Computer Language Processing

Quiz Solution

Wednesday, November 27, 2019

This page intentionally left blank.

Exercise 1: Lexing (20 points)

Question 1.1

Circle all of the regular expressions whose language is **disjoint** from the language of $(\mathsf{a}|\mathsf{b})^+\mathsf{c}^*$:

A. $\mathsf{a^*b^*c^*}$

B. $\mathbf{c}^+$

C. $(\mathsf{b}|\mathsf{c})^+$

D. $(\mathbf{a}|\mathbf{b})^+\mathbf{c}^+\mathbf{a}^+$

Question 1.2

Consider the lexical analyzer with token classes defined by following regular expressions:

$$\begin{array}{lll} \langle\ \textsc{Imp} & := & \texttt{=>}, \\ \ \ \textsc{LessEq} & := & \texttt{<=}, \\ \ \ \textsc{Less} & := & \texttt{<}, \\ \ \ \textsc{Greater} & := & \texttt{>}\ \rangle \end{array}$$

Circle all the correct statements amongst the ones below, assuming the longest match rule is applied:

A. <=> will be tokenized to Less, Imp.

B. **<=< will be tokenized to LessEq, Less.**

C. **<==> will be tokenized to LessEq, Imp.**

D. <> will be tokenized to Greater, Less.

Exercise 2: Grammars & Parsing (20 points)

Consider the following grammar:

$\langle \mathit{Object} \rangle$ ::= ‘`{`’ $\langle \mathit{ObjectRest} \rangle$

$\langle \mathit{ObjectRest} \rangle$ ::= $\langle \mathit{ObjectContent} \rangle$ ‘`}`’ | ‘`{`’ ‘`}`’

$\langle \mathit{ObjectContent} \rangle$ ::= STRING ‘`:`’ $\langle \mathit{Value} \rangle$ $\langle \mathit{ObjectContentRep} \rangle$

$\langle \mathit{ObjectContentRep} \rangle$ ::= ε | ‘`,`’ $\langle \mathit{ObjectContent} \rangle$

$\langle \mathit{Array} \rangle$ ::= ‘`[`’ $\langle \mathit{ArrayRest} \rangle$

$\langle \mathit{ArrayRest} \rangle$ ::= $\langle \mathit{ArrayContent} \rangle$ ‘`]`’ | ‘`]`’

$\langle \mathit{ArrayContent} \rangle$ ::= $\langle \mathit{Value} \rangle$ $\langle \mathit{ArrayContentRep} \rangle$

$\langle \mathit{ArrayContentRep} \rangle$ ::= ε | ‘`,`’ $\langle \mathit{ArrayContent} \rangle$

$\langle \mathit{Value} \rangle$ ::= $\langle \mathit{Object} \rangle$ | $\langle \mathit{Array} \rangle$ | STRING | ‘`true`’ | ‘`false`’

Note that we do **not** have an end-of-file marker in the above grammar. Do **not** add one to the grammar. Do **not** assume such an implicit marker is present.

Question 2.1

Which of the following statements are true? Circle all correct answers. Hint: be bold!

A. **STRING is a terminal**.

B. ⟨*Object*⟩ is a terminal.

C. **⟨*ArrayContent*⟩ is a non-terminal**.

D. **`'true'` is a terminal**.

Question 2.2

Which of the following statements are true? Circle the correct answers.

A. ⟨*ObjectContent*⟩ is nullable.

B. ⟨*ArrayRest*⟩ is nullable.

C. **⟨*ArrayContentRep*⟩ is nullable.**

D. ⟨*Value*⟩ is nullable.

Question 2.3

Which of the following statements are true? Circle the correct answers.

A. FIRST(⟨*ObjectContent*⟩) contains `','`.

B. **First(⟨*Object*⟩) is disjoint from First(⟨*Array*⟩).**

C. FIRST(⟨*Value*⟩) is a subset of FIRST(⟨*Array*⟩).

D. **First(⟨*ArrayRest*⟩) contains `'['`.**

Question 2.4

Which of the following statements are true? Circle the correct answers.

