

Computational Syntax Exam

LT2214, University of Gothenburg, 4 June 2025

This exam has five questions. Each question is worth 12 points. You need 30 points for the mark G, 45 for VG.

You can write your answers directly below the questions on the question paper. The available space gives an indication of how long answers are expected: they should be short. You are free to use separate sheets of paper for drafting your answers and copy the final answers to the question paper. If you, for any reason, want to submit your answers on separate papers, please indicate this clearly on the question paper, and mark each separate sheet with your exam code and the numbers of the questions answered.

For your help, we provide two tables: UD labels and GF constructs.

Examiner: Aarne Ranta

Email: aarne.ranta@cse.gu.se

Phone: 031-772 10 82 (also connects to mobile phone)

relation	explanation	dependent-head	example
acl	clausal modifier of noun	S-N	<i>the <u>moon</u> as we see it</i>
acl:relcl	relative clause modifier	S-N	<i>the <u>moon</u> that we see</i>
advcl	adverbial clause modifier	S-C	<i>I <u>leave</u> if she goes</i>
advmod	adverbial modifier	ADV-C	<i>he <u>sleeps</u> now</i>
amod	adjectival modifier	ADJ-N	<i>black <u>cat</u></i>
appos	appositional modifier	N-N	<i><u>Macron</u>, the president</i>
aux	auxiliary	AUX-S	<i>does he <u>sing</u></i>
case	case marking	ADP-N	<i>on the <u>moon</u></i>
cc	coordinating conjunction	CCONJ-C	<i>and <u>dogs</u></i>
ccomp	clausal complement	S-C	<i>I <u>know</u> that he runs</i>
compound	compound	N-N	<i>data <u>science</u></i>
conj	conjunct	C-C	<i><u>cats</u> and dogs</i>
cop	copula	AUX-S	<i>he is <u>old</u></i>
csubj	clausal subject	S-S	<i>that is moves is <u>clear</u></i>
dep	unspecified dependency	C-C	(if nothing else works)
det	determiner	DET-N	<i>the <u>cat</u></i>
expl	expletive	PRON-S	<i>there <u>is</u> hope</i>
fixed	fixed multiword expression	ADP-C	<i><u>because</u> of</i>
flat	flat multiword expression	PROPN-PROPN	<i><u>Adam</u> Smith</i>
iobj	indirect object	N-VERB	<i>she gave us a hint</i>
mark	marker	PART/SCONJ-S	<i>to <u>go</u></i>
nmod	nominal modifier	NOUN-NOUN	<i><u>man</u> on the moon</i>
nmod:poss	possessive modifier	N-NOUN	<i>my <u>cat</u></i>
nsubj	nominal subject	N-S	<i>John <u>walks</u></i>
nsubj:pass	nominal subject of passive	N-VERB	<i>John was <u>seen</u></i>
nummod	numeric modifier	NUM-N	<i>five <u>cats</u></i>
obj	object	N-VERB	<i>she <u>sees</u> us</i>
obl	oblique nominal	N-S	<i>she <u>comes</u> with us</i>
parataxis	parataxis	VERB-VERB	<i>I <u>said</u>: come here</i>
punct	punctuation	PUNCT-S	<i>I <u>see</u> !</i>
root	root	S-	<i>John **walks</i>
xcomp	open clausal complement	S-S	<i>I <u>want</u> to go</i>

Syntactic relations used in UD standard 2, together with their typical uses. In the examples, the dependent is **boldfaced** and its head underlined. S means a sentence-like, N a noun-like, and C any kind of phrase.

