17-355/17-665/17-819: Program Analysis

Fall 2023 Midterm Exam # 1

Claire Le Goues, Fraser Brown, and Daniel Ramos

Name: ____

Andrew ID: ____

Study Guide Instructions: This study guide is intended to help you prepare for the first midterm. It consists of several types of material:

1. Material providing content or context that *will actually be on the test*, especially the background on regular expressions (Question 1, Operational Semantics) and the background on alias pairs analysis (Question 2). This material is provided so you can familiarize yourself with it ahead of time, and not have to wade through background material and wrap your head around it in a limited timespan.
2. Question *templates* that specify a type of question that may or may not be asked, but without particulars. Questions 1(a) and 1(b), or all of Question 3, are good examples. You can practice answering those kinds of questions by filling in reasonable examples for the blanks, or otherwise making sure you understand how to answer this kind of question.
3. Concrete questions, which may or may not be on the exam, or may be similar to those that will be on the exam.

We cannot promise to have full coverage of all material in the course so far, or all of the questions that may ultimately be on the exam. However, we have attempted to be thorough, and have tried to give you a sample of the *types* of questions we will be asking about the material in the course. We expact that if you carefully study this material and the lecture notes, you will be well-prepared for the exam.

You will be permitted to bring as many pages of notes/study material as you want, including (but not limited to) this study guide with your notes (if you want).

Question 1: Operational Semantics (0 points)

*Note: the exam **will** contain questions based on this idea of specifying inference rules for regular expressions, and so we include this material in the guide so you can wrap your head around how regular expressions work and could be precisely specified in advance.*

Consider the following abstract grammar for regular expressions:

$$\begin{array}{rcll} e & ::= & \text{“}x\text{”} & \text{singleton — matches the character } x \\ & | & \texttt{empty} & \text{skip — matches the empty string} \\ & | & e_1 e_2 & \text{concatenation — matches } e_1 \text{ followed by } e_2 \\ & | & e_1 \mid e_2 & \text{or — matches } e_1 \text{ or } e_2 \\ & | & e* & \text{Kleene star —matches 0 or more occurrences of } e \end{array}$$

We also give an abstract grammar for strings (modeled as lists of characters; we write ″bye″ as shorthand for ″b″ :: ″y″ :: ″e″ :: nil):

$$\begin{array}{rcll} s & ::= & nil & \text{empty string} \\ & | & \text{“}x\text{”} :: s & \text{string with first character } x\text{, and other characters } s \end{array}$$

We introduce a new judgement to give large-step operational semantics rules of inference for regular expressions matching strings:

$$\vdash\ e\ \texttt{matches}\ s\ \texttt{leaving}\ s'$$

A regular expression e applied to string s means that e matches some prefix of s, leaving the suffix s' unmatched. If $s' = nil$, then e matched s exactly; if $s' = s$, then e does not match any part of s. For example: $\vdash$ “h” `matches`“$hello$” `leaving` “$ello$”; the concatenation construct means that $\vdash$ “he” `matches`“$hello$” `leaving` “llo”. Note that this semantics may be non-deterministic! We come back to this later.

Here are two of the simpler rules of inference for regular expressions:

$$\frac{s = \text{“}x\text{″} :: s'}{\vdash \text{“}x\text{″}\ \texttt{matches}\ s\ \texttt{leaving}\ s'}\ \textit{singleton} \qquad \frac{}{\vdash \texttt{empty}\ \texttt{matches}\ s\ \texttt{leaving}\ s}\ \textit{skip}$$

(a) Precisely specify [one of Kleene star, concatenation, or][1] language construct(s) via one or more large-step operational semantics inference rules.

(b) Consider these potential rules of inference for [one of Kleene star, concatenation, or]:

$$\frac{premises - 1}{conclusion - 1}\ \textit{rule-1} \qquad \frac{premises - 2}{conclusion - 2}\ \textit{rule-2-WRONG}$$

i. Recall that a logical system is *complete* if every true statement is provable; it is *sound* if every provable statement is true. Assuming the other rules are correct, the `rule-2-WRONG` rule makes our overall system:

○ Unsound

○ Incomplete

ii. Prove it, by giving either an example of a true statement that cannot be proven with this rule, or a provable judgement that is untrue.

iii. Write a correct rule for the language construct.