A. FOLLOW(⟨*Object*⟩) is empty.

B. **Follow(⟨*ArrayContent*⟩) is equal to {`']'`}.**

C. **Follow(⟨*ArrayRest*⟩) contains `','`.**

D. FOLLOW(⟨*Value*⟩) contains `'{'`.

Question 2.5

Which of the following statements are true? Circle the correct answers.

A. The grammar is ambiguous.

B. **The grammar is LL(1).**

C. The grammar is in Chomsky normal form.

D. **The language defined by the grammar is non-regular.**

Exercise 3: L^* Membership-Checking Algorithm (20 points)

Let $L \subseteq A^*$ be a language given by a (terminating) algorithm `fL` : $A^* \to \{true, false\}$ such that, for all $w \in A^*$,

$$(\texttt{fL}(w) = true) \iff (w \in L)$$

In other words, `fL` checks whether a word belongs to L. (We do not have any other information about the language L.)

Let $S = L^*$.

Your goal

Write an algorithm `fS` using Scala-like notation to check if word belongs to S, that is, `fS` : $A^* \to \{true, false\}$ such that

$$(\texttt{fS}(w) = true) \iff (w \in S)$$

Your algorithm can inspect the word w as well as apply the function `fL` on arbitrary words. For full points, your solutions should only invoke `fL` a polynomial (in the size of the input word) number of times. You may use any auxiliary data structures from Scala library. For your convenience, a small API for `List` and `Array` follows.

API

Methods of `List[A]`

- `def apply(n: Int): A`: Returns the `n`th element of `this` list.
- `def take(n: Int): List[A]`: Returns first `n` elements of `this` list.
- `def drop(n: Int): List[A]`: Returns a copy of `this` list with the first `n` elements dropped.
- `def splitAt(n: Int): (List[A], List[A])`: Returns the pair `(this.take(n), this.drop(n))`.

Methods of `Array[A]`

- `def apply(n: Int): A`: Returns the `n`th element of `this` array.
- `def update(n: Int, x: A): Unit`: Updates the `n`th element of `this` array. Modifies the array.

Methods of the `Array` companion object

- `Array.tabulate[A](n: Int)(f: Int => A)`: Returns a new one-dimensional array of size `n`, initially populated by `f`.
- `Array.tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Array[Array[A]]`: Returns a new two-dimensional array of size `n1` $\times$ `n2`, initially populated by `f`.
- `Array.ofDim[A](n: Int): Array[A]`: Returns a new one-dimensional array of size `n`.
- `Array.ofDim[A](n1: Int, n2: Int): Array[Array[A]]`: Returns a new two-dimensional array of size `n1` $\times$ `n2`.

```
def fS[A](fL: List[A] => Boolean, w: List[A]): Boolean = {

  val n = w.size

  if (n == 0) {
    return true
  }

  // Make an array for every start position and length that will
  // record if the subsequence is a sentence of words from fL.
  val isSentence = Array.tabulate[Boolean](n, n + 1) { (i: Int, j: Int) =>
    // We initially only add single words to the array.
    fL(w.drop(i).take(j))
  }

  // Then, we populate the array by combining smaller subsequences into larger ones.
  for (j <- 2 to n) {  // j is the length of the subsequence.
    for (i <- 0 to (n - j)) {  // i is the index of the beginning of the subsequence.
      for (k <- 1 to (j - 1)) {  // k is the length of the first half.
        if (isSentence(i)(k) && isSentence(i+k)(j-k)) {
          isSentence(i)(j) = true
        }
      }
    }
  }

  // Check if the entire input is a sentence.
  isSentence(0)(n)
}
```

Or, even better:

```
def fS[A](fL: List[A] => Boolean, w: List[A]): Boolean = {

  val n = w.size

  // The following array contains, for every index,
  // true if we are allowed to start a word there.
  val isStartPos = Array.tabulate[Boolean](n + 1)(_ == 0)

  for (i <- 0 until n) {  // i is the index of the beginning of the word.
    if (isStartPos(i)) {  // Check if we can start from i.
      for (j <- (i + 1) to n) {  // j is the index just after the word.
        val candidate = w.drop(i).take(j - i)  // Get the candidate word.
        if (fL(candidate)) {  // Check if it is accepted by fL.
          // If it is the case, we record that we can start a word here.
          isStartPos(j) = true
        }
      }
    }
  }

  // Check if we can start right after the given input.
  isStartPos(n)
}
```

Exercise 4: Type Checking and Inference (20 points)

Consider the following typing rules for a simple language with integers, pairs and functions:

$$\frac{n \text{ is an integer literal}}{\Gamma \vdash n : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 * e_2 : \texttt{Int}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : (T_1, T_2)} \qquad \frac{\Gamma \vdash e : (T_1, T_2)}{\Gamma \vdash \mathit{fst}(e) : T_1} \qquad \frac{\Gamma \vdash e : (T_1, T_2)}{\Gamma \vdash \mathit{snd}(e) : T_2}$$

$$\frac{\Gamma \oplus \{(x, T_1)\} \vdash e : T_2}{\Gamma \vdash x \Rightarrow e : T_1 \Rightarrow T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \Rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1(e_2) : T_2} \qquad \frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$

Question 4.1

Consider the following type derivation, with type variables $\mathbf{T_1}, \dots, \mathbf{T_5}$, where $\Gamma_0 = \emptyset$ and $\Gamma = \{(x, \mathbf{T_2})\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{(x, \mathbf{T_2}) \in \Gamma}{\Gamma \vdash x : \mathbf{T_2}}}{\Gamma \vdash \mathit{fst}(x) : \mathbf{T_4}} \quad \dfrac{\dfrac{(x, \mathbf{T_2}) \in \Gamma}{\Gamma \vdash x : \mathbf{T_2}}}{\Gamma \vdash \mathit{snd}(x) : \mathbf{T_5}}}{\Gamma \vdash \mathit{fst}(x)(\mathit{snd}(x)) : \mathbf{T_3}}}{\Gamma_0 \vdash x \Rightarrow \mathit{fst}(x)(\mathit{snd}(x)) : \mathbf{T_1}}$$

Circle all the correct answers:

A. There are no assignments of $\mathbf{T_1}, \dots, \mathbf{T_5}$ such that the resulting derivation is valid.

B. In all valid derivations, $\mathbf{T_3}$ is equal to $\mathbf{T_5}$.

C. **There does not exist valid derivations where $\mathbf{T_1}$ is `Int`.**

D. **In all valid derivations, $\mathbf{T_2}$ is equal to $(\mathbf{T_4}, \mathbf{T_5})$.**

E. In all valid derivations, $\mathbf{T_3}$ is equal to $\mathbf{T_2} \Rightarrow \mathbf{T_1}$.

Question 4.2

For which of the following expressions does type inference using unification succeed? Circle the correct answers.

A. $\mathbf{y} \Rightarrow (\mathbf{x} \Rightarrow (\mathbf{x}, \mathbf{y}))$

B. $x \Rightarrow (y \Rightarrow (x(y) + y(x))$

C. $\mathbf{f} \Rightarrow (\mathbf{x} \Rightarrow \mathbf{f}(\mathbf{f}(\mathbf{x})))$

D. $f \Rightarrow (f(x \Rightarrow 4) + f(5))$

This page intentionally left blank.

Exercise 5: Designing a Type System (20 points)

Consider an expression language with a halving operator on even numbers. We are designing an operational semantics and a type system that ensures that we never half an odd number.

$\langle Expr \rangle$ ::= `half(` $\langle Expr \rangle$ `)` | $\langle Expr \rangle$ `+` $\langle Expr \rangle$ | INTEGER

The values of our language are all integers. We denote values by n and k.

