local memory cell associated with *X*.