EPFL Computer Language Processing Exam (CS-320)

November 2022

Keep in mind the following:

1. **The exam is from 13:15 to 15:00. Do not open the exam until we tell you to.**
2. **Place your CAMIPRO card on your desk.**
3. **Put all electronic devices in a bag away from the bench.**
4. **Write your final answers using a permanent pen (no graphite, no "frixion" pen).**
5. **The only written material you are allowed to use in the exam is one sheet (two A4 pages) of any content that you prepare, either hand-written or printed.**
6. **Fill in all your answers on the given exam sheet. Do not submit additional sheets. Do not unstaple the sheets.**
7. Each question is scored independently. When you need to circle answers, then circling the correct set of answers gives you full points. If you circle some of the wrong answers or do not circle all the correct answers, you will obtain zero points on that question.
8. We advise you to first solve questions that you find easier.
9. The maximal number of points on the exam is **30**.

How to circle answers:

(1.) One correct solution; use hollow circle to circle the number only

2. Wrong answer; leave as is

(3.) Another correct solution; use hollow circle to circle the number only

4. Another wrong answer; leave as is

Write your full name on the line below:

_____

Write your SCIPER on the line below:

_____

1 Longest-Match Rule (2pt)

Consider the lexical analyzer accepting sequences over the alphabet $A = \{\mathbf{0}, \mathbf{1}, \mathbf{.}\}$, with token classes defined by following regular expressions (where float constants have only one decimal place):

float ::= $(\mathbf{0} \mid \mathbf{1})^*$ **.** $(\mathbf{0} \mid \mathbf{1})^?$

int ::= **1** $(\mathbf{0} \mid \mathbf{1})^*$

zero ::= **0**

Note that in the notation above, the parentheses), (as well as *, $^?$, | are not part of the input alphabet but part of the notation for regular expressions.

Circle numbers next to correct statements regarding the rules above, assuming the longest match rule is applied:

1. **100.00** will be tokenized to sequence **float**
2. **100.00.1** will be tokenized to sequence **float, zero, float**
3. **10.10** will be tokenized to sequence **float, int**
4. **011.0** will be tokenized to sequence **zero, float**

2 Epsilon Closure (4pt)

Consider the following representation of a NFA (non-deterministic finite automaton) in Scala

```
abstract class Transition
type State = Int
case class Epsilon(to: State) extends Transition
case class CharTransition(char: Char, to: State) extends Transition
case class NFA(initial: State, accepting: Set[State], delta: Map[State, Set[Transition]]) {
    def transitions(from: State): Set[Transition] = delta.getOrElse(from, Set.empty)

    def successors(from: State, visited: Set[State]): Set[State] =
        transitions(from).collect { case Epsilon(to) if !visited.contains(to) ⇒ to }

    def epsilonClosure(states: Set[State]) =
      def epsilonClosureRec(states: Set[State], acc: Set[State]): Set[State] = ???
      epsilonClosureRec(states, Set.empty)
}
```

The alphabet of our NFA is the Char type in Scala. Each state in the NFA is represented by an integer and a set of transitions to other states. A transition results in a destination state and is labeled either by ε or a character c : Char. We would like to complete the function epsilonClosure (which finds states reachable through epsilon transitions), so that it matches the definition seen in class. Circle all correct implementations of epsilonClosureRec among these, if any:

1. ```
   def epsilonClosureRec(states: Set[State], acc: Set[State]): Set[State] =
     states.foldLeft(acc ++ states)((foldAcc, state) ⇒
       epsilonClosureRec(successors(state, foldAcc), foldAcc))
   ```

2. ```
   def epsilonClosureRec(states: Set[State], acc: Set[State]): Set[State] =
     states.foldLeft(acc ++ states)((foldAcc, state) ⇒
       epsilonClosureRec(successors(state, acc), foldAcc))
   ```

Now circle all correct implementations among these two (if any):

1. ```
   def epsilonClosureRec(states: Set[State], acc: Set[State]): Set[State] =
     val expansion = states.flatMap(state ⇒ successors(state, acc))
     if expansion.size > 0 then epsilonClosureRec(expansion, acc ++ states) else acc ++ states
   ```

2. ```
   def epsilonClosureRec(states: Set[State], acc: Set[State]): Set[State] =
     val expansion = states.flatMap(state ⇒ successors(state, acc))
     if expansion.size > 0 then epsilonClosureRec(expansion, acc ++ states) else acc
   ```

See next page for excerpts from the standard library reference for foldLeft and collect.

- **def** foldLeft[B](z: B)(op: (B, A) ⇒ B): B

  Applies a binary operator to a start value and all elements of this set, going left to right.

  - **B** the result type of the binary operator.
  - **z** the start value.
  - **op** the binary operator.
  - **returns** the result of inserting op between consecutive elements of this set, going left to right with the start value z on the left: op(...op(z, x1), x2, ..., xn) where x1, ..., xn are the elements of this set. Returns z if this set is empty.

- **def** collect[B](pf: PartialFunction[A, B]): Set[B]

  Builds a new set by applying a partial function to all elements of this set on which the function is defined.

  - **B** the element type of the returned set.
  - **pf** the partial function which filters and maps the set. It acts both as a filter (by filtering out elements for which the function is not defined) and as a mapping.
  - **returns** a new set resulting from applying the given partial function pf to each element on which it is defined and collecting the results. The order of the elements is preserved.

LL(1) Parsing

First, consider the following grammar with non-terminals S,A and terminals **EOF**,), (,], [.

S ::= A **EOF**

A ::= (A) A | A [A] | ε

3 Circle all true statements about the grammar above (2pt)

1. "[]()([])" is accepted by the grammar
2. The grammar is LL(1)
3. The grammar is ambiguous
4. NULLABLE(A) == true
5. NULLABLE(S) == true

4 Circle the correct answer (1pt)

1. FIRST(S) == { EOF }
2. FIRST(S) == { (, [}
3. FIRST(S) == { (,) , EOF }
4. FIRST(S) == { (, [, EOF }
5. FIRST(S) == { (,), [,] , EOF }

5 Circle the correct answer (1pt)

1. FOLLOW(A) == {),] }
2. FOLLOW(A) == {),] , EOF }
3. FOLLOW(A) == { (, [,) ,] }
4. FOLLOW(A) == { (, [,], EOF }
5. FOLLOW(A) == { (, [,) ,], EOF }

6 Fill the LL(1) parsing table for the above grammar (2pt)

	EOF	(	)	[	]
S					
A					

Now consider the following **different grammar**:

S ::= B **EOF**

B ::= (B) B | **empty**

where **empty** is a special keyword.
A student writes a parser in Scala for this language and implements the following case classes

case class B(children: Either[(parOpenToken, B, parCloseToken, B),
EmptyKW])

where parOpenToken, parCloseToken, EmptyKW are types representing the tokens $($, $)$, **empty**.

7 Circle all the correct answers about the above class B (1pt)

1. The case class B *is suitable* as a node of an *abstract syntax tree (AST)* because it can be traversed recursively and is the most abstract representation sufficient to interpret and compile the program.
2. The case class B *is suitable* as a node of a *parse tree* because the children of each node correspond to the right-hand side of the corresponding grammar rule.
3. The case class B *is not suitable* as a node of a parse tree because it is too abstract for this purpose.
4. The case class B *is not suitable* as a node of an *abstract syntax tree (AST)* because it contains unnecessary terminals.

Now consider the **following grammar**:

S ::= C **EOF**

C ::= ⟨IntegerLiteral⟩ + C | ⟨IntegerLiteral⟩

with terminals ⟨IntegerLiteral⟩, +, **EOF**.
The student implements the following case class:

case class C(children:Either[(Int, C), Int])

8 Circle all correct answers about the above case class C (1pt)

1. The class C is suitable as a node of an abstract syntax tree (AST) because it represents the information about the program that suffices to interpret and compile it.
2. The class C is suitable as a node of the parse tree because the children of each node correspond to the right-hand side of the corresponding grammar rule.
3. The class C is not suitable as a node of the AST because it contains unnecessary terminals.
4. The class C does not represent a parse tree node because it does not represent all terminals.

9 Properties of Grammars (1pt)

Circle all correct answers.

S ::= (S) R ::= (T) | T T ::= X or X | X Z X | ε X ::= true | false | (X) Y ::= or Z ::= and	1. There are exactly 3 unproductive symbols 2. There is exactly 1 unreachable symbol 3. There are exactly 3 unit productions 4. There is exactly 1 unit production 5. There are exactly 2 unproductive symbols 6. There are exactly 4 unreachable symbols 7. There are exactly 5 unreachable symbols 8. Z is unreachable

Terminals are:), (, or, true, false, and. Starting symbol is S.

10 Is this grammar in Chomsky Normal Form (CNF)? (1pt)

Circle all correct answers.

S ::= ε | M N | O C S ::= ⟨IntegerLiteral⟩ O ::= (C ::=) M ::= $-$ N ::= ⟨IntegerLiteral⟩	1. No, because some rules do not have a valid number of symbols on the right-hand side 2. Yes, because only the starting symbol has ε on the right hand side and every other right hand side either has exactly two non-nonterminals, or exactly one terminal. 3. No, because the starting symbol has an ε 4. No, because there is a unit production 5. No, because integer literals are not allowed in CNF 6. No, because the starting symbol should not have a terminal on its right-hand side

Terminals are:), (, $-$, ⟨IntegerLiteral⟩. Starting symbol is S.

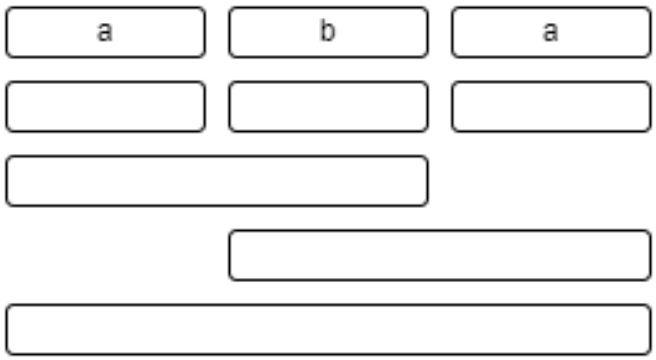
11 Is this grammar in Chomsky Normal Form (CNF)? (1pt)

Circle all correct answers.

S ::= ε | C C ::= C_1 B C_1 ::= A X X ::= true | false A ::= [B ::=]	1. No, because some rules do not have a valid number of symbols on the right-hand side 2. Yes, because every rule has at most 2 symbols on the right-hand side, all terminals have an associated non-terminal, only the starting symbol is nullable, and there are no unit productions, unproductive or unreachable symbols 3. No, because there is a unit production 4. No, because not all terminals are on the right-hand side 5. No, because the starting symbol has an ε

Terminals are:], [, true, false. Starting symbol is S.

12 Fill the CYK parse table for the word "aba" and the following grammar. (2pt)

S ::= A B | t_1 A | c t_1 ::= a A ::= t_1 A | c B ::= t_2 B | b t_2 ::= b	

Terminals are: a, b, c.

13 What grammars could we obtain from the Initial Grammar after reducing the number of symbols on all right-hand sides to at most two? (2pt)

Assume the algorithm that introduces no more non-terminals than the one presented in the lecture. Circle numbers for all grammars that could be obtained.

Initial Grammar: S ::= (L) | a L ::= L , S | S	1. Grammar: S ::= X | a L :: = L V | S V = , S X ::= (L)	2. Grammar: S ::= O L C | a L ::= L V S | S O = (L C =) V = ,
3. Grammar: S ::= O C | a L ::= V S | S O = (L C =) V = L ,	4. Grammar: S ::= Y | a L ::= J | S X = (L Y = X) Z = L , J = Z S	5. Grammar: S ::= X | a L :: V S | S V = L , X = O C O = (C = L K K =)

Terminals are: a,), (and the comma **,** .

14 Typing Rules (3pt)

Complete (on the next page) the type derivation for the body of the function f.

```
def f(x: Int, u: Int, v: Int): Int = {
    if (x < u) {
        u
    } else if (v < x) {
        v
    } else {
        x
    }
}
```

Use the types rules shown below.

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 + e_2 : Int} \qquad \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 \ \star \ e_2 : Int}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 \ \texttt{\&\&} \ e_2 : Bool} \qquad \frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 \ \texttt{||} \ e_2 : Bool}$$

$$\frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 < e_2 : Bool} \qquad \frac{\Gamma \vdash b : Bool \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \textbf{if } (b) \ e_1 \ \textbf{ else } e_2 \ : T}$$

$$\dfrac{\dfrac{\dfrac{(x, Int) \in \Gamma}{\Gamma \vdash x : Int} \quad \dfrac{(u, Int) \in \Gamma}{\Gamma \vdash u : Int}}{\Gamma \vdash x < u : Bool} \quad \dfrac{(u, Int) \in \Gamma}{\Gamma \vdash u : Int} \quad \dfrac{\dfrac{\dfrac{(\quad,\quad) \in \Gamma}{\Gamma \vdash \quad :} \quad \dfrac{(\quad,\quad) \in \Gamma}{\Gamma \vdash \quad :}}{\Gamma \vdash \quad :} \quad \dfrac{(\quad,\quad) \in \Gamma}{\Gamma \vdash \quad :} \quad \dfrac{(\quad,\quad) \in \Gamma}{\Gamma \vdash \quad :}}{\Gamma \vdash \quad :}}{\Gamma \vdash \textbf{if}(x < u)u \ \textbf{else if}(v < x)v \ \textbf{else} \ x : Int}$$

15 Type Inference (3pt)

For which of the following expressions does type inference using unification succeed? For the + operator, assume the type rules as in the previous question. Circle the correct answers.

1. x ⇒ y ⇒ y(z ⇒ 6) + y(7)
2. g ⇒ f ⇒ x ⇒ g(f(x))
3. x ⇒ y ⇒ ((z ⇒ y), y)
4. g ⇒ f ⇒ x ⇒ g(f(x)) + f(g(x)) + x

16 Unification Algorithm (3pt)

Consider a programming language with pairs and the usual typing rules, as in the lecture. Apply the unification algorithm on the following function:

```
def swap(t) = {
    (t._2, t._1)
}
```

assuming the following type variables assigned to tree nodes:

((t: τ)._2: τ_1, (t: τ)._1: τ_2): τ_3

Write each step of the unification algorithm, mentioning what rule of the algorithm you are applying. We provide you with the initial step.

1. $\tau = (\tau_{10}, \tau_1)$
   $\tau = (\tau_2, \tau_{20})$
   $\tau_3 = (\tau_1, \tau_2)$
2.
3.
4.

Write down an expression for the argument (t) and return type of swap in terms of types τ_1 and τ_2:

(argument) t :

(result) swap(t) :