COMS 3003A

Assignment 2

May 15, 2025

This assignment contributes 10% to your final mark
Due Date: 27 May 2025

A template python file is provided as `student_template.py` that has an implementation of an Expression Tree (see section 2). The use of this file is not mandatory, but several helper functions (printing, access etc.) as well as an example of how the data structure can be used is provided.

(50 Points) Part A: Computational First Order Logic

In Python, build a framework to reason over First Order Logical formula's and models. All listed files are available in `Assignment2PartA.zip`.

- An expression φ will be encoded as the string $\langle\varphi\rangle$ according to the rules listed in section 1.
- Any output formulas ψ should also be encoded to $\langle\psi\rangle$ according to the rules in section 1.
- A model $\mathfrak{M}$ will be encoded as the string $\langle\mathfrak{M}\rangle$ as a JSON string, using the format described in `model_format.json`.

1. (**10 Points**) Write a Python script that takes a word $w \in \Sigma^*$ as input and decides if that word is a well formed string.
   - A well formed expression is a string in a First Order language that is in the form of a formula (Course Notes, Definition 5.3.11) or term (Course Notes, Definition 5.3.7).
   - Well formed expressions are therefore the interpretable strings of a first order language.
   - For this question, a well formed expression is therefore any string correctly formed by the rules described in Section 1 and the definitions of formulas and terms.

Input	Output
$@x(P(x))$	Formula
$f(x)$	Term
$af(([])$	None
$\&\#P()$	None

2. (**10 Points**) Write a Python script that takes a formula φ as input, and then outputs the signature Ω and list of variables Var of that formula.

   (a) The output must be sorted using Pythons `sorted` method

Input	Output

$@x(P(x))$	`predicates: [P]`
	`functions: []`
	`constants: []`
	`variables: [x]`

$\&(@x(P(x, s(y))), Q(0))$	`predicates: [P,Q]`
	`functions: [s]`
	`constants: [0]`
	`variables: [x,y]`

3. (**15 Points**) Write a Python script that takes a formula φ as input and then converts it into Negation-Normal-Form (NNF).

   - NNF Wikipedia page (https://en.wikipedia.org/wiki/Negation_normal_form)
   - NNF Transformation Rules:
     (a) $\neg(\neg(A)) \implies A$
     (b) $\neg(A \vee B) \implies (\neg A) \wedge (\neg B)$
     (c) $\neg(A \wedge B) \implies (\neg A) \vee (\neg B)$
     (d) $A \to B \implies (\neg A) \vee B$
     (e) $\neg(\forall x(A)) \implies \exists x(\neg A)$
     (f) $\neg(\exists x(A)) \implies \forall x(\neg A)$
   - Each rule should be evaluated from left to right as encountered in the formula.

Input	Output
$!(!(A))$	A
$!(@x(A))$	$\#x(!(A))$
$!(!(@x(A)))$	$@x(A)$

4. (**15 Points**) Write a Python script that takes as input a model $\mathfrak{M}$, and a list of formula $[\varphi]$, and decides if each formula φ_i is satisfied by $\mathfrak{M}$ ($\mathfrak{M} \vDash \varphi_i$ for all $\varphi_i \in [\varphi]$).

   - The first value of input is $\langle \mathfrak{M} \rangle$
   - Each subsequent input value is a formula φ_i, and you must then output `T` if $\mathfrak{M} \vDash \varphi_i$ or `F` if $\mathfrak{M} \nvDash \varphi_i$.
   - Repeat the previous step until the input is the string `done`, where you must terminate
   - The format of a model is provided in `Q4/model_format.json`
   - Two sample models, "Finite-Naturals" and "JumpFish" are in `Q4/sample1_model.json` and `Q4/sample2_model.json` respectively
   - Example input and output is provided in `Q4/sample(i)_in.txt` and `Q4/sample(i)_out.txt`

Input	Output
$\langle \mathfrak{M} \rangle$	
$\langle \varphi_1 \rangle$	`T`
$\langle \varphi_2 \rangle$	`F`
...	...
$\langle \varphi_N \rangle$	`T`
`done`	

(50 Points) Part B: Turing Machines and First Order Logic

In Python, construct the formula φ_M for a Turing Machine M. All listed files are available in `Assignment2PartB.zip`.

- A Turing Machine M will be encoded as a JSON string $\langle M \rangle$ using the format in the `machine_format.json` file.
- An expression φ is encoded into string $\langle \varphi \rangle$ according to the rules in Section 1.
- The input and output for each question is in the following format:

Input	Output
$\langle M \rangle$	$\langle \varphi \rangle$

  The JSON TM must be read from stdin, and the string formula must be output to stdout.

