Quiz

CS-320, Computer Language Processing, Fall 2016

Wednesday, November 23rd, 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	15	
2	20	
3	15	
4	25	
5	25	
Total	100	

Problem 1: Left-canceling Languages (15 points)

Let $\Sigma = \{a, b\}$ for a, b distinct, and L_1, L_2, L range over subsets of Σ^* (languages). Remember that for languages, concatenation is defined by

$$L_1 L_2 = \{u_1 u_2 \mid u_1 \in L_1, u_2 \in L_2\}$$

We say that L **left-cancels** if and only if, for every L_1, L_2,

$$LL_1 = LL_2 \text{ implies } L_1 = L_2.$$

For all of the following questions, a correct answer is sufficient for full points; you do not need to show how you obtained your answer (but you may do so as you may exceptionally obtain partial credit for incorrect solution).

a) **[2 pts]** Does $L = \emptyset$ left-cancel?

b) **[2 pts]** Does $L = \{\varepsilon\}$ left-cancel?

c) **[7 pts]** Give a regular expression describing an infinite language L that left-cancels.

d) **[4 pts]** Give a context-free grammar for another language L that left-cancels and that is not a regular language.

Problem 2: Lexical Analysis (20 points)

Suppose that token classes of a lexer are given by the following regular expressions:

- HEX:$(0|1|a)(0|1|a)*$
- ID: $a(a|b|0|1)*$
- EOF: \$

Suppose also that there are no comments or white spaces in this language and that HEX has priority over ID. The input is a string satisfying the regular expression

$$(0|1|a|b)^*\$$$

where \$ is a special end-of-file character.

a) [5 pts] Show the automaton for the lexical analyzer that accepts the above two token classes. Assume that the execution mechanism interprets the automaton by applying longest match rule and restarting the state machine after each token is identified.

b) **[3 pts]** Consider the following input string:

$$01ab\$$$

What is the result of running the lexical analyzer on this input using longest-match rule and the priority of HEX over ID?

c) [**6 pts**] Show a deterministic finite state machine that accepts precisely those strings of characters that are accepted by repeatedly running the above lexical analyzer with longest match rule and priority, until it processes the entire input. Unlike the diagram in part "a)", your machine should have only **one** kind of state, which accepts the **entire** input. Your machine should **not** accept a string if the longest match rule is violated. The machine simply gives accept/reject answers; it need not indicate how the sequence was split into tokens.

d) **[6 pts]** Generalizing the previous part, consider an algorithm that given two arbitrary regular expressions e_1 and e_2 for token classes, constructs a deterministic automaton that accepts strings of characters corresponding to multiple tokens of token classes e_1, e_2 and EOF (defined as \$), while applying longest match rule. You can describe the automaton by defining mathematically its transitions if it is easier or shorter than drawing the diagram. If the problem is not solvable in general, show regular expressions e_1 and e_2 for which it is not possible to build such an automaton.

Problem 3: Parsing and Grammars (15 points)

Consider the grammar

$$
\begin{aligned}
&\textit{decl} ::= \textit{varDecl} \mid \textit{funDecl}\\
&\textit{varDecl} ::= \textit{type}\ \texttt{ID};\\
&\textit{funDecl} ::= \textit{type}\ \texttt{ID}\ (\textit{optIDs});\\
&\textit{optIDs} ::= \varepsilon \mid \textit{IDs}\\
&\textit{IDs} ::= \texttt{ID} \mid \textit{IDs},\ \texttt{ID}\\
&\textit{type} ::= \texttt{int} \mid \textit{type}*
\end{aligned}
$$

a) **[5 pts]** Compute nullable and first for all non-terminals of the above grammar, using the following table.

non-terminal	✓ = *nullable*	✓ = in *first*					
		`ID`	;	(	)	`int`	*
decl							
varDecl							
funDecl							
optIDs							
IDs							
type							

b) **[2 pts]** Explain why the above grammar is not LL(1).

c) [**8 pts**] Give an LL(1) grammar describing the same sequences of tokens as the previous grammar.

Problem 4: Interval Types (25 points)

Consider a language that allows integer valued variables to be typed using a range of values it can possibly take. Let $\mathbb{Z}_{32} = \{x \in \mathbb{N} \mid -2^{31} \leq x \leq 2^{31} - 1\}$ denote the set of 32 bit integers. The types of our language are: $T ::= Bool \mid [a, b]$, where $a \in \mathbb{Z}_{32}$, $b \in \mathbb{Z}_{32}$ and $a \leq b$. If an expression e has an interval type $[a, b]$ then it means that its runtime value has to be an integer v in the range $a \leq v \leq b$.
Consider a functional language that has the following expressions. Integer and boolean constants: $\mathbb{Z}_{32} \cup \{true, false\}$, identifiers, denoted id, primitive operations over expressions: { **==**, **+**, **/** }, if-else expression: if(e_1) e_2 else e_3, let expression: let $id = e_1$ in e_2, and function calls: $f(e_1, \cdots, e_n)$, and function definitions. Find below the usual type rules for some of the constructs of the language.
Type Rules:

