Quiz

CS-320, Computer Language Processing, Fall 2015

Wednesday, November 18th, 2015

Version 1

General notes about this quiz

- Have your CAMIPRO card ready on the desk.
- You are allowed to use any printed material (using standard fonts, 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.
- Use separate sheets, for each question, to write your answers. No sheet of paper should contain answer to two or more questions at the same time.
- 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	

Prelude. In this quiz, you consider the scenario where you are hired by a new Swiss start-up company that aims to disrupt the smartwatch market. The main product is a wearable device that runs apps in a variant of HTML5 so energy-efficiently that it can be powered solely by absorbing the heat of the person wearing it. You are in charge of building the compiler for this platform. You soon discover that using off-the-shelf regular expression libraries depletes a fully charged watch battery before processing `<title>Textme</title>`, so you need to build the compiler making use of what you learned in the class.

Problem 1: Lexer for XML (25 points)

The goal of this exercise is to develop a lexer for XML. The input to the lexer is a stream of characters belonging to the set $\Sigma = \{a, \cdots, z, A, \cdots Z, 0, \cdots, 9, <, >, /, \{, :, \}, '', !, -, WS\}$, where *WS* denotes a white space character. As a shorthand, we will use *letter* to denote any character in $\{a, \cdots, z, A, \cdots Z\}$, *digit* to denote any of $0, \cdots, 9$, and *special* to denote any of {, }, :. (Note that *special* does not include other non-alpha-numeric characters like $<, >, '', !, -$.)
Your goal is to construct a lexer (i.e, an automaton) that tokenizes the input stream into the tokens listed below. The tokens are shown on the left and their defining regular expressions are shown on the right.

Token name	**Regular expression**
OP	$<$
CL	$>$
OPSL	$< /$
CLSL	$/ >$
EQ	$=$
NAME	$letter(letter \mid digit)^*$
NONNAME	$(digit \mid special)(letter \mid digit \mid special)^*$
STRING	$''(letter \mid digit \mid special)^*''$
COMMENT	$<! - - (letter \mid digit \mid special)^* - - >$
SKIP	*WS*

a) **[15 pts]** Construct the labelled DFA described in the lectures for the tokens define above. Note that every final state should be labelled by the token class(es) it accepts.

Consider the following XML string.

```
<jsonmessage>
   <!--CommunicationOfJSonObjects-->
   <from ip="">EPFLserver</from>
   <message>{"field":1}</message>
</jsonmessage>
```

b) **[10 pts]** Show the list of tokens that should be generated by the lexer for the above XML string. You should use the *Maximum Munch Rule* to tokenize. You **need not** show SKIP tokens, which correspond to whitespaces.

Problem 2: Parser for XML (25 points)

In this exercise, your are required to design an LL(1) parser for XML. Consider the following grammar that describes syntactically correct XML strings. The terminals of the grammar are the tokens that would be generated by the lexer (which you designed in the previous exercise).

element	→	OP NAME attributes CL content OPSL NAME CL
	|	OP NAME attributes CLSL
attributes	→	attributes attr | ϵ
attr	→	NAME EQ STRING
content	→	textOpt | body
textOpt	→	NAME | NONNAME | STRING | ϵ
body	→	body elemOrComment | elemOrComment
elemOrComment	→	element | COMMENT

Remark: the above grammar is a simplified version of the Antlr v4 XML grammar available at `https://github.com/antlr/grammars-v4/xml/XMLParser.g4`

a) [25 pts] Convert the above grammar to an equivalent grammar that is LL(1). Show the *Nullable* non-terminals of **your** grammar, and the *First* and *Follow* sets of each non-terminal.

Solution

The following three non-terminals have productions that violate the LL(1) property:
(a) `element` - it has two right-hand-sides whose First symbols are intersecting
(b) `attributes` - it has left-recurison (which implies that its right-hand-sides will have intersecting first symbols)
(c) `body` - it also has left-recursion.

To transform the productions of `element` to LL(1), we only have to do left-factoring, which would yield:

element	→	OP NAME attributes suffix
suffix	→	CL content OPSL NAME CL | CLSL

Now, consider the non-terminal `attributes`. Its purpose is to generate 'attr' zero or more times. In other words, `attributes` → (`attr`)*. Therefore, we can replace the left-recursive rule by the equivalent right-recursive rule: `attributes` → `attr attributes` | ϵ.

We can apply a similar reasoning to convert the productions of `body`. It is easy to see that `body` generates one or more `elemOrComment`. An equivalent right-recursive rule could be: `body` → `elementOrComment body` | `elementOrComment`. However, these productions still have intersecting first symbols, so we left-factor them again as follows:

body	→	elementOrComment bodySuf
bodySuf	→	body | ϵ

Problem 3: CNF grammar for JSON (25 points)

Consider the following grammar that accepts JSON (JavaScript Object Notation) strings.

json	$\rightarrow$	object
object	$\rightarrow$	{ pairs } | { }
pairs	$\rightarrow$	pairs STRING : value | ϵ
value	$\rightarrow$	STRING | NUMBER | object

a) [10 pts] Convert the grammar to Chomsky's Normal Form. Recall that a grammar in CNF should satisfy the following properties.

1. terminals t occur alone on the right-hand side: X ::= t
2. no productions of arity more than two
3. no nullable symbols except for the start symbol
4. no single non-terminal productions X ::= Y
5. no unproductive non-terminal symbols
6. no non-terminals unreachable from the start symbol

It is sufficient if you only write the final grammar that you obtain.

b) [15 pts] Check using the CYK algorithm if the word: "{ STRING : { } }" can be parsed by the grammar. Below you can find the template of the CYK parse table for the string. Fill in the entries of the table. It is sufficient to show only the entries that correspond to a parse tree of the string.
Hint: You could manually come up with a parse tree of your CNF grammar for the word, and then fill in the required entries of the CYK table that corresponds to the parse tree.

