Makeup Quiz

CS-320, Computer Language Processing, Fall 2016

Thursday, December 15, 2016

General notes about this quiz

- Have your CAMIPRO card ready on the desk.
- You are allowed to use any printed material (no cursive or script fonts) that you brought yourself to the exam. You are not allowed to use any notes that were not typed-up. Also, you are not allowed to exchange notes or anything else with other students taking the quiz.
- Try to write your answers in the space provided in the question paper
- If you need separate sheets, for each question, write your answers on a separate sheet.
- Make sure you write your name on each sheet of paper.
- Use a permanent pen with dark ink.
- It is advisable to do the questions you know best first.
- You have in total **3 hours 40 minutes**.

Exercise	**Points**	
1	25	
2	25	
3	25	
4	25	
Total	100	

Problem 1: Automata and Lexical Analysis (25 points)

Say we are given a set of token classes tok_1, tok_2, $\cdots$, tok_n which are defined by regular expressions: r_1, r_2, $\cdots$, r_n. A triple $\langle tok_i, tok_j, tok_k \rangle$ means that a string belonging to the token class tok_i can be *concatenated* to a string belonging to the token class tok_j to form a string belonging to the token class tok_k. For example, consider the following set of tokens

- EQ: =
- IMP: =>
- GT: >
- ID: $(a|b)(a|b|0|1)^*$
- DIG: $(0|1)^*$

For these tokens, the list of such triples we would like to compute are $\langle EQ, GT, IMP \rangle$, $\langle ID, ID, ID \rangle$, $\langle DIG, DIG, DIG \rangle$ and $\langle ID, DIG, ID \rangle$.

a) [25 pts] Provide an algorithm for computing all the triples as described above for any arbitrary set of token classes. You are allowed to use any reductions or algorithms or properties explained in the lectures as subroutines in you algorithm. Remember that the inputs to you algorithm are the token classes tok_1, tok_2, $\cdots$, tok_n and their defining regular expressions r_1, r_2, $\cdots$, r_n.

Problem 2: Parsing and Grammars (25 points)

Consider the grammar for postfix expressions, where $\{a, b, c, +, *, -\}$ are terminals.

$$E ::= E\ E\ +\ |\ E\ E\ *\ |\ E\ E\ -\ |\ ID$$
$$ID ::= ID\ c\ |\ ID\ b\ |\ a$$

a) [5 pts] Show all parse trees for the string: $abcaa + *$

Let $L(G)$ denote the set of strings that are parsable by the above grammar. Let S_n denote the set of strings of length n belonging to the grammar. That is, $S_n = \{w \mid w \in L(G) \wedge |w| = n\}$. Let P_n be the set of parse trees of strings belonging to the set S_n. That is, $P_n = \{t \mid w \in S_n \wedge t \text{ is a parse tree of } w\}$.

b) [5 pts] Can the set P_n have more than 6^n elements? Justify your answer.

c) [5 pts] Is this grammar LL(1)? If not, provide one reason.

d) [**10 pts**] We know we can use the CYK parsing algorithm to check whether a string belongs to the above grammar. However, we also know that the CYK algorithm runs in $O(n^3)$ time for this grammar. Can you provide a linear time parsing algorithm for this grammar? That is, given a string w we need to check within $O(|w|)$ time if $w \in L(G)$. You are allowed to use any algorithm we described in the lectures as a subroutine if necessary.

Problem 3: Union Types (25 points)

Consider a simple language with arithmetic operations +,/, if-else expression, assignment and block statements, that has the following types: Pos, Neg, Int and $Bool$. Consider a type system for the language with the following type rules: (Assume there can be no overflows.)

$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{k > 0}{\vdash k : Pos} \quad \frac{k < 0}{\vdash k : Neg}$$

$$\frac{k \in \{true, false\}}{\vdash k : Bool} \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2} \qquad Pos <: Int \quad Neg <: Int$$

$$\frac{\Gamma \vdash x : Pos \qquad \Gamma \vdash y : Pos}{\Gamma \vdash x + y : Pos} \qquad \frac{\Gamma \vdash x : Neg \qquad \Gamma \vdash y : Neg}{\Gamma \vdash x + y : Neg} \qquad \frac{\Gamma \vdash x : Int \qquad \Gamma \vdash y : Int}{\Gamma \vdash x + y : Int}$$

