CS 241E Final Review

Fall 2019

1 Closures

Consider the partially-implemented Lacs program below.

```
def main(a:Int,b:Int):Int = {
    var pair:(Int)=>Int;
    pair = newPair(a,b);
    pair(2);
    pair(0) - pair(1)
}

// Creates a pair of integers.
// This is represented as a closure which takes one argument: an integer
// representing a choice of method to perform on the pair.
// The following methods are available:
// 0    (getFirst): Return the first integer.
// 1   (getSecond): Return the second integer.
// 2 (compareSwap): If the first integer is smaller than the second integer,
//                  swap them. Return 1 if a swap occurred and 0 otherwise.
def newPair(a:Int, b:Int):(Int)=>Int = {
    ???
    def    getFirst():Int = { ??? }
    def   getSecond():Int = { ??? }
    def compareSwap():Int = { ??? }
    def selectMethod(m:Int):Int = {
        if(m==0) {
            getFirst()
        } else {
            if(m==1) {
                getSecond()
            } else {
                if(m==2) {
                    compareSwap()
                } else {
                    1/0
                }
            }
        }
    }
    ???
}
```

1. What common mathematical function does `main` implement? (Hint: it's not plain subtraction.)
2. Complete the implementation of `newPair` by filling in the question marks.
3. For each of the following procedures, state whether its frame and parameter chunk are stored on the stack or the heap, and explain why.
   - `main`
   - `newPair`
   - `selectMethod`
   - `compareSwap`
4. When the line `pair = newPair(a,b)` executes, the memory address of a chunk gets stored in the variable `pair`. What does this chunk contain?
5. When the closure call `pair(2)` happens, which procedure do we actually jump to? How does `main` know which procedure to jump to?
6. When the closure call `pair(2)` happens, which procedure's frame does `main` pass in as a static link? How does `main` know to pass in this particular static link?
7. When `compareSwap` swaps the two elements of the pair, it modifies variables in some chunk of memory.
   - Why was this chunk created? Is it just a chunk to store closure information, or is it the frame or parameter chunk of a procedure?
   - When was this chunk created? What line of `main` results in this chunk being allocated?
   - Explain why this chunk needs to be stored on the heap rather than the stack.
8. Draw diagrams of the stack and heap after the line `pair = newPair(a,b)` has been executed.
   - Assume the "simple heap allocator" is being used (i.e., don't worry about garbage collection).
   - For each chunk on the stack and heap, describe its purpose (e.g., closure chunk for procedure X, parameter chunk for procedure Y, frame for procedure Z).
   - List the variables in each chunk, not including temporary variables used for things like binary operations and if statements. You should include static links. You can write "standard frame variables" as shorthand for the dynamic link, saved program counter, and parameter pointer, which appear in the frame of every procedure. You should know what values are stored in these variables and why saving these values is important.
   - For integer variables, you do not need to list the value.
   - For pointer variables, indicate which chunk the variable points to; if the pointer variable is the static link of a procedure with no outer procedure, indicate that it points to address zero.
   - A chunk representing a closure contains two variables: `closureCode` containing the address of the code to be executed, and `closureEnvironment` containing the environment. For `closureCode`, indicate which procedure's address is stored in the variable. For `closureEnvironment`, indicate what the variable points to.
   - You do not need to include the chunk header information (size and number of pointer variables).
   - You do not need to include the parameter chunk or frame for the special "start" procedure automatically added by the compiler; assume the first two chunks allocated are the parameter chunk of `main` followed by the frame of `main`.

2 Tail Calls

Consider the following program:

```
def main(a:Int,b:Int):Int = {
    tailCallOne(a,b)
}
def tailCallOne(a:Int,b:Int):Int = {
    tailCallTwo(a,b)
}
def tailCallTwo(a:Int,b:Int):Int = {
    a+b
}
```

Suppose `main(1,2)` is executed.

1. Assuming *no* tail call optimization is performed, which procedure frames or parameter chunks are stored on the stack at the program point right before `tailCallTwo` executes the line `a+b`?
2. Assuming that tail call optimization *is* performed, which procedure frames or parameter chunks are stored on the stack at the program point right before `tailCallTwo` executes the line `a+b`?
3. Consider the following modified program. Now `tailCallOne` is a nested procedure of `main` with no parameters, and it accesses the parameters of `main`.