- A sample Turing Machine is provided `sample1_input.txt` (`sample1_machine.json`) with output examples for Question 2 and Question 7.

1. (**4 Points**) Construct φ_ε from Turing Machine M

$$\varphi_\varepsilon := \big(Q_0(0) \wedge C(0,1)\big) \wedge \big(S_1(0,0) \wedge (\forall x((\neg(x=0)) \rightarrow S_0(0,x)))\big)$$

2. (**10 Points**) Construct φ_Q from Turing Machine M

$$\varphi_Q := \forall x\big(\bigvee_{k=0}^{|Q|-1} (Q_k(x) \wedge \bigwedge_{j \neq k} (\neg(Q_j(x))))\big)$$

3. (**10 Points**) Construct φ_Δ from Turing Machine M

$$\varphi_\Delta := \forall x(\forall y\big(\bigvee_{k=0}^{|\Delta|-1} (S_k(x,y) \wedge \bigwedge_{j \neq k} (\neg(S_j(x,y))))\big))$$

4. (**3 Points**) Construct φ_C from Turing Machine M

$$\varphi_C := \forall x(\exists y\big(C(x,y) \wedge (\forall z(C(x,z) \rightarrow (z=y)))\big))$$

5. (**10 Points**) Construct φ_δ from Turing Machine M

$$\varphi_\delta := \bigwedge_{I \in \delta} \varphi_I$$

$$\begin{aligned}
&\text{LEFT } \varphi_I := \forall x(\forall y\big(\\
&\qquad\qquad ((Q_i(x) \wedge C(x,y)) \wedge S_m(x,y)) \rightarrow (\\
&\qquad\qquad\quad \underline{(\underline{(Q_j(s(x)) \wedge \underline{\exists v((s(v)=y) \wedge (C(s(x),v))}} \wedge S_l(s(x),y))\wedge} \\
&\qquad\qquad\qquad \underline{\forall z((\neg(z=y)) \rightarrow (\bigwedge_{k=0}^{|\Delta|-1} (S_k(x,z) \rightarrow S_k(s(x),z))))} \\
&\qquad\qquad) \\
&\qquad \big))
\end{aligned}$$

$$
\begin{aligned}
\text{RIGHT } \varphi_I :=\ & \forall x(\forall y\big(\\
& \quad (\underline{(Q_i(x) \wedge C(x,y)}) \wedge S_m(x,y)) \rightarrow (\\
& \qquad \underline{(\underline{(Q_j(s(x)) \wedge C(s(x),s(y))} \wedge S_l(s(x),y))\wedge} \\
& \qquad\quad \underline{\forall z((\neg(z=y)) \rightarrow (\bigwedge_{k=0}^{|\Delta|-1} (S_k(x,z) \rightarrow S_k(s(x),z))))} \\
& \quad) \\
& \big))
\end{aligned}
$$

6. (**3 Points**) Construct φ_{HALT} from Turing Machine M

$$\varphi_{\text{HALT}} := \exists x(Q_1(x) \vee Q_2(x))$$

7. (**10 Points**) Construct φ_M from Turing Machine M

$$\varphi_M := ((((\varphi_\varepsilon \wedge \varphi_Q) \wedge \varphi_\Delta) \wedge \varphi_C) \wedge \varphi_\delta) \rightarrow \varphi_{\text{HALT}}$$

1 First Order Logic as Strings

This section covers how first order words must be encoded for this assignment. A given first order word φ is encoded by:

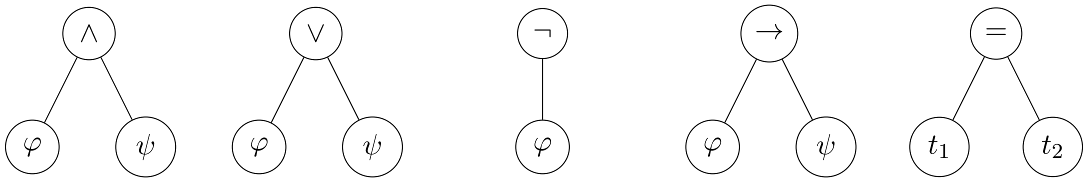
1. The standard First Order alphabet of $\{=, \wedge, \vee, \neg, \rightarrow, \forall, \exists\}$ will be replaced with $\{=, \&, |, !, >, @, \#\}$.
   - Note that $a > b$ does not mean that a is greater then b, but rather a implies b.
2. $\langle\varphi\rangle$ is always in prefix notation
3. $\langle\varphi\rangle$ may have a space in as an input string, **but NO SPACES must be in the OUTPUT**.
4. $\langle\varphi\rangle$ will always have the evaluation order specified by brackets i.e. every bracket will always be specified.
5. Every subscript in φ will be encoded as the object's symbol followed by the subscript in square brackets. Empty square brackets are not a valid encoding of an object with no subscript.
   - $\langle P_2\rangle = P[2]$
   - $\langle f_1\rangle = f[1]$
   - $\langle a_{12}\rangle = a[12]$
   - $\langle P\rangle = P \neq P[]$
6. $\langle\varphi\rangle$ will have its predicates, functions, constants and variables identifiable by
   - Predicates are represented as one or more uppercase Roman letters with any arity ($P \in Pred_\Omega \iff \langle P\rangle = P = P(...)$)
   - Functions are represented as one or more lowercase Roman letters with an arity of at least 1 ($f \in Fun_\Omega \iff \langle f\rangle = f(...)$)
   - Constants are represented as one or more arabic numerals with an arity of zero, that are not in square brackets ($0 \in Con_\Omega \iff \langle 0\rangle = 0$)
   - Variables are represented as one or more lowercase Roman letters with an arity of zero ($x \in Var \iff \langle x\rangle = x$)
   - Regular Expression rules:
     - $P \in Pred_\Omega \iff \langle P\rangle \in$ `^[A-Z]+(\[\d+\])?$`
     - $f \in Fun_\Omega \iff \langle f\rangle \in$ `^[a-z]+(\[\d+\])?$`
     - $0 \in Con_\Omega \iff \langle 0\rangle \in$ `^(?<!\[)[0-9]+(?!\[)$`
     - $x \in Var \iff \langle x\rangle \in$ `^[a-z]+(\[\d+\])?$`
     - You can learn more about regex at https://regex101.com/ and https://en.wikipedia.org/wiki/Regular_expression.
7. Enumerated connectives ($\bigvee$ or $\bigwedge$) are grouped from left to right, and evaluated as one group
   - $\bigvee_{i=1}^{4} \varphi_i = ((\varphi_1 \vee \varphi_2) \vee \varphi_3) \vee \varphi_4$
   - $\psi \wedge \bigwedge_{3}^{i=1} \varphi_i = \psi \wedge ((\varphi_1 \wedge \varphi_2) \wedge \varphi_3)$

2 First Order Logic as a Data Structure

The supplied template file `student_template.py` uses a data structure known as an Expression Tree. This section details how this data structure works, and how a First Order word is expressed in the tree.

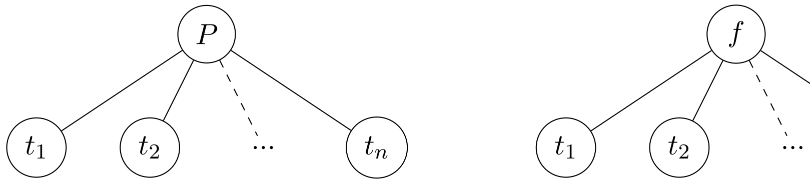
Definition 2.1 *A first order word can be expressed as a recursive tree using the following rules:*

- *The logical connectives and equality letter ($\wedge, \vee, \neg, \rightarrow, =$) can be expressed as the trees*



  where φ, ψ are formulas, and t_1, t_2 are terms.

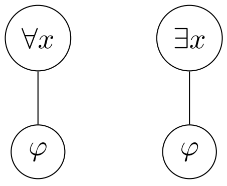
- *An n-ary predicate P, and function f can be expressed as the trees*



P t_1 t_2 $\cdots$ t_n
f t_1 t_2 $\cdots$ t_n

  where $t_1, t_2, ..., t_n$ are terms.