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{k \in \{true, false\}}{\vdash k : Bool} \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash \text{if}(e_1)\ e_2 \text{ else } e_3 : Bool}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 == e_2 : Bool} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \oplus \{(id, T_1)\} \vdash e_2 : T}{\Gamma \vdash \text{let } id = e_1 \text{ in } e_2 : T}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \cdots \qquad \Gamma \vdash e_1 : T_n \qquad \Gamma \vdash f : T_1 \times \cdots \times T_n \to T}{\Gamma \vdash f(e_1, \cdots, e_n) : T}$$

$$\frac{\Gamma \oplus \{(x_1, T_1), \cdots, (x_n, T_n)\} \vdash e : T}{\Gamma \vdash \textbf{def } f(x_1 : T_1, \cdots, x_n : T_n) : T \ = \ e : ok}$$

a) [5 pts] Complete the following sub-type relation on interval types.

$$\frac{?}{[a, b] <: [c, d]}$$

b) [2 pts] Since our language has arithmetic operations and 32 bit integers, we have to deal with overflows, which are situations where a result of an operation is too large or too small to fit within the available bits. Assume that you are given an operation $+_{32} : \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathbb{Z}_{32}$ that can add two 32 bit integers and return a 32 bit integer using a two's complement representation. This function has the following property (+ is the normal addition over integers):

$$\begin{array}{ll} (n +_{32} m) < 0 & \text{if } n + m > 2^{31} - 1 \\ (n +_{32} m) \geq 0 & \text{if } n + m < -2^{31} \\ (n +_{32} m) = n + m & \text{Otherwise} \end{array}$$

Define a function $\mathit{overflow}([a, b], [c, d])$ that takes two interval types and returns true if and only if there exists two integers belonging to the two types, respectively, whose *sum* results in an overflow. You can use only the operation $+_{32}$, logical operations like &&, || and !, and comparison operations like <, <= etc. to define the function.

$$\mathit{overflow}([a, b], [c, d]) = \quad ?$$

c) **[8 pts]** Complete the type rules for the expressions listed below. Notice that we have two rules for *addition*: one where overflow can happen and another where overflow cannot happen. For the division of two expressions e_1/e_2, ensure that the divisor (e_2) can take only **positive** (excluding zero) values. Fill in the missing conditions of the rules appropriately. In your answer, you can use the $+_{32}$ operation, integer division and any other standard operations on integers like exponentiation, abs etc. Make sure that your type rules are as precise as possible, unless their is a possibility of an overflow. That is, if an expression e is given a type T then it should not be possible to also type it as T' such that $T' <: T$, unless e may result in an overflow. If e may result in an overflow (given the types of the variables used by e), the type of e need not be as precise as possible.

$$\frac{k \in \mathbb{Z}_{32}}{\vdash k : \ ?}$$

$$\frac{? \qquad\qquad overflow(\quad ? \quad)}{\Gamma \vdash e_1 + e_2 : \ ?}$$

$$\frac{? \qquad\qquad !overflow(\quad ? \quad)}{\Gamma \vdash e_1 + e_2 : \ ?}$$

$$\frac{?}{\Gamma \vdash e_1 / e_2 : \ ?}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : [a, b] \qquad \Gamma \vdash e_2 : [c, d]}{\Gamma \vdash \text{if}(e_1) \ e_2 \ \text{else} \ e_3 : \ ?}$$

d) **[2 pts]** Say we now augment our language with an integer array type $Array[k]$ that has size k. We also introduce two array expressions: **new** $Array(k)$ and $a(i)$. The expression **new** $Array(k)$ creates a new array of size k for storing integers of arbitrary value. The expression $a(i)$ reads the element at the index i of the array a. Complete the type rule for $a(i)$ shown below so that it type checks if and only if it does not throw ArrayIndexOutOfBoundExpection. The exception would be thrown if the value of i is outside the possible indices of the array i.e, 0 and size of a minus 1.

$$\frac{k \geq 0 \land k \in \mathbb{Z}_{32}}{\vdash \textbf{new} \ Array(k) : Array[k]} \qquad\qquad \frac{?}{\Gamma \vdash a(i) : \ ?}$$

e) **[5 pts]** Consider the following program. Come up with types for the function search so that the program type checks in your type system. You **need not** show the type derivation. It suffices to show the types for the parameters and return value.

```
def main(a: Array[100]) {
  search(a, 0, 100, −10)
}

def search(a: ?        , i: ?        , j: ?        , key: ?        ): ?        = {
  if (a(i) == key)) i
  else if(i == j) −1
  else {
    let mid = (i + j)/ 2 in
      let r1 = search(a, i, mid, key) in
        if(r1 == −1) search(a, mid, j, key)
        else r1
 }
}
```

f) **[3 pts]** Say we change the type of the array a to $Array[2^{31} - 1]$ and the call to search in the function main to search(a, 0, $2^{31} - 1$, −10). Can you now come up with a type for search so that the program type checks in your type system. If so, show the type of search. If not, briefly explain why. Here again you **need not** show the type derivation.