   ```
   def main(a:Int,b:Int):Int = {
       def tailCallOne():Int = {
           tailCallTwo(a,b)
       }
       tailCallOne()
   }
   def tailCallTwo(a:Int,b:Int):Int = {
       a+b
   }
   ```

   Explain why performing tail call optimization on the call to `tailCallOne` is invalid.

4. ```
   def factorial(n:Int):Int = { if(n <= 1) { 1 } else { n * factorial(n-1) } }
   def sum(n:Int,a:Int):Int = { if(n != 0) { sum(n-1,n+a) } else { a } }
   ```

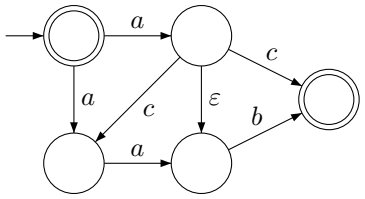
   Is the recursive call to `factorial` a tail call? Is the recursive call to `sum` a tail call? Explain.

3 Regular Languages

1. Draw DFAs for the empty language $\emptyset$ which contains no words, and the language $\{\varepsilon\}$ that contains only the empty word. Then write the formal definitions for these DFAs (for an arbitrary alphabet Σ).
2. Consider the language of all words over $\{a, b\}$ that end with $abba$.
   (a) Write a regular expression for this language.
   (b) Draw a DFA for this language.
   (c) Reverse the direction of every transition arrow in your DFA. Is the resulting automaton deterministic or nondeterministic (that is, is it still a DFA or is it now an NFA)?
   ```

3. A NUM token in Lacs consists of a single digit in the range 0–9, or two or more digits the first of which is not 0. Write a regular expression for valid NUM tokens.

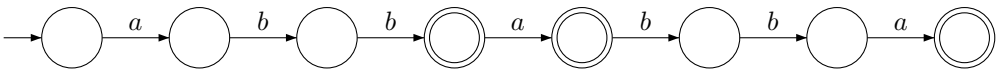
4. This NFA recognizes a finite language. Give a regular expression for the language, then list all the words in the language.



5. Draw a diagram of the DFA $(\Sigma, Q, q_0, A, \delta)$ where
   - $Q = \{0, 1, 2\}$, $\Sigma = \{a, b, c\}$, $q_0 = 0$ and $A = \{0, 2\}$.
   - $\delta(q, a) = (q + 1) \bmod 3$
   - $\delta(q, b) = \begin{cases} 1 - q, & \text{if } q \neq 2; \\ q, & \text{otherwise.} \end{cases}$
   - $\delta(q, c) = \begin{cases} 2, & \text{if } q = 1; \\ q, & \text{otherwise.} \end{cases}$

4 Scanning

Here is a DFA.



1. Suppose L is the language accepted by the above DFA. Draw a finite automaton that recognizes L^*. You can (and should) use ε-transitions. (Hint: Don't forget that L^* always contains ε.)

2. We say a word w can be *scanned with respect to* L if there exist words $w_1, \ldots, w_n \in L$ such that $w = w_1 \cdots w_n$. Explain how your finite automaton for L^* could be used to implement an algorithm that checks if a word can be scanned with respect to L.

   You don't need to give a step-by-step description of the algorithm or pseudocode for the algorithm, just explain how the finite automaton for L^* could help with the task.

3. Give an example of a word over $\{a, b\}$ that cannot be scanned with respect to L.

4. Give an example of a word that can be scanned with respect to L in multiple ways. That is, there should be at least two *different* sequences of words $w_1, \ldots, w_m \in L$ and $w'_1, \ldots, w'_n \in L$ such that $w = w_1 \cdots w_m = w'_1 \cdots w'_n$.

5. Determine the result of using Maximal Munch with this DFA on the following input words. If the scan is successful, give the resulting sequence of tokens. If it is not successful, give the tokens that were scanned before the error occurred, followed by "ERROR". Additionally, if Maximal Munch is not successful, indicate whether the word can be scanned with respect to L, and if it can, describe a valid scan of the word.

   (a) *abbaabbabb*

   (b) *abbaabbabbabba*

(c) *abbabbaabbaabba*

(d) *abbabbaabbabbabbabba*

5 Context-Free Languages

1. Give a context-free grammar for the language of balanced sequences of parentheses (sequences where every left parenthesis has a matching right parenthesis). For example, the sequence (()()(()))() is balanced, but the sequence (()(is not.
2. Consider the following context-free grammar:

$$S \to \varepsilon \mid aSbS \mid bSaS$$

   (a) For each of the following words, give a derivation of the word, and draw a parse tree.

      i. *baba*

      ii. *aaabbb*

      iii. *baabba*

   (b) Describe the language this grammar generates.

