Computer Language Processing

Exercise Sheet 07

November 10, 2022

Welcome to the seventh exercise session of CS320 !
This week, the goal is to get prepared for the midterm.

Exercise Lexical analysis

To practice lexical analysis, you can do the following exercises from past exams :

Exam 2018 : Exercise 1

Exam 2019 : Exercise 1

Exercise Parsing LL(1)

To practice LL(1) parsing, you can do the following exercises from past exams :

Exam 2009 : Exercise 2

Exam 2014 : Problem 3

Exam 2018 : Exercise 2

Exam 2019 : Exercise 2

Exercise CYK parsing and Chomsky's normal form

To practice CYK parsing and Chomsky normal form, you can do the following exercises from past exams :

Exam 2013: Ex3

Exam 2014: Ex4

Exam 2015: Ex3

(2015 Ex 3) Consider the following grammar that accepts JSON (JavaScript Object Notation) strings:

json ::= object

object ::= { pairs } | { }

pairs ::= pairs STRING : value | ε

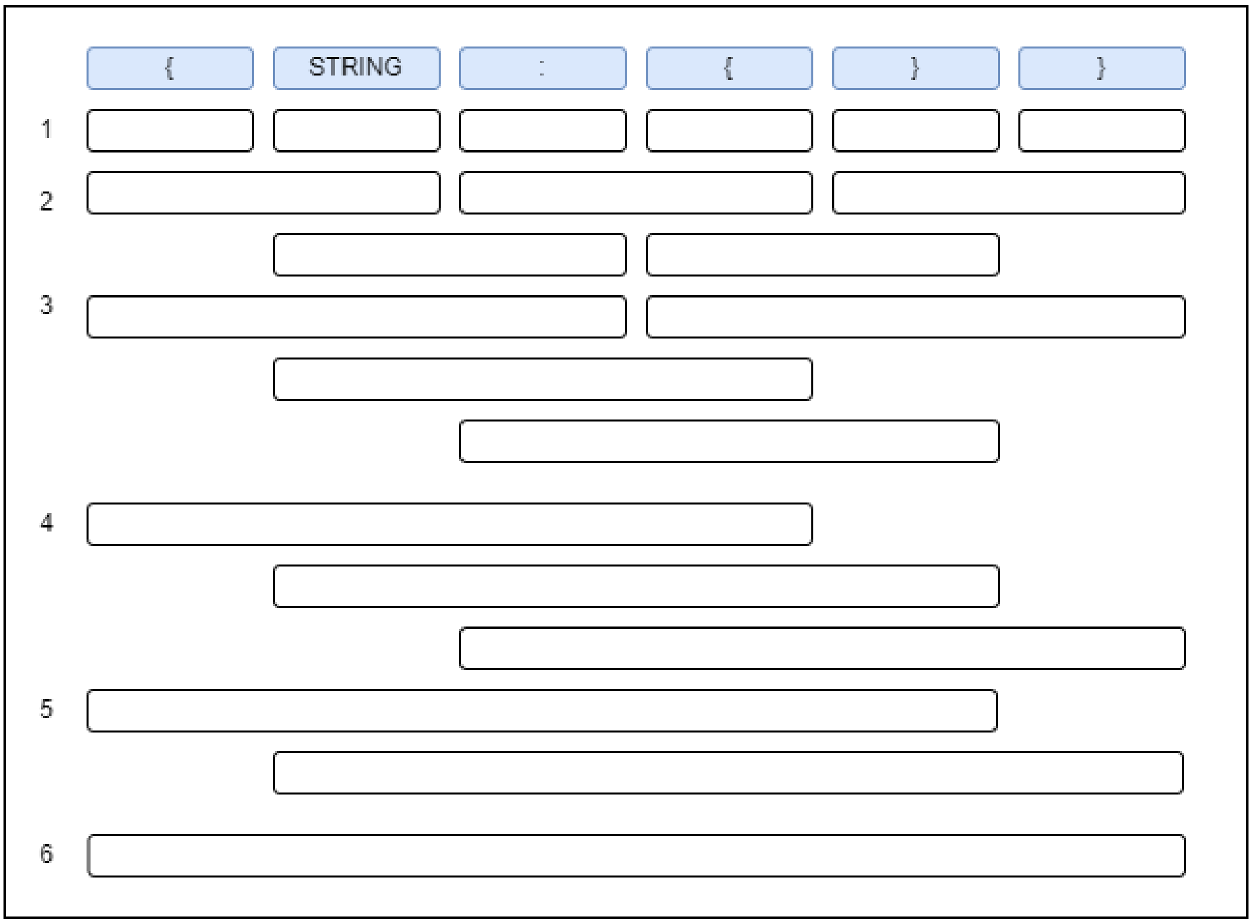
value ::= STRING | NUMBER | object

a) Convert the grammar to Chomsky Normal Form. Recall that a grammar in CNF should satisfy the following properties:

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

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

b) 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.