[1] An actual exam question would specify

(c) We noted above that these semantics by be non-deterministic! That is, it is possible to apply the same regular expression to one string in two different ways such that you get two different answers. We can write this more formally as:

$$\exists r \in e\ .\quad \exists s, s', s'' \in \texttt{Str}\ .\quad \vdash\ r\ \texttt{matches}\ s\ \texttt{leaving}\ s' \quad \wedge \vdash\ r\ \texttt{matches}\ s\ \texttt{leaving}\ s'' \quad \wedge s' \neq s''$$

Prove this, by giving an example expression e and string s, and two different valid derivations for matching e to s that results in two different strings "left"

i. Give an example e

ii. Given an example s

iii. Give an s' and a valid derivation for $\vdash\ e\ \texttt{matches}\ s\ \texttt{leaving}\ s'$ using the rules of inference for regular expression above (you can use your new rule if you want).

iv. Give a different s'' and a valid valid derivation for $\vdash\ e\ \texttt{matches}\ s\ \texttt{leaving}\ s''$ using the rules of inference for regular expression above (you can use your new rule if you want).

(d) This non-determinism is sub-optimal; we would instead prefer operational semantics for a judgement that returns the *set* of all possible suffixes. If S is a set of strings s, we could change our previous judgement accordingly, to: $\vdash e$ matches s leaving S, and then use rules of inference like the following:

$$\frac{}{\vdash\ \text{``}x\text{''}\ \texttt{matches}\ s\ \texttt{leaving}\ \{s' | s = \text{``}x\text{''} :: s'\}}\ \textit{singleton}' \qquad \frac{}{\vdash\ \textit{empty}\ \texttt{matches}\ s\ \texttt{leaving}\ \{s\}}\ \textit{empty}'$$

$$\frac{\vdash e_1\ \texttt{matches}\ s\ \texttt{leaving}\ S \quad \vdash e_2\ \texttt{matches}\ s\ \texttt{leaving}\ S'}{\vdash e_1 | e_2\ \texttt{matches}\ s\ \texttt{leaving}\ S \cup S'}\ \textit{or}$$

Do one of the following:

- *either* give operational semantics rules of inference for [one of: conctenation, or, Kleene star]. You may not place a derivation inside a set constructor, as in $\{x | \exists y.\ \vdash e\ \texttt{matches}\ x\ \texttt{leaving}\ y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in two–five sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but "wrong" rules of inference, and demonstrate that each one is incorrect—either unsound or incomplete—with respect to our intuitive notion of regular expression matching.

(e) Your roommate looks over your shoulder as you are reviewing for this exam, and sees the following inference rule for small-step semantics for WHILE3ADDR copy:

$$\frac{P[n] = x := y}{P \vdash \langle E, n \rangle \leadsto \langle E[x \mapsto E[y]], n+1 \rangle}\ \textit{step-copy}$$

They ask "Hey why do you need that premise in that rule? You wouldn't need it in the WHILE equivalent small-step rules..."

(f) Induction on the structure of expressions is sufficient to prove many properties about them (like the one we did in class, showing that the number of literals and variable occurrences in some expressions is L(a) = O(a) + 1). Why, in one sentence, can't we induct on statement structure to prove most other interesting properties about WHILE (and have to induct on the structure of the derivation, instead)?

Question 2: Analysis Specification (0 points)

*Note: the exam **will** contain questions based on this idea of alias pairs analysis; we will introduce pointer analysis in class before the exam. An in depth understanding of actual pointer analysis algorithms is not required to understand/answer these questions or any questions we will put on the exam. We reprint some introductory material on pointers from the text in case you want to start studying before we cover it.*

Pointers are variables whose value refers to another value elsewhere in memory, by storing the address of that stored value. Ignoring for the moment memory allocation and arrays, we can decompose all pointer operations in C into four types:

$$\begin{array}{lcll} I & ::= & \ldots & \\ & | & p := \&x & \text{taking the address of a variable} \\ & | & p := q & \text{copying a pointer from one variable to another} \\ & | & *p := q & \text{assigning through a pointer} \\ & | & p := *q & \text{dereferencing a pointer} \end{array}$$

Pointers matter in analyzing real programs. Consider for example constant-propagation analysis of the following program:

$$\begin{array}{rl} 1: & z := 1 \\ 2: & p := \&z \\ 3: & *p := 2 \\ 4: & \text{print } z \end{array}$$

To analyze this program correctly we must be aware that at instruction 3, p points to z. If this information is available we can use it in a flow function as follows:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(y) \mid z \in \textit{must-point-to}(p)]$$