   (c) Prove that this grammar is ambiguous.

3. Give an example of a context-free language that is not regular, and an example of a language that is not context-free.

6 Parsing

1. The following grammar for arithmetic expressions with addition, multiplication and the number two is ambiguous:

$$\begin{aligned} expr &\to expr\ op\ expr \\ expr &\to 2 \\ op &\to + \mid * \end{aligned}$$

   This does not cause a problem for the CYK parsing algorithm. The CYK algorithm can parse ambiguous grammars and will return one of the possible parse trees. However, depending on what you do with the parse tree, the result could be undesirable.

   (a) Consider an expression evaluator that works as follows. First, it runs the CYK parsing algorithm on the expression to produce a parse tree. Next, it recursively evaluates the expression using the returned parse tree, as follows:

      - If the production at the root of the tree is $expr \to expr\ op\ expr$, recurse on the two child *expr* nodes to determine their value, then perform the operation derived from *op* on the two values and return the result.
      - If the production at the root of the tree is $expr \to 2$, return 2.

      Give a parse tree for the expression $2 + 2 * 2$ which will cause this expression evaluator to return an incorrect result.

(b) Give a different grammar for this expression language that is guaranteed to produce correct parse trees. You may introduce additional non-terminal symbols and assume the expression evaluator will handle them appropriately. (Hint: look at the Lacs grammar for inspiration.)

2. This question is about the inner workings of the CYK parser. Here is a brief review of how the CYK parser works. The CYK parser takes two inputs: a grammar, and a sequence of terminal symbols. It then calls a recursive function which does the actual parsing work. This recursive function, which we will call R, takes two arguments:

   - A sequence of terminals and non-terminals, which we will call α.
   - A substring of the input, which we will call x.

   The function R works as follows. If $\alpha \Rightarrow^* x$, then R returns a sequence of parse trees, one for each symbol in α, such that leaves of the trees spell out x when concatenated together. Otherwise, R returns a value to signal that the parse failed (in Scala we use "None" for this).

   Suppose we use the following grammar as input to our CYK parser:

   $$S \to A \mid SA$$
   $$A \to a$$

   For each of the following calls to R, draw the sequence of parse trees resulting from the call, or write "None" if the parse fails.

   (a) $R(A, a)$

   (b) $R(S, a)$

   (c) $R(aA, aa)$

   (d) $R(SA, aa)$

   (e) $R(S, aa)$

3. Give an example of an ambiguous grammar such that there are infinitely many parse trees for a single word. Explain how the CYK parsing algorithm is able to parse this grammar in finite time.

4. Give an example of a grammar that cannot be parsed by an LR parser.

5. Imagine you are designing the Marmoset tests for the Earley parsing bonus problem. You want to make sure that students cannot pass the tests by just making the `parseEarley` function do a call to `parseCYK`. What differences between the CYK and Earley algorithms could you exploit to achieve this?

6. Define the correct prefix property and explain why it is useful for a programming language parser to have this property.

7 Context-Sensitive Analysis

1. ```
   def main(a:Int,b:Int):Int = { f() }
   def f():Int = {
       def f():Int = { 0 }
       f()
   }
   ```

   Will this Lacs code compile? If it compiles and runs, will it recurse infinitely, or return zero? Explain.
   ```

2. In this question, you are asked to write out the symbol tables of some Lacs procedures. The symbol table should contain all the variable and procedure names defined in the procedure (including parameters), and it should specify whether each name corresponds to a procedure or a variable.

   Include type information about each variable and procedure. For example, for the procedure `f` below you would state that it is of type `(Int,Int)=>Int`.

```
def f(a:Int, b:Int):Int = {
    var c:Int;
    def g(a:Int, b:(Int)=>Int):Int = {
        b(a)
    }
    def h(c:Int):Int = {
        def g():Int = {
            c-b
        }
        g()
    }
    c = a+b;
    g(c,h)
}
```

   (a) Write out the symbol table of `f`.

   (b) Write out the symbol table of `g`. Indicate which variables are inherited from the symbol table of `f` and which ones were overriden by `g`.

   (c) Write out the symbol table of `h`. Indicate which variables are inherited from the symbol table of `f` and which ones were overriden by `h`.

3. Check the body of each of the following procedures for type correctness. If the body has no type errors, state the return type of the procedure. Otherwise, describe the type error.