Exercise Types rules, preservation and inferences

(2013 Ex 1 - we removed subtyping as it was not presented this year)

Consider the following typed language on immutable identifiers:

$$expr ::= \textbf{val } ident = expr; expr \qquad \text{Variable binding} \tag{4.1}$$
$$expr ::= i \qquad \text{where i is an integer constant} \tag{4.2}$$
$$expr ::= (ident : T) \Rightarrow expr \qquad \text{Creates an anonymous function} \tag{4.3}$$
$$expr ::= expr(expr) \qquad \text{Applies a function} \tag{4.4}$$
$$T ::= Int \qquad \text{Int type} \tag{4.5}$$
$$T ::= T \Rightarrow T \qquad \text{Function type} \tag{4.6}$$
$$T ::= Collection[T] \qquad \text{Collection type} \tag{4.7}$$

For the purpose of this exercise, we add a polymorphic type $Collection[T]$ to the language, where T can be any existing type. We extend the expression syntax with existing customized function symbols:

$$expr ::= \text{EmptyCol}[\alpha] \qquad \text{Creates an empty collection of elements of type } \alpha \tag{4.8}$$
$$expr ::= \text{add}(expr, expr) \qquad \text{Adds an element to a collection} \tag{4.9}$$
$$expr ::= \text{permute}(expr) \qquad \text{Returns a function from a collection} \tag{4.10}$$
$$expr ::= \text{map}(expr) \qquad \text{Returns a function from a collection} \tag{4.11}$$
$$expr ::= \text{flatMap}(expr) \qquad \text{Returns a function from a collection} \tag{4.12}$$

Informally, the semantics of the extensions is the following:

- EmptyCol$[\alpha]$ (line 4.8) takes a type parameter and returns an empty collection of this type.
- add(e_1, e_2) (line 4.9) takes a collection e_1 and an element e_2 and returns the collection e_1 to which the element has been added.
- permute$(expr)$ (line 4.10) takes a collection, and returns a function which is a permutation mapping of the elements of the collection. See example below.
- map$(expr)$ (line 4.11) takes a collection and returns a function. This function accepts a mapping from an element to another element and returns the mapped original collection.
- flatMap$(expr)$ (line 4.12) takes a collection and returns a function. This function accepts a mapping from an element to a collection of elements and returns the union of all images of elements of the original collection.

For example, if $x = \text{add}(\text{add}(\text{add}(\text{EmptyCol}[Int], 1), 2), 3)$, it could be that:

$$\text{permute}(x)(1) == 2$$

$$\text{permute}(x)(2) == 3$$

$$\text{permute}(x)(3) == 1$$

For any other integer, i, permute$(x)(i)$ could be either 1, 2 or 3.

If collections were sets, we would also have the following (to illustrate):

$$\text{map}(x)(a \Rightarrow 1) = \{1\}$$

$$\text{flatMap}(x)(a \Rightarrow \text{add}(x, a + 3)) = \{1, 2, 3, 4, 5, 6\}$$

$$\text{map}(x)(\text{permute}(x)) = \text{map}(x)((i : Int) \Rightarrow \text{permute}(x)(i)) = x$$

The type rules for expressions are the following:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash f : T \Rightarrow U}{\Gamma \vdash f(e) : U} \qquad \frac{\Gamma \vdash e_1 : T \qquad \Gamma, (x, T) \vdash e_2 : U}{\Gamma \vdash (\text{val } x = e_1; e_2) : U}$$

a) Complete the following type rule templates (replace all ??? occurences) so that they are consistent with the described meaning of the extensions and sufficient to type check the extension if we exclude subtyping. You can abbreviate *Collection*[*T*] as *C*[*T*]. Here is a example rule for flatMap:

$$\frac{\Gamma \vdash e : Collection[T]}{\Gamma \vdash \text{flatMap}(e) : (T \Rightarrow Collection[T]) \Rightarrow Collection[T]}$$

$$\frac{???}{\Gamma \vdash \text{EmptyCol}[\alpha] : ???} \qquad \frac{???}{\Gamma \vdash \text{add}(e_1, e_2) : ???} \qquad \frac{???}{\Gamma \vdash \text{permute}(e) : ???} \qquad \frac{???}{\Gamma \vdash \text{map}(e) : ???}$$

b) Prove that the following program type checks by writing the type derivation tree. You can write a separate tree for each right-hand side of variable definition.

> **val** x = add(add(EmptyCol[Int], 1), 2)
> **val** z = add(add(x, 3), 4)
> **val** y = add(EmptyCol{Int ⇒ Int}, permute(z))
> flatMap(y)(map(x))

(2018 Ex 4.2)
For which of the following expressions does type inference using unification succeed? Circle the correct answers.

1. y ⇒ (x ⇒ (x, y))
2. x ⇒ (y ⇒ (x(y) + y(x))
3. f ⇒ (x ⇒ f (f (x)))
4. f ⇒ (f (x ⇒ 4) + f (5))

Unification algorithm

Apply the unification algorithm on the following function:

```
    def curry(f) = {
        x ⇒ y ⇒ f ((x,y))
    }
```

((x: τ) ⇒ ((y: τ_1) ⇒

((f: τ_2)(((x: τ), (y: τ_1)): τ_3)): τ_4

): τ_5

): τ_6

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

1. $\tau_2 = (\tau_3 \Rightarrow \tau_4)$
   $\tau_3 = (\tau, \tau_1)$
   $\tau_5 = (\tau_1 \Rightarrow \tau_4)$
   $\tau_6 = (\tau \Rightarrow \tau_5)$
2.
3.

Write down an expression for the type of the argument (f) and of the result of curry in terms of types τ, τ_1, τ_4:

(argument)	f :	
(result)	curry(f) :	