Construct	Notation	Example
Abstract syntax module	<code>abstract</code>	<code>abstract Lang =...</code>
Concrete syntax module	<code>concrete</code>	<code>concrete LangEng of Lang Lang =...</code>
Resource module	<code>resource</code>	<code>resource ResEng Lang =...</code>
Module extension	<code>**</code>	<code>abstract Lang = Noun,Verb **...</code>
Module opening	<code>open</code>	<code>resource ResEng = open Prelude in...</code>
Abstract syntax category	<code>cat</code>	<code>cat NP</code>
Abstract syntax function	<code>fun</code>	<code>fun Pred : NP -> VP -> S</code>
Linearization type	<code>lincat</code>	<code>lincat N = {s : Number => Str}</code>
Linearization rule	<code>lin</code>	<code>lin Pred np vp = np.s!Nom ++ vp!np.a</code>
Parameter type	<code>param</code>	<code>param Case = Nom Acc</code>
Auxiliary operation	<code>oper</code>	<code>oper addS : Str -> Str = \x -> x + "s"</code>
String concatenation	<code>++</code>	<code>"loves" ++ "Mary"</code>
Token concatenation	<code>+</code>	<code>"Maria" + "m" ↓ "Mariam"</code>
Function type	<code>-></code>	<code>NP -> VP -> S</code>
Function application	<code>f a b</code>	<code>Pred np vp</code>
Function abstraction	<code>\ -></code>	<code>\x,y -> x + y</code>
Table type	<code>=></code>	<code>Case => Str</code>
Table	<code>table</code>	<code>table {Nom =>"she" ; Acc =>"her"}</code>
Selection from table	<code>!</code>	<code>she_NP.s ! Acc ↓ "her"</code>
Table with one branch	<code>\\ =></code>	<code>\\p,q => np ! q ! p</code>
Record type	<code>{...:...}</code>	<code>{s : Str ; g : Gender}</code>
Record	<code>{...=...}</code>	<code>{s = "doctor" ; g = Fem}</code>
Projection from record	<code>.</code>	<code>{s = "doctor"}.s ↓ "doctor"</code>
Record update	<code>**</code>	<code>doctor_N ** {g = Masc}</code>
Case expression	<code>case</code>	<code>case np.a of {Ag n _ => cn.s ! n}</code>
Tuple type	<code>*</code>	<code>Number * Case</code>
Tuple	<code><,></code>	<code><Sg,Dat></code>
Comment	<code>--</code>	<code>-- comment till the end of line</code>
Comment	<code>{- -}</code>	<code>{- comment of any length -}</code>

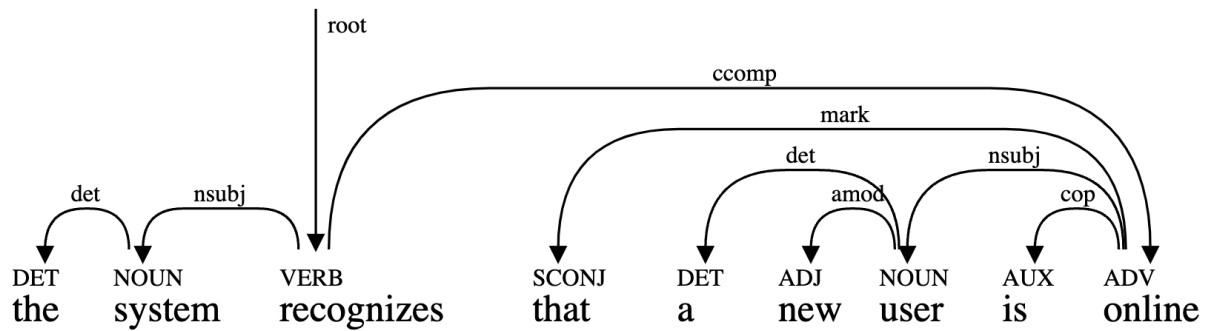
This is a reading guide for GF notation. The first five rows are about modules, the next six list the different kinds of rules. The rest are expressions for types and objects. The notation $e \Downarrow v$ means that expression e is computed to value v .

Question 1 (12p)

Consider the following English sentence:

the system recognizes that a new user is online

Draw a graphical UD tree:



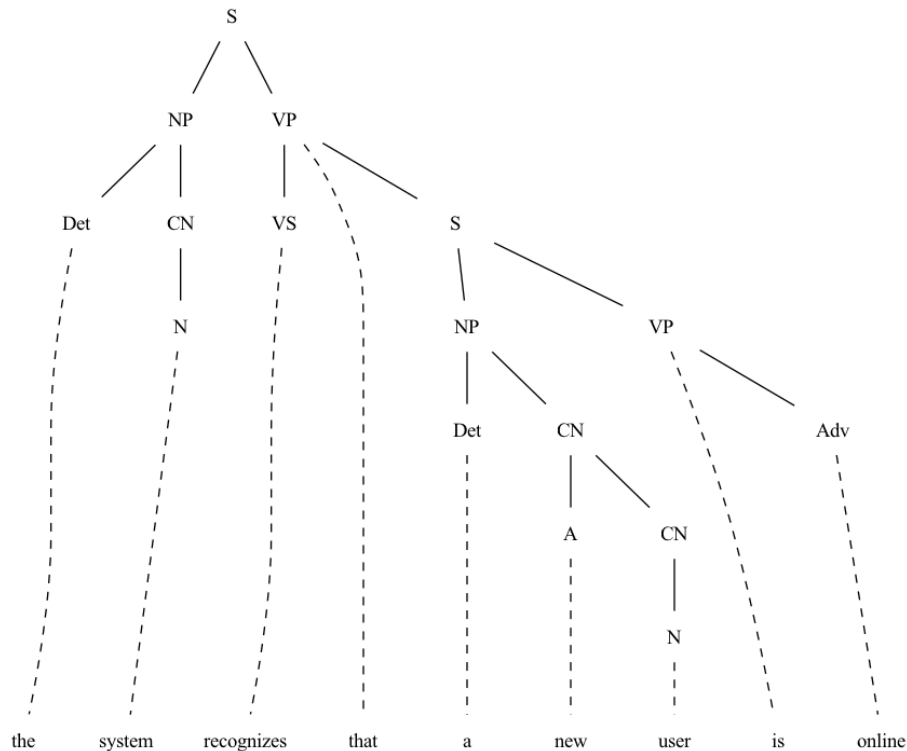
Write the CoNLL-U listing here, showing just the following six fields:

	ID	FORM	LEMMA	UPOS	HEAD	DEPREL
for instance	4	us	we	PRON	3	obj

1	the	the	DET	2	det	
2	system	system	NOUN	3	nsubj	
3	recognizes	recognize	VERB	0	root	
4	that	that	SCONJ	9	mark	
5	a	a	DET	7	det	
6	new	new	ADJ	7	amod	
7	user	user	NOUN	9	nsubj	
8	is	be	AUX	9	cop	
9	online	online	ADV	3	ccomp	

Question 2 (12p)

Draw a phrase structure tree for the same English sentence as in Question 1. Make sure that each combination of a head with its dependents is a phrase (i.e. a subtree) in this tree. Whenever possible, use conventional names of phrase categories - at least, S, NP, VP, AP, Adv.



Write a set of phrase structure rules (a.k.a. BNF rules or context-free rules), in format like

VP ::= "is" AP

that correspond to the tree that you drew.

```

S    ::= NP VP
NP   ::= Det CN
CN   ::= N
VP   ::= VS "that" S
CN   ::= A CN
VP   ::= "is" Adv
Det  ::= "the"
Det  ::= "a"
N    ::= "system"
N    ::= "user"
A    ::= "new"
VS   ::= "recognizes"
Adv  ::= "online"
  
```

Question 3 (12 p)

The following context-free grammar contains rules for drawing figures:

```

Command ::= "draw" Object
Object  ::= "a" Size Color Shape
Size    ::= "big" | "small"
Color   ::= "blue" | "green"
Shape   ::= "circle" | "square"

```

(The vertical bar is used to indicate alternative right-hand sides: each of the last three rules is hence a shorthand for two rules.) Write the GF abstract syntax and concrete syntax of a grammar that recognizes the same language as this grammar. Use the same category (cat) names, but invent the function (fun) names yourself.

```

abstract Draw = {

cat
  Command ; Object ; Size ; Color ; Shape ;
fun
  MkCommand : Object -> Command ;
  AObject : Size -> Color -> Shape -> Object ;
  big_Size : Size ;
  small_size : Size ;
  blue_Color : Color ;
  green_Color : Color ;
  circle_Shape : Shape ;
  square_Shape : Shape ;
}

concrete DrawEng of Draw = {

lincat
  Command, Object, Size, Color, Shape = Str ;
lin
  MkCommand object = "draw" ++ object ;
  AObject size color shape = "a" ++ size ++ color ++ shape ;
  big_Size = "big" ;
  small_size = "small" ;
  blue_Color = "blue" ;
  green_Color = "green" ;
  circle_Shape = "circle" ;
  square_Shape = "square" ;
}

```

Question 4 (12p)

Consider the same context-free grammar as in Question 3, with the following rules added:

```
Object ::= "two" Size Color Shapes
Shapes ::= "circles" | "squares"
```

In GF, you don't need to have a separate category Shapes: it is enough to have just one category Shape, with a linearization type that has a variable feature that takes care of "circle" vs. "circles" and "square" vs. "squares". Thus only one new function is needed in the abstract syntax, say

```
fun TwoObjects : Size -> Color -> Shape -> Object
```

To adapt your GF grammar, you need to change a couple of things and add some new things. Show how to do this:

4.1. Add a suitable parameter type to capture the variation in Shape.

```
param Number = Sg | Pl ;
```

4.2. Define a new linearization type (lincat) for Shape so that it takes care of this variation.

```
lincat Shape = Number => Str ;
```

4.3. Define the linearization rule of TwoObjects.

```
lin TwoObject size color shape = "two" ++ size ++ color ++ shape ! Pl ;
```

4.4. Change the linearization rule of the previously given function forming objects with the indefinite article "a", so that it is correct with respect to the new linearization type of Shape.

```
lin AObject size color shape = "a" ++ size ++ color ++ shape ! Sg ;
```

4.5. Change the linearization rules for "circle" and "square" so that they match the new linearization type of Shape.

```
lin
circle_Shape = table {Sg => "circle" ; Pl => "circles"} ;
square_Shape = table {Sg => "square" ; Pl => "squares"} ;
```

Question 5 (12p)

Extracting the FEATS field of a Czech UD treebank for NOUN yields the following 52 combinations of features (sorted alphabetically, shown in two columns):