Problem 5: Code Generation for Pattern Matching (25 points)

Consider a language that has a built-in list case class having two constructors Cons and Nil as shown below.

```
sealed abstract class List
case class Cons(head: Int, tail: List) extends List
case class Nil() extends List
```

Say our language has a pattern matching expression as in Scala of the following form.

```
e match {
  case pat_1 ⇒ e_1
  ...
  case pat_n ⇒ e_n
}
```

In the above match expression, e, $e_1 \cdots e_n$ are expressions belonging to the language, and each pat_i, $1 \leq i \leq n$ is a pattern whose syntax is defined by the following grammar.

$$pattern ::= \mathsf{Cons}(\_, pattern) \mid \mathsf{Cons}(\_, \_) \mid \mathsf{Nil}()$$

Underscore (_) is a wildcard that matches any value.

a) [10 pts] Your first task is to define a function $branchPat(pat, tl, fl)$ similar to the *branch* function we explained in the lectures. The function should match the pattern *pat* against *the value at the top of the stack*. If the matching succeeds it should jump to the label *tl*, and otherwise to the label *fl*. You are allowed to use any valid Java byte code instruction. However for your reference, we have listed some byte code instructions you may need at the end of the problem.
Hint: (a) You can invoke branchPat recursively on sub-patterns if necessary. (b) You can duplicate the top of the stack using the byte code instruction dup *and discard the top of the stack using the byte code instruction* pop.

If $pat = \mathsf{Cons}(\_, pat2)$,

$$branchPat(pat, tl, fl) = \quad ?$$

If $pat = \mathsf{Cons}(\_, \_)$,

$$branchPat(pat, tl, fl) = \quad ?$$

If $pat = \mathsf{Nil}()$,

$$branchPat(pat, tl, fl) = \quad ?$$

b) [5 pts] Use the *branchPat* function to define the translation for the match expression. The label lafter is the label you should jump to after the match expression. You can use $[e]$ and $[e_i]$ to denote the code of the expressions.

$[e$ **match** {
  case $pat_1 \Rightarrow e_1$ $\quad = \quad ?$
  ...
  case $pat_n \Rightarrow e_n$
}] lafter

Say we now want to allow binders (i.e, identifiers) in pattern matching and also optional guards. The new patterns are defined by the following grammar.

pattern ::= *simplePattern* **if** *guard*
simplePattern ::= *id* | Cons(*id*, *simplePattern*) | Nil()

id is an identifier such as x, y etc. and *guard* is a boolean expression. The pattern evaluates to true only if *simplePattern* matches the object reference at the top of the stack, and the *guard* evaluates to true. Given a match case: "**case** Cons(x, _) **if** $guard \Rightarrow e_1$", the *guard* and the body of a match case e_1 can use the binder x in the pattern of the match case. The binders of the match cases that did not succeed can take any value.

c) [10 pts] Define the function *branchPat* for the new patterns with binders and guards. Assume that the names of binders in the patterns are distinct. You can refer to the address of a binder x of a pattern using $\#x$. For instance, istore_$\#x$ binds x to the value at the top of the stack. You can also use the function $branch(guard, tl, fl)$ that jumps to tl if *guard* evaluates to true and to fl otherwise.

If $pat = pat2$ **if** *guard*,

$$branchPat(pat, tl, fl) = \quad ?$$

If $pat = x$,

$$branchPat(pat, tl, fl) = \quad ?$$

If $pat = \mathsf{Cons}(x,\ pat2)$,

$$branchPat(pat, tl, fl) = \quad ?$$

Java byte code instructions

iload_#x	Loads the integer value of the local variable x on the stack.
iconst_x	Loads the integer constant x on the stack.
istore_#x	Stores the current value on top of the stack in the local variable in x
iadd	Pop two (integer) values from the stack, add them and put the result back on the stack.
isub	Pop two (integer) values from the stack, subtract them and put the result back on the stack.
ifXX L	Pop one value from the stack, compare it zero according to the operator XX. If the condition is satisfied, jump to the instruction given by label L. XX $\in$ { eq, lt, le, ne, gt, ge, null, nonnull }
if_icmpXX L	Pop two values from the stack and compare against each other. Rest as above.
goto L	Unconditional jump to instruction given by the label L.
dup	Duplicate the word currently at the top of the stack.
pop	Discard the word currently at the top of the stack.
getfield #field	consume an object reference from stack, then dereference the field of that object given by (field,class) stored in the #field pointer in the constant pool and put the value of the field on the stack.
putfield #field	consume an object reference obj and a value v from the stack and store v it in the #field of obj.
instanceof #classId	consume an object reference from stack and push 1 onto the stack if it is an instance of the class pointed to by the offset #classId in the constant pool. Otherwise, push 0 onto the stack.
aload_#x	loads a reference onto the stack from a local variable #x
astore_#x	stores a reference at the top of the stack to a local variable #x