When we know exactly what a variable x points to, we have *must-point-to* information. This isn't always possible, depending on the programming language. Also, sometimes we could be uncertain about which of several locations p could point to, such as if it's set to a different location in the **then** or **else** parts of an **if** statement. We call tracking this kind of situation *may-point-to* information, and we could use it in constant propogation like:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(z) \sqcup \sigma(y) \mid z \in \textit{may-point-to}(p)]$$

An alternative to a constraint-based approach (as we will cover in class) is an *alias pairs* analysis, which computes, at each program point, a set of pairs of expressions that may alias one another. An expression is either a variable such as x, or a single dereference of a pointer variable such as *x. We do not track aliased pairs including more dereferences—that is, nothing like ***x. To illustrate, the pair (*x, y) means that x may point to y, whereas the pair (*x, *y) means that x and y may point to the same memory location.[2]

For example, consider the following program:

[2] We assume for simplicity that the pair (x,y) cannot occur, which is true in C and Java, but not C++.

```
1 :   s := 2
2 :   x := &y
3 :   y := &z
4 :   t := &s
5 :   w := t
```

This analysis would compute the following pair sets immediately after each program location:

location	alias pairs
1	∅
2	{ (*x, y) }
3	{ (*x, y) (*y, z) }
4	{ (*x, y) (*y, z) (*t, s) }
5	{ (*x, y) (*y, z) (*t, s) (*w, s) (*w, *t) }

(a) Define a lattice L and analysis information σ for this analysis.

(b) What do *top* and *bottom* correspond to in this lattice? (the answer $\top$ is incorrect). [3].

(c) Define the *ordering relation* between lattice elements. i.e., when is $\sigma_1 \sqsubseteq \sigma_2$?

(d) Define the *join operation* on lattice elements. i.e.,. what is $\sigma_1 \sqcup \sigma_2$?

(e) Assume we have the alias information σ = FOO, and consider analyzing the statement BAR.

  i. Which alias pairs in the state should be *killed* by the statement?

  ii. Which alias pairs should be *generated* by the statement?

(f) Consider the statement EXAMPLE CODE. If the alias information before the statement is σ = EXAMPLE1, what is the alias information after the statement?

(g) Assuming monotonic flow functions, will the analysis on the alias pair lattice defined above terminate? Why or why not? Your answer should be precise in terms of bounding the height of the lattice.

(h) Imagine we relax the assumption about the number of dereferences tracked. E.g., consider an alternative lattice that allows a pair (****x,y) (whereas the original analysis only allows pairs like (*x, y). Is the analysis guaranteed to terminate on this new lattice? Why or why not?

[3] cf. constant propagation: $\top = \mathcal{Z}$

Question 3: Soundness (0 points)

Imagine we want to extend X analysis to a language with Y. Consider the following *incorrect* flow function:

$$f_{FOO}[\![CODE]\!](\sigma) \quad = \sigma[...update...]$$

This function is incorrect because it does X; to see this, consider code that does Y.

(a) Prove that this flow function is not locally sound.

(b) Specify a correct flow function.

(c) Prove that your new flow function is monotonic.

(d) Why is it a good idea to apply a widening operator only at loop heads in a control flow graph?

(e) Why don't we need a widening operator for zero or sign analysis?

(f) Why can't we use the basic operational semantics to reason about the correctness of a reaching definitions analysis?

Question 4: Interprocedural Analysis (0 points)

Imagine you are would like to implement an interprocedural X analysis. Consider the following simple test code:

...example omitted...

(a) Imagine you wanted to use default assumptions around function calls.

   i. Provide an example of default assumptions that produces sound but imprecise results for the example.

   ii. Provide an example of default assumptions that produces more precise output on this example.

   iii. Provide an example for which the more precise assumptions produces incorrect dataflow output.

(b) Would porting an intraprocedural analysis and applying it to the interprocedural control flow graph produce satisfactory analysis output on this example? Why or why not?

(c) Provide an example program that demonstrates a case where function inlining is a bad solution to the interprocedural control flow problem, and explain why it shows that.

(d) What is one reason that dynamic dispatch poses a challenge to interprocedural dataflow analysis?

(e) Assuming a context-sensitivity limit of X, which context-sensitivity-limiting approach would provide more precise results for the following example function:

...example omitted...

○ Call strings

○ Contexts

Justify your answer: