Computer Language Processing

Quiz Solution

Wednesday, November 27, 2019

Exercise	**Points**	**Points Achieved**
1	20	
2	20	
3	10	
4	30	
Total	80	

This page intentionally left blank.

Exercise 1: Regular Languages (20 points)

Consider the alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$.

Question 1.1

Which of the following regular expressions exactly describe the language of all words that contain an `a` immediately followed by `b`, and contain exactly one `c`. Circle letters next to **all** correct options.

A. `(ab)c(a|b)*`

No (incomplete).

B. `a*b*(ca*b*ab)*a*b*`

No

C. `(a|b|c)*ab(a|b|c)*`

No

D. `(a|b)*((ab(a|b)*c(a|b)*)|(c(a|b)*ab(a|b)*))`

Yes

E. `(a|b)*(ab|c)(a|b)*(ab|c)(a|b)*`

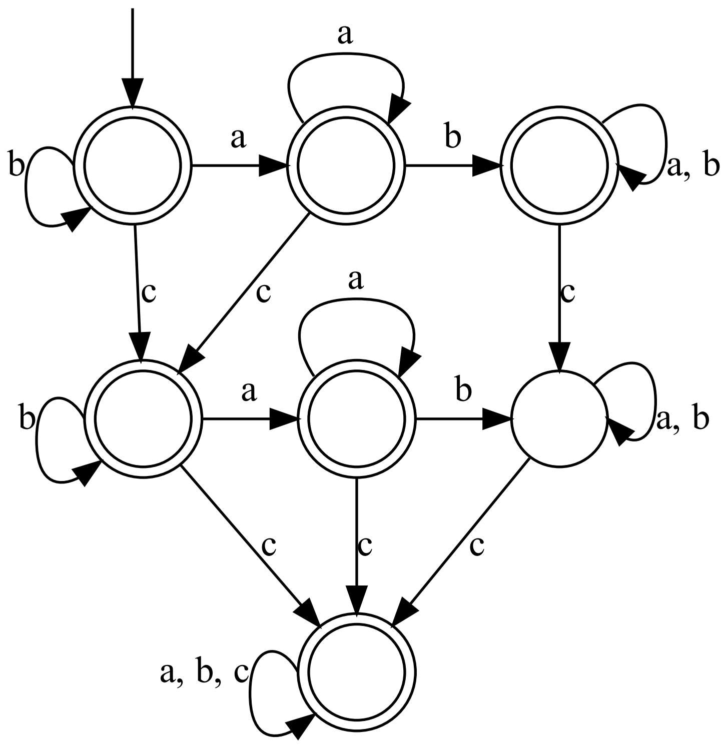
No

F. `((a|b)*ab(a|b)*c(a|b)*)|((a|b)*c(a|b)*ab(a|b)*)`

Yes

Question 1.2

Give a *minimal* deterministic finite-state automaton that describes the *complement* of the language from Question 1.1. Write down all the states and transitions explicitly.



Exercise 2: Context-free Grammars (20 points)

Consider the following context-free grammar, where `<E>` is the starting symbol:

$\langle E \rangle$ ::= $\langle ER \rangle$ $\langle D \rangle$

$\langle ER \rangle$ ::= E '`||`' | ε

$\langle D \rangle$ ::= $\langle B \rangle$ | $\langle DR \rangle$ $\langle B \rangle$

$\langle DR \rangle$ ::= $\langle B \rangle$ $\langle DS \rangle$

$\langle DS \rangle$ ::= '`&&`' $\langle DR \rangle$

$\langle B \rangle$ ::= '`true`' | '`false`' | $\langle O \rangle$ $\langle EC \rangle$

$\langle O \rangle$ ::= '`(`'

$\langle EC \rangle$ ::= $\langle E \rangle$ '`)`'

Question 2.1

Which of the following sequences are accepted by the grammar? Circle letters next to **all** correct options.

A. `true`

Yes

B. `true || false`

Yes

C. `( true && false )`

No

D. `true == false || true`

No

E. `( true || false || true ) || ( true || false )`

Yes

Question 2.2

Which of the following statements are true? Circle letters next to **all** correct options.

A. `<D>` is nullable.

No

B. `<DS>` is productive.

No

C. `)` is in `FOLLOW(<EC>)`.

Yes

D. `(` is in `FIRST(<B>)`.

Yes

Question 2.3

Which of the following statements are true? Circle letters next to **all** correct options.

A. The grammar is LL(1).

No

B. The language described by the grammar is regular.

No

C. The grammar is in Chomsky Normal Form (CNF).

No

D. The grammar contains infinitely many words.

Yes

Exercise 3: Type Inference (10 points)

Consider the following type system for a minimal language with anonymous functions and applications.

$$\frac{\Gamma[x \mapsto S] \vdash e : T}{\Gamma \vdash (x \Rightarrow e) : S \Rightarrow T} \qquad \frac{\Gamma \vdash e_1 : S \Rightarrow T \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1(e_2) : T} \qquad \frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$

Like in Scala, application has a priority over anonymous function creation, so that, for example, $x \Rightarrow (y \Rightarrow y)(x)$ denotes $x \Rightarrow ((y \Rightarrow y)(x))$.

For each of the following expressions, determine the result of type inference via unification. That is, state whether a most general type can be inferred, and if so, **write it out**.

E.g., for $(x \Rightarrow x)$ the answer is *yes*, and its most general type is $A \Rightarrow A$.

A. $(x \Rightarrow (y \Rightarrow y)(x)\ )$

Yes, $A \Rightarrow A$

B. $(f \Rightarrow (x \Rightarrow f(f(x))\ )\ )$

Yes, $(A \Rightarrow A) \Rightarrow A \Rightarrow A$

C. $(f \Rightarrow (x \Rightarrow (g \Rightarrow f(g(x))\ )(x)\ )\ )$

No; To see why the occurs check must fail at some point during unification, note that in $(g \Rightarrow f(g(x)))(x)$ we are essentially applying x to x.

D. $(f \Rightarrow (g \Rightarrow (x \Rightarrow f(x)(g(x))\ )\ )\ )$

Yes, $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$

E. $(f \Rightarrow (g \Rightarrow (x \Rightarrow g(f(x))\ )\ )\ )$

Yes, $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C$

Exercise 4: Lowering between Typed Languages (30 points)

Consider the following simple language, $\mathcal{L}^H$ ("L-high"), featuring integer constants and lists of integers:

$$\langle v \rangle := n \mid \text{nil} \mid n :: \langle v \rangle \qquad \text{(values)}$$

$$\langle e \rangle := \langle v \rangle \mid x \mid \textbf{let } x = \langle e \rangle; \langle e \rangle \mid \textbf{given } h :: t = \langle e \rangle \textbf{ then } \langle e \rangle \textbf{ else } \langle e \rangle \qquad \text{(expressions)}$$

where h, t and x are variables and n stands for integer constants.

The language's operational semantics is given by the following rules:

$$\frac{e_1 \rightsquigarrow e_1'}{(\textbf{let } x = e_1;\ e_2) \rightsquigarrow (\textbf{let } x = e_1';\ e_2)} \qquad \frac{}{(\textbf{let } x = v;\ e_2) \rightsquigarrow [x \mapsto v]e_2} \qquad \frac{e_1 \rightsquigarrow e_1'}{(e_1 :: e_2) \rightsquigarrow (e_1' :: e_2)}$$

$$\frac{e_1 \rightsquigarrow e_1'}{(\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow (\textbf{given } h :: t = e_1' \textbf{ then } e_2 \textbf{ else } e_3)} \qquad \frac{e_2 \rightsquigarrow e_2'}{(n :: e_2) \rightsquigarrow (n :: e_2')}$$

$$\frac{}{(\textbf{given } h :: t = \text{nil} \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow e_3} \qquad \frac{}{(\textbf{given } h :: t = v_1 :: v_2 \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow [h \mapsto v_1][t \mapsto v_2]e_2}$$

The types and (partial) typing rules of $\mathcal{L}^H$ are as follows:

$$\langle T \rangle ::= \texttt{Int} \mid \texttt{List} \qquad \text{(types)}$$

$$\frac{n \text{ is an integer literal}}{\Gamma \vdash^H n : \texttt{Int}} \qquad \frac{(x, T) \in \Gamma}{\Gamma \vdash^H x : T} \qquad \frac{}{\Gamma \vdash^H \text{nil} : \texttt{List}}$$

$$\frac{\Gamma \vdash^H e_1 : \texttt{Int} \quad \Gamma \vdash^H e_2 : \texttt{List}}{\Gamma \vdash^H (e_1 :: e_2) : \texttt{List}} \qquad \frac{\Gamma \vdash^H e_1 : S \quad \Gamma[x \mapsto S] \vdash^H e_2 : T}{\Gamma \vdash^H (\textbf{let } x = e_1;\ e_2) : T}$$

Question 4.1 (3 points)

What values do the following expressions evaluate down to?