Solution

Try these questions at `grammar.epfl.ch`.

Problem 4: Intersection Types (25 points)

In this exercise, we will consider the notion of intersection of types. Let T_1 and T_2 be two types belonging to our langauge. An expression has an intersection type $T_1 \wedge T_2$ iff it can be typed as both T_1 and T_2. Therefore, we have the following type rules.

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash e : T_2}{\Gamma \vdash e : T_1 \wedge T_2} \qquad \frac{\Gamma \vdash e : T_1 \wedge T_2}{\Gamma \vdash e : T_1} \qquad \frac{\Gamma \vdash e : T_1 \wedge T_2}{\Gamma \vdash e : T_2}$$

We consider $T_1 \wedge T_2$ and $T_2 \wedge T_1$ to be the same. In the above rules T_1 and T_2 can also be function types like $R \to S$.

a) **[5 pts]** If T_1 and T_2 are arbitrary types, consider the following three expressions denoting types: $T_1 \wedge T_2$, T_1, and T_2. State all subtyping relations that you believe should hold among the 3×3 possible pairs of expressions.

Enter `<:` if the type corresponding to the row is a subtype of the type corresponding to the column; enter / if this is not necessarily the case.

<:	T_1	T_2	$T_1 \wedge T_2$
T_1	<:	/	/
T_2	/	<:	/
$T_1 \wedge T_2$	<:	<:	<:

In the next part of the exercise, you are required to come up with a type derivation involving intersection types. Consider a language, similar to the one descried in lecturecise 12, with arithmetic operations, if-else expressions, assignment and block statements, that has the following types: Pos, Neg, Int and $Bool$. (We provide all the type rules that you may need for this exercise at the end of this question.)
Consider the function f shown below. Γ_0 is the initial type environment before the beginning of the function.

$\Gamma_0 = \{$ divk : $(Pos \to Pos) \wedge (Neg \to Neg)\}$

```
def f(x : Int) : Int {
  if(x > 0) divk(x)
  else
   if(x < 0) divk(x)
   else x
}
```

`divk` is a function (e.g. like $x => 10/x$) that maps positive integers to positive integers and negative integers to negative integers. Observe that with intersection types we can type the function as $(Pos \to Pos) \wedge (Neg \to Neg)$.

b) **[20 pts]** Complete the type derivation for the body of the function f, shown on page 5, by filling in the holes marked with ?. If the expression will not type check, show the step where the type derivation cannot proceed. You will need to use only the type rules of intersection types and the types rules shown below.

Solution

There are two correct derivations for this question, one of which is shown below.
Type Rules:

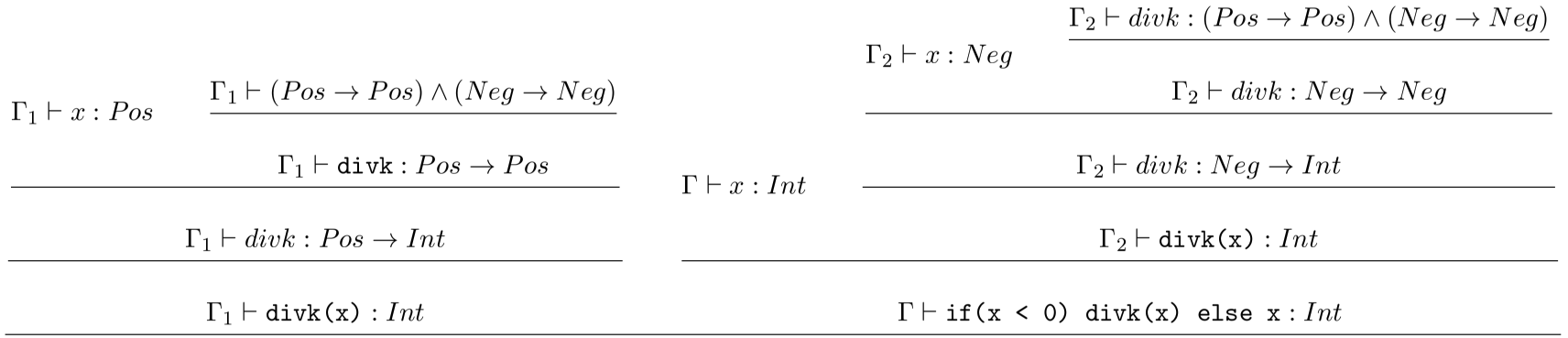
$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T} \qquad Pos <: Int \quad Neg <: Int \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

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

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \oplus \{(x, Pos)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(x > 0)\ e_1\ \text{ else }\ e_2 : T}$$

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \oplus \{(x, Neg)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(x < 0)\ e_1\ \text{ else }\ e_2 : T} \qquad \frac{\Gamma \vdash b : Bool \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(b)\ e_1\ \text{ else }\ e_2 : T}$$

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



$$\Gamma \vdash \texttt{if(x > 0) divk(x) else if(x < 0) divk(x) else x} : \mathit{Int}$$

where,

$\Gamma_1 = \Gamma \uplus \{(x, Pos)\}$

$\Gamma_2 = \Gamma \uplus \{(x, Neg)\}$ where, $\Gamma = \{(x, Int), divk : (Pos \to Pos) \wedge (Neg \to Neg)\}$