Animacy=Anim Case=Acc Gender=Masc Number=Plur	Case=Acc Gender=Fem Number=Plur
Animacy=Anim Case=Acc Gender=Masc Number=Sing	Case=Acc Gender=Fem Number=Sing
Animacy=Anim Case=Dat Gender=Masc Number=Plur	Case=Acc Gender=Neut Number=Plur
Animacy=Anim Case=Dat Gender=Masc Number=Sing	Case=Acc Gender=Neut Number=Sing
Animacy=Anim Case=Gen Gender=Masc Number=Plur	Case=Dat Gender=Fem Number=Plur
Animacy=Anim Case=Gen Gender=Masc Number=Sing	Case=Dat Gender=Fem Number=Sing
Animacy=Anim Case=Ins Gender=Masc Number=Plur	Case=Dat Gender=Neut Number=Plur
Animacy=Anim Case=Ins Gender=Masc Number=Sing	Case=Dat Gender=Neut Number=Sing
Animacy=Anim Case=Loc Gender=Masc Number=Plur	Case=Gen Gender=Fem Number=Plur
Animacy=Anim Case=Loc Gender=Masc Number=Sing	Case=Gen Gender=Fem Number=Sing
Animacy=Anim Case=Nom Gender=Masc Number=Plur	Case=Gen Gender=Neut Number=Plur
Animacy=Anim Case=Nom Gender=Masc Number=Sing	Case=Gen Gender=Neut Number=Sing
Animacy=Anim Case=Voc Gender=Masc Number=Sing	Case=Ins Gender=Fem Number=Plur
Animacy=Inan Case=Acc Gender=Masc Number=Plur	Case=Ins Gender=Fem Number=Sing
Animacy=Inan Case=Acc Gender=Masc Number=Sing	Case=Ins Gender=Neut Number=Plur
Animacy=Inan Case=Dat Gender=Masc Number=Plur	Case=Ins Gender=Neut Number=Sing
Animacy=Inan Case=Dat Gender=Masc Number=Sing	Case=Loc Gender=Fem Number=Plur
Animacy=Inan Case=Gen Gender=Masc Number=Plur	Case=Loc Gender=Fem Number=Sing
Animacy=Inan Case=Gen Gender=Masc Number=Sing	Case=Loc Gender=Neut Number=Plur
Animacy=Inan Case=Ins Gender=Masc Number=Plur	Case=Loc Gender=Neut Number=Sing
Animacy=Inan Case=Ins Gender=Masc Number=Sing	Case=Nom Gender=Fem Number=Plur
Animacy=Inan Case=Loc Gender=Masc Number=Plur	Case=Nom Gender=Fem Number=Sing
Animacy=Inan Case=Loc Gender=Masc Number=Sing	Case=Nom Gender=Neut Number=Plur
Animacy=Inan Case=Nom Gender=Masc Number=Plur	Case=Nom Gender=Neut Number=Sing
Animacy=Inan Case=Nom Gender=Masc Number=Sing	Case=Voc Gender=Fem Number=Plur
Animacy=Inan Case=Voc Gender=Masc Number=Sing	Case=Voc Gender=Fem Number=Sing

A quick inspection shows that there are four different types of features, each with different possible values - for instance, two Number values and seven Case values.

5.1. The first task is straightforward: define each feature type as a GF param type, where each feature value is a constructor.

```
param
  Animacy = Anim | Inanim ;
  Case = Acc | Dat | Gen | Ins | Loc | Nom | Voc ;
  Gender = Masc | Fem | Neut ;
  Number = Sing | Plur ;
```

(continues on next page)

5.2. The second task needs more reflection. First, notice that Animacy combines only with one of the genders, Masc. Now, redefine the Gender type so that the Masc constructor takes an Animacy argument.

```
param Gender = Masc Animacy | Fem | Neut ;
```

5.3. Define the linearization type of nouns so that Gender is an inherent feature, whereas Case and Number are variable features.

```
lincat N = {s : Number => Case => Str ; g : Gender} ;
```

5.4. What are the combinations that your param type definitions imply, but which are not found in the listing? The reason is that these combinations are not found in Czech.

Case Voc (vocative) in Masc Plur and in Neut

5.5. Define the param types in such a way that they no longer imply the existence of those combinations (but continue to imply all the existing ones).

I realized, too late, that this subquestion was not relevant for nouns, but only for adjectives. This is because gender is not a variable feature for nouns but only for adjectives. Therefore, the missing combinations are not a problem for noun inflection tables. Therefore, this subquestion is ignored in grading: you can get full points for question 5 without answering 5.1.

But here is a possible parameter type for adjectives. The idea is to remove Voc from cases, and add separate constructions for those gender combinations where vocative is meaningful.

```
param
  Case = Acc | Dat | Gen | Ins | Loc | Nom ;
  AForm =
    ACase Gender Number Case
    | AVocMascSg Anim
    | AVocFem Number
```