- *The quantifiers $\forall$ and $\exists$ can be expressed as the trees*



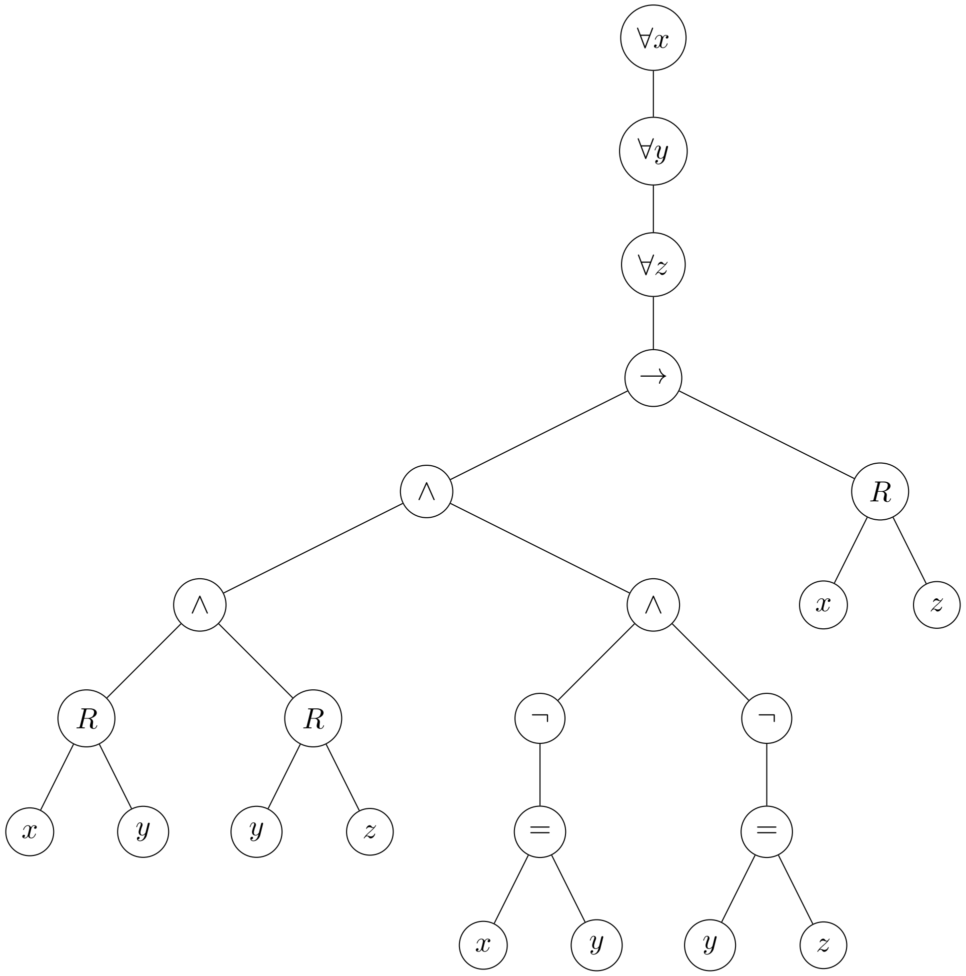
  where φ is a formula, and x is a variable.

Recursive Tree's can be useful to visualize the order of evaluation of a formula, as well as provide an easier to read notation for very large and complex formulas. In example 1, the complex formula for transitivity is difficult to read with the full set of brackets indicating evaluation order. When represented as a tree, it is very easy to identify each component of the formula, and what part of the formula it should be evaluated under.

Example 1 *Transitivity for a property R over 3 distinct objects has the formula*

$$\forall x \forall y \forall z(((R(x,y) \wedge R(y,z)) \wedge (\neg(x=y) \wedge \neg(y=z))) \rightarrow R(x,z))$$

and tree



Instead of using a string based data structure for representing a formula, the equivalent recursive trees offer a way to store the same formula as a Tree data structure called an Expression Tree.

This data structure has the psuedocode

```
ExpressionTree {
    value: string,
    children: List[ExpressionTree],
    parent: ExpressionTree
}
```

The methods `add_child`, `remove_child` and, `replace` help maintain the ExpressionTree by keeping the parents and children correctly synced.

```
add_child(A: ExpressionTree, B: ExpressionTree) {
    A.children.add B
    B.parent = A
}
remove_child(A: ExpressionTree, child_index: integer) {
    A.children[child_index].parent = null
    A.children.remove child_index
}
replace(A: ExpressionTree, B: ExpressionTree) {
    A.parent = B.parent
    A.parent.children[index of B in A.parent.children] = A
    B.parent = null
}
```

An operation over an ExpressionTree is typically performed recursively. The operation is applied from leaf-to-root by calling the operation after the recursive call, and root-to-leaf by calling the operation before the recursive call.

```
    root_to_leaf_operation(node: ExpressionTree) {
        Operation(node)
        for each child in node.children {
            root_to_leaf_operation(child)
        }
    }
    leaf_to_root_operation(node: ExpressionTree) {
        for each child in node.children {
            root_to_leaf_operation(child)
        }
        Operation(node)
    }
```

As an example, recovering the prefix string of a formula can be done by printing each nodes value before printing it's children (root-to-leaf), and the postfix by printing after it's children (leaf-to-root).

```
    print_prefix(node: ExpressionTree) {
        print node.value
        print "("
        for each child in node.children {
            print_prefix(child)
            print ","
        }
        print ")"
    }
    print_postfix(node: ExpressionTree) {
        print "("
        for each child in node.children {
            print_postfix(child)
            print ","
        }
        print ")"
        print node.value
    }
```