Question 5.1

In this first question, we will design the operational semantics of our language. Semantics should define rule that apply to as many expressions as possible subject to the following constraints:

- Our operational semantics should not permit halving unless the value of an integer constant is even
- It should only perform evaluation of operands from left to right

Circle a **minimal** set of operational semantics rules that describe this behavior.

$$\frac{e \rightsquigarrow e'}{\texttt{half}(e) \rightsquigarrow e'}$$
(A) NO

$$\frac{n \text{ is a value} \quad n = 2k}{\texttt{half}(n) \rightsquigarrow k}$$
(B) YES

$$\frac{n \text{ is a value}}{\texttt{half}(n) \rightsquigarrow \lfloor \frac{n}{2} \rfloor}$$
(C) NO

$$\frac{\texttt{half}(e) \rightsquigarrow \texttt{half}(e')}{\texttt{half}(e) \rightsquigarrow e'}$$
(D) NO

$$\frac{e \rightsquigarrow e'}{\texttt{half}(e) \rightsquigarrow \texttt{half}(e')}$$
(E) YES

$$\frac{e' \rightsquigarrow \texttt{half}(e)}{\texttt{half}(e) \rightsquigarrow e'}$$
(F) NO

$$\frac{n_1 \text{ is a value} \quad n_2 \text{ is a value} \quad k = n_1 + n_2 \quad n_1 \text{ is odd} \quad n_2 \text{ is odd}}{n_1 \texttt{ + } n_2 \rightsquigarrow k}$$
(G) NO

$$\frac{e \rightsquigarrow e' \quad n \text{ is a value}}{n \texttt{ + } e \rightsquigarrow n \texttt{ + } e'}$$
(H) YES

$$\frac{e_2 \rightsquigarrow e_2'}{e_1 \texttt{ + } e_2 \rightsquigarrow e_1 \texttt{ + } e_2'}$$
(I) NO

$$\frac{n_1 \text{ is a value} \quad n_2 \text{ is a value} \quad k = n_1 + n_2 \quad n_1 \text{ is even} \quad n_2 \text{ is even}}{n_1 \texttt{ + } n_2 \rightsquigarrow k}$$
(J) NO

$$\frac{n_1 \text{ is a value} \quad n_2 \text{ is a value} \quad k = n_1 + n_2}{n_1 \texttt{ + } n_2 \rightsquigarrow k}$$
(K) YES

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \texttt{ + } e_2 \rightsquigarrow e_1' \texttt{ + } e_2}$$
(L) YES

Question 5.2

In this second part, we will design a type system for our language. The following expressions should type check:

```
4 + 5
half(2 + 4)
half(2) + 2
half(half(2) + half(2))
```

Circle a subset of the following rules that form a *sound* type system for our language: if a program type checks, then it must evaluate to a constant using the rules of operational semantics in the previous part.

Make sure that the above expressions can be typed. Do not include rules that are redundant with other rules you circled: if removing a rule does not decrease the set of programs that type check, then remove the rule.

$$\frac{\Gamma \vdash n : \texttt{Even}}{\Gamma \vdash \texttt{half}(n) : \texttt{Even}}$$
(A) NO

$$\frac{\Gamma \vdash n : \texttt{Integer}}{\Gamma \vdash \texttt{half}(n) : \texttt{Integer}}$$
(B) NO

$$\frac{\Gamma \vdash n : \texttt{Even}}{\Gamma \vdash \texttt{half}(n) : \texttt{Integer}}$$
(C) YES

$$\frac{n \text{ is an integer literal} \quad n \text{ is even}}{\Gamma \vdash n : \texttt{Even}}$$
(D) YES

$$\frac{n \text{ is an integer literal} \quad n \text{ is odd}}{\Gamma \vdash n : \texttt{Integer}}$$
(E) YES

$$\frac{\Gamma \vdash n : \texttt{Even}}{\Gamma \vdash n : \texttt{Integer}}$$
(F) YES

$$\frac{\Gamma \vdash n : \texttt{Integer}}{\Gamma \vdash n : \texttt{Even}}$$
(G) NO

$$\frac{}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Integer}}$$
(H) NO

$$\frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Integer}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Integer}}$$
(I) YES

$$\frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Even}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Even}}$$
(J) NO

$$\frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Even}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Integer}}$$
(K) NO

$$\frac{\Gamma \vdash e_1 : \texttt{Even} \quad \Gamma \vdash e_2 : \texttt{Even}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Even}}$$
(L) YES

$$\frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Integer}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Even}}$$
(M) NO

$$\frac{\Gamma \vdash e : \texttt{Integer}}{\Gamma \vdash e \texttt{ + } e : \texttt{Even}}$$
(N) YES