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \vdash y : Pos}{\Gamma \vdash x/y : Int} \qquad \frac{\Gamma \vdash x : Int \qquad \Gamma \vdash y : Neg}{\Gamma \vdash x/y : Int} \qquad \frac{(x,T) \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash x = e : Unit}$$

$$\frac{s_1 : Unit \qquad \Gamma \vdash \{s_2 \cdots s_n\} : Unit}{\Gamma \vdash \{s_1; \cdots ; s_n;\} : Unit} \qquad \frac{\Gamma \oplus (x,T) \vdash \{s_2 \cdots s_n\} : Unit}{\Gamma \vdash \{\text{var } x : T; s_2; \cdots ; s_n;\} : Unit}$$

$$\frac{\Gamma \vdash c : Bool \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if } (c)\ e_1 \ \text{ else } \ e_2 : T}$$

Consider the code snippet shown in Figure 1. Γ_0 is the initial type environment before the beginning of the code snippet.

$\Gamma_0 = \{(x, Pos), (y, Neg), (b1, Bool), (b2, Bool)\}$

```
{
  var z : Int
  z = if (b1) x else y;
  x = x + 1;
  y = y + (−1);
  x = x / (if (b2) y else z);
}
```

Figure 1: A code snippet that does not type check

a) [5 pts] Will the code snippet shown in Figure 1 type check as per the given type rules? If not show the first statement that will not type check.

Consider an extension to the type system that can type expressions with unions of two or more types belonging to the language e.g. like $(Neg \cup Pos)$, $((Pos \cup Neg) \cup Bool)$. A variable or expression that has a *union type* can take values that belong to any of the types in the union. For example, if x has type $(Neg \cup Pos)$ it implies that at runtime x can take negative or positive values but cannot be zero. More generally, if $x : (A \cup B)$, the runtime value of x should belong to type A or type B (or both). Consider the following extension to the above type system that allows union types.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : (A \cup B)} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : (A \cup B)}$$

Several axioms that hold for union of two sets also holds for union of two types. In particular, the following axioms can be used to simplify (or normalize) the union types that arise during type checking.

$$\begin{aligned} (A \cup B) &= (B \cup A) && \text{Commutativity} \\ (A \cup (B \cup C)) &= ((A \cup B) \cup C) && \text{Associativity} \\ (A \cup A) &= A && \text{Idempotence} \end{aligned}$$

b) [5 pts] Complete the following subtyping rule

$$\frac{?}{\Gamma \vdash (A \cup B) <: T}$$

c) [5 pts] Define a type rule for *if-else* expression that uses union types. Make sure that your type rule enables you to type check the code snippet shown in Figure 1. You can change the declared type of the local variable z.

$$\frac{\Gamma \vdash c : ? \qquad \Gamma \vdash e_1 : ? \qquad \Gamma \vdash e_2 : ?}{\Gamma \vdash \text{if } (c)\ e_1 \ \text{ else } \ e_2 : ?}$$

d) [3 pts] Define a more permissive type rule for the division expression using union types. As before, make sure that your type rule enables you to type check the code snippet shown in Figure 1.

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \vdash y : ?}{\Gamma \vdash x/y : ?}$$

e) **[7 pts]** Show the type derivation for the statement "x = x / (if (b2) y else z)" of Figure 1 as per your type rules. You may choose an appropriate type for the variables x, y and z that is consistent with the code snippet and the type rules you came up with. Use the axioms, such as associativity and commutativity, wherever needed to simplify the union types. You need not explicitly show the application of the axioms.

Problem 4: Tail-call Elimination (25 points)

In this exercise, will consider an optimization that converts recursion into loops in specific scenarios called tail-call elimination. Let *rec* be a recursive function in the program of the following form:

```
def rec(x) = {
  if (c) {
    s1;
    ...
    sn;
    return r;
  } else {
    sn+1;
    ...
    sm;
    return rec(z);
  }
}
```

In the above code snippet $s1$, $s2$, $\cdots$ are arbitrary statements that are not function calls. Our goal is to transform such code snippets to an equivalent code that does not have recursive calls.

a) [25 pts] Provide a translation for the tail recursive functions of the above form using only the statements in original program, and additionally a `while` loop. You can create any temporary variables you may need. Note that you need not translate the function to bytecode. It is sufficient to show a code snippet that does not have recursion, but is equivalent to the given code snippet.

[def rec(x) = {if(c) { $s_1; ...s_n;$ return r; } *else* { $s_{n+1}; ...; s_m;$ return rec(z); }] =

```
def rec(x) = {
  ?
```