1. $(\textbf{let } xs = 1 :: \text{nil};\ 2 :: xs) \overset{*}{\rightsquigarrow} 2 :: 1 :: \text{nil}$
2. $(\textbf{given } h :: t = 3 :: (4 :: \text{nil}) \textbf{ then } h :: (h :: t) \textbf{ else } \text{nil}) \overset{*}{\rightsquigarrow} 3 :: 3 :: 4 :: \text{nil}$
3. $(\textbf{let } as = 1 :: \text{nil};\ (\textbf{given } a :: bs = as \textbf{ then } (\textbf{given } b :: cs = bs \textbf{ then } 2 \textbf{ else } 3) \textbf{ else } 4) :: as) \overset{*}{\rightsquigarrow} 3 :: 1 :: \text{nil}$

Question 4.2 (4 points)

The type system above is missing a rule for **given** expressions. Complete the rule below such that the resulting type system is *sound* and all of the expressions in Question 4.1 are typable.

$$\frac{\Gamma \vdash^{H} e_1 : \texttt{List} \quad \Gamma[h \mapsto \texttt{Int}][t \mapsto \texttt{List}] \vdash^{H} e_2 : T \quad \Gamma \vdash^{H} e_3 : T}{\Gamma \vdash^{H} (\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3) : T}$$

Now consider another language, $\mathcal{L}_L$ ("L-low"), which is equivalent to $\mathcal{L}^H$ save for its operation on lists:

$$\langle v \rangle ::= n \mid \text{nil} \mid n :: \langle v \rangle \qquad \text{(values; as in } \mathcal{L}^H\text{)}$$

$$\langle e \rangle ::= \langle v \rangle \mid x \mid \textbf{let } x = \langle e \rangle; \langle e \rangle \mid \text{head}(\langle e \rangle) \mid \text{tail}(\langle e \rangle) \mid \textbf{ifNonEmpty}(\langle e \rangle) \textbf{ then } \langle e \rangle \textbf{ else } \langle e \rangle \qquad \text{(expressions)}$$

Its operational semantics is the same as $\mathcal{L}^H$'s for all common expressions. The new operations are characterized as follows:

$$\frac{e_1 \rightsquigarrow e_1'}{(\textbf{ifNonEmpty}(e_1) \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow (\textbf{ifNonEmpty}(e_1') \textbf{ then } e_2 \textbf{ else } e_3)}$$

$$\frac{}{(\textbf{ifNonEmpty}(\text{nil}) \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow e_3} \qquad \frac{e \rightsquigarrow e'}{\text{head}(e) \rightsquigarrow \text{head}(e')} \qquad \frac{}{\text{head}(v_1 :: v_2) \rightsquigarrow v_1}$$

$$\frac{}{(\textbf{ifNonEmpty}(v_1 :: v_2) \textbf{ then } e_2 \textbf{ else } e_3) \rightsquigarrow e_2} \qquad \frac{e \rightsquigarrow e'}{\text{tail}(e) \rightsquigarrow \text{tail}(e')} \qquad \frac{}{\text{tail}(v_1 :: v_2) \rightsquigarrow v_2}$$

In compilers programs typically undergo a series of transformations from higher-level to lower-level representations – this is often referred to as "lowering". We would like to define a lowering from $\mathcal{L}^H$ to $\mathcal{L}_L$, along with an appropriate type system for $\mathcal{L}_L$, so that lowered expressions remain well-typed. In particular, we want to define a translation function $[\![\cdot]\!]$ that maps each well-typed expression of $\mathcal{L}^H$ to a well-typed expression of $\mathcal{L}_L$. We will solve this problem by first choosing an appropriate set of typing rules for $\mathcal{L}_L$, and only then defining the translation function.

Question 4.3 (3 points)

In $\mathcal{L}^H$ list-operations could only get stuck when invoked on non-lists, e.g. $\textbf{given } h :: t = 1 \textbf{ then } 2 \textbf{ else } 3$.

In $\mathcal{L}_L$ we additionally have list-operations that can get stuck when invoked on a list. Give an example of one such stuck expression in $\mathcal{L}_L$.

E.g., head(nil) or tail(nil)

Question 4.4 (10 points)

We define a type system for $\mathcal{L}_L$. We use the types of $\mathcal{L}^H$, along with a new type of non-empty lists `NEList`:

$\langle T \rangle ::= \texttt{Int} \mid \texttt{List} \mid \texttt{NEList}$ (types)

Circle a subset of the following typing rules, such that:

- all of the following expressions are **typable**:
  - $\text{head}(0 :: 1 :: \text{nil})$
  - $\textbf{let } k = 2;\ \textbf{ifNonEmpty}(\text{tail}(3 :: \text{nil}))\ \textbf{then } k\ \textbf{else } 4$
  - $\textbf{let } ys = \text{nil};\ \textbf{ifNonEmpty}(ys)\ \textbf{then } \text{head}(ys) :: k :: \text{tail}(ys)\ \textbf{else } k :: \text{nil}$
- the type system is **sound** with respect to $\mathcal{L}_L$'s operational semantics (i.e., evaluation can't get stuck),
- the subset is **minimal** (i.e., no typing rule is redundant).

$$\boxed{\frac{n \text{ is an integer literal}}{\Gamma \vdash_L n : \texttt{Int}}} \qquad \boxed{\frac{(x, T) \in \Gamma}{\Gamma \vdash_L x : T}} \qquad \boxed{\frac{\Gamma \vdash_L e_1 : S \quad \Gamma[x \mapsto S] \vdash_L e_2 : T}{\Gamma \vdash_L (\textbf{let } x = e_1;\ e_2) : T}}$$

$$\boxed{\frac{}{\Gamma \vdash_L \text{nil} : \texttt{List}}} \qquad \frac{}{\Gamma \vdash_L \text{nil} : \texttt{NEList}} \qquad \frac{\Gamma \vdash_L e : \texttt{List}}{\Gamma \vdash_L e : \texttt{NEList}} \qquad \boxed{\frac{\Gamma \vdash_L e : \texttt{NEList}}{\Gamma \vdash_L e : \texttt{List}}}$$

$$\frac{\Gamma \vdash_L e_1 : \texttt{Int} \quad \Gamma \vdash_L e_2 : \texttt{NEList}}{\Gamma \vdash_L (e_1 :: e_2) : \texttt{List}} \qquad \boxed{\frac{\Gamma \vdash_L e_1 : \texttt{Int} \quad \Gamma \vdash_L e_2 : \texttt{List}}{\Gamma \vdash_L (e_1 :: e_2) : \texttt{NEList}}} \qquad \frac{\Gamma \vdash_L e_1 : \texttt{Int} \quad \Gamma \vdash_L e_2 : \texttt{List}}{\Gamma \vdash_L (e_1 :: e_2) : \texttt{List}}$$

$$\frac{\Gamma \vdash_L e : \texttt{NEList}}{\Gamma \vdash_L \text{head}(e) : \texttt{List}} \qquad \boxed{\frac{\Gamma \vdash_L e : \texttt{NEList}}{\Gamma \vdash_L \text{head}(e) : \texttt{Int}}} \qquad \frac{\Gamma \vdash_L e : \texttt{List}}{\Gamma \vdash_L \text{tail}(e) : \texttt{List}} \qquad \boxed{\frac{\Gamma \vdash_L e : \texttt{NEList}}{\Gamma \vdash_L \text{tail}(e) : \texttt{List}}}$$

$$\boxed{\frac{\Gamma \vdash_L e_1 : \texttt{List} \quad \Gamma \vdash_L e_2 : T \quad \Gamma \vdash_L e_3 : T}{\Gamma \vdash_L (\textbf{ifNonEmpty}(e_1)\ \textbf{then } e_2\ \textbf{else } e_3) : T}} \qquad \boxed{\frac{\Gamma \vdash_L x : \texttt{List} \quad \Gamma[x \mapsto \texttt{NEList}] \vdash_L e_2 : T \quad \Gamma \vdash_L e_3 : T}{\Gamma \vdash_L (\textbf{ifNonEmpty}(x)\ \textbf{then } e_2\ \textbf{else } e_3) : T}}$$