   (a)
```
def f(b:(Int,Int)=>Int):??? = {
    b(b(1,2),b(3,4))
}
```

   (b)
```
def f(b:(Int,Int)=>Int):??? = {
    b(1,b(2,3),4)
}
```

   (c)
```
def f(a:Int,b:(Int)=>(Int)=>Int):??? = {
    if(a>0) {
        b(a)
    } else {
        b(0)
    }
}
```

   (d)
```
def f(a:Int,b:(Int)=>(Int)=>Int):??? = {
    if(a>0) {
        b(a)
    } else {
        b
    }
}
```

(e)
```
def f(a:(Int)=>(Int,Int)=>Int,b:(Int)=>(Int,Int)=>Int,c:Int):??? = {
    var d:((Int,Int)=>Int,Int)=>(Int)=>Int;
    def e(a:(Int,Int)=>Int,b:Int):(Int)=>Int = {
        a(b,c)
    }
    d = e;
    e(if(c>0) { a(c) } else { b(c) }, c)
}
```

(f)
```
def f(a:(Int)=>(Int,Int)=>Int,b:(Int)=>(Int,Int)=>Int,c:Int):??? = {
    var d:((Int,Int)=>Int,Int)=>(Int)=>Int;
    def e(a:(Int,Int)=>Int,b:Int):(Int)=>Int = {
        def d(c:Int):Int = {
            a(b,c)
        }
        d
    }
    d = e;
    e(if(c>0) { a(c) } else { b(c) }, c)
}
```

8 Memory Management

1. In this question, we will use the "explicit allocation and deallocation" method of memory management. In this method two functions are provided:

   - `malloc(requested_bytes)`, which allocates a block of size `requested_bytes + 8` (8 bytes for a header, and the rest free to use) and returns the address of the block. If there is not enough space for the block, or `requested_bytes` is not a positive multiple of 4, an error is returned.
   - `free(block_address)`, which frees the block at the specified address. If there are free blocks adjacent to the block that was just freed, these blocks will be coalesced (merged) into a single free block.

   Suppose we have an empty heap with 64 bytes of available memory (not including the "dummy block" at the start, which consists only of an 8-byte header and is used to make it easier to implement `malloc` and `free`).

   Describe a sequence of `malloc` and `free` calls that results in the heap having 24 or more bytes of free space, but enough fragmentation that a `malloc(8)` call will fail.

2. Below is a diagram of a small stack and heap. The heap contains 128 bytes in total and is divided into two 64 byte semispaces. The heap starts at address 128. Currently, the first semispace (lower addresses) is the "from-space" and the second semispace (higher addresses) is the "to-space".

   Suppose garbage collection is performed with the stack and heap in this state. This will result in all the reachable blocks in the first semispace of the heap being copied into the second semispace. Additionally, the size of each reachable block in the first semispace will be negated (to mark the block as copied), and the "number of pointers" field in each reachable block will be overwritten with the block's new address. All reachable pointers to the moved block should be updated. Unreachable blocks in the first semispace are not modified.

   Fill out the "After Garbage Collection" diagram of the heap and stack with the correct values.

The order in which the chunks are copied to the heap matters to an extent. You should first copy the chunks that are reachable from the stack, and then the chunks that are reachable from those chunks, and so on. However, it does not matter whether (for example) you scan the stack from top to bottom or from bottom to top, as this is just an implementation detail that does not affect the correctness of the algorithm.

Stack
32
4
128
152
128
192
140
42

Before Garbage Collection			
Heap Semispace 1		Heap Semispace 2	
Address	Value	Address	Value
128	12	192	0
132	1	196	0
136	168	200	0
140	12	204	0
144	1	208	0
148	140	212	0
152	16	216	0
156	2	220	0
160	128	224	0
164	168	228	0
168	24	232	0
172	2	236	0
176	152	240	0
180	0	244	0
184	128	248	0
188	140	252	0

After Garbage Collection			
Heap Semispace 1		Heap Semispace 2	
Address	Value	Address	Value
128		192	
132		196	
136		200	
140		204	
144		208	
148		212	
152		216	
156		220	
160		224	
164		228	
168		232	
172		236	
176		240	
180		244	
184		248	
188		252	

Stack

3. (a) Define what it means for a block in memory to be live.

   (b) Define what it means for a block in memory to be reachable.

   (c) If a block is reachable, is it necessarily live?

   (d) If a block is live, is it necessarily reachable?

   (e) If a block is unreachable, can it be live?

   (f) If a block is not live, can it be reachable?