$$\frac{\Gamma[x \mapsto \texttt{NEList}] \vdash_L x : \texttt{List} \quad \Gamma[x \mapsto \texttt{NEList}] \vdash_L e_2 : T \quad \Gamma \vdash_L e_3 : T}{\Gamma \vdash_L (\textbf{ifNonEmpty}(x)\ \textbf{then } e_2\ \textbf{else } e_3) : T}$$

$$\frac{\Gamma \vdash_L x : \texttt{List} \quad \Gamma[x \mapsto \texttt{NEList}] \vdash_L e_2 : T \quad \Gamma[x \mapsto \texttt{NEList}] \vdash_L e_3 : T}{\Gamma \vdash_L (\textbf{ifNonEmpty}(x)\ \textbf{then } e_2\ \textbf{else } e_3) : T}$$

Hint: Remember that e refers to all expressions, whereas x only refers to variables!

Question 4.5 (10 points)

Finally, we will construct a translation function $[\![\cdot]\!]$ which maintains the result of expressions, i.e.,

$$\forall e \in \mathcal{L}^H, v. \quad e \overset{*}{\rightsquigarrow} v \quad \Rightarrow \quad [\![e]\!] \overset{*}{\rightsquigarrow} v$$

Furthermore, expressions well-typed in $\mathcal{L}^H$ should remain well-typed in $\mathcal{L}_L$ after translation:

$$\forall \Gamma, e \in \mathcal{L}^H. \quad \exists T.\, \Gamma \vdash^H e : T \quad \Rightarrow \quad \exists T'.\, \Gamma \vdash_L [\![e]\!] : T'$$

Ensure that the size of each rewritten expression $[\![e]\!]$ is linear in the size of the original expression e.

The first three rules for variables, integer constants and nil, are already given to you:

$[\![x]\!] := x$ $\qquad$ $[\![n]\!] := n$ $\qquad$ $[\![\text{nil}]\!] := \text{nil}$

Circle a subset of rules forming a total function $[\![\cdot]\!] : \mathcal{L}^H \to \mathcal{L}_L$ satisfying the above constraints.

$\boxed{[\![\textbf{let } x = e_1;\ e_2]\!] := (\textbf{let } x = [\![e_1]\!];\ [\![e_2]\!])}$ $\qquad$ $[\![\textbf{let } x = e_1;\ e_2]\!] := (\textbf{let } x = e_1;\ [\![e_2]\!])$

$[\![\textbf{let } x = e_1;\ e_2]\!] := [x \mapsto [\![e_1]\!]]\, [\![e_2]\!]$

$[\![n :: e]\!] := (n :: e)$ $\qquad$ $[\![n :: e]\!] := (n :: [\![e]\!])$ $\qquad$ $[\![e_1 :: e_2]\!] := (e_1 :: [\![e_2]\!])$ $\qquad$ $\boxed{[\![e_1 :: e_2]\!] := ([\![e_1]\!] :: [\![e_2]\!])}$

$[\![\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] := (\textbf{given } h :: t = [\![e_1]\!] \textbf{ then } [\![e_2]\!] \textbf{ else } [\![e_3]\!])$

$[\![\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] := (\textbf{let } x = [\![e_1]\!];\ \textbf{given } h :: t = x \textbf{ then } [\![e_2]\!] \textbf{ else } [\![e_3]\!])$

$[\![\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] := (\textbf{ifNonEmpty}([\![e_1]\!]) \textbf{ then } [\![e_2]\!] \textbf{ else } [\![e_3]\!])$

$[\![\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] := (\textbf{ifNonEmpty}([\![e_1]\!]) \textbf{ then } [h \mapsto \text{head}([\![e_1]\!])][t \mapsto \text{tail}([\![e_1]\!])]\, [\![e_2]\!] \textbf{ else } [\![e_3]\!])$

$\boxed{[\![\textbf{given } h :: t = e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] := (\textbf{let } x = [\![e_1]\!];\ \textbf{ifNonEmpty}(x) \textbf{ then } [h \mapsto \text{head}(x)][t \mapsto \text{tail}(x)]\, [\![e_2]\!] \textbf{ else } [\![e_3]\!])}$

(where $[x \mapsto e_1]e_2$ is the substitution of all x in e_2 by e_1, as usual.)