CSCI3180 Assignment 2: Prolog

2024-2025, Term 2

Introduction

In this two-part assignment, you'll use Prolog (1) to implement an evaluator for expressions over inductively-defined natural numbers and (2) to implement a queue processing system. You'll complete this assignment by modifying the file `assignment_2.pl`. There are a total of **100 points** for this assignment.

Guidelines

You should follow the below guidelines during completion of the assignment:

1. Modify the file `assignment_2.pl` by replacing instances of `% your code here` with one or more lines of your own code.
2. Do not modify anything else in `assignment_2.pl` (e.g., do not reorder exercises, change other comments, etc.).
3. **With the exception of Exercise 1, your code should not use any Prolog arithmetic operators**. Violation of this guideline will result in a reduction of 20 points from the assignment.

Failure to adhere to the guidelines may result in a reduction of points from your assignment.

Submission Guidelines

The submission deadline for the assignment is **20 March 2025 at 11:59pm**. There will be a late submission penalty of **1 point per 5-minutes late**, where the amount of time late is rounded *up* to the nearest 5 minutes. You should complete the following by the deadline:

- Ensure that you have completed the declaration at the top of `assignment_2.pl`.
- On Blackboard, for Assignment 2, attach and submit *only* your completed version of `assignment_2.pl`. (Do not zip or otherwise compress the file.) It *must* be called `assignment_2.pl`.

You can submit as many times as you want, but **only the latest submission will be graded**. You are encouraged to submit early to prevent any issues!

Part 1: Natural Numbers (50 pts)

In the first part of the assignment, you'll be using Prolog to evaluate arithmetic expressions over natural numbers.

Below is an inductive definition of the natural numbers in Prolog:

```prolog
nat(z). % base case: zero
nat(s(N)) :- nat(N). % inductive case
```

Note that this corresponds to the following (hopefully familiar) grammar:

$$\langle nat \rangle ::= z | s(\langle nat \rangle)$$

This also corresponds to the following inference rules, where N is a metavariable:

$$(\textsc{Zero})\ \text{nat(z)}$$

$$(\textsc{Inductive})\ \frac{\text{nat}(N)}{\text{nat(s}(N))}$$

Note that we have exactly one unique `nat` term for each natural number (0, 1, 2, 3, . . .). In particular, each `nat` term represents the natural number that corresponds to the number of occurrences of `s` in the term:

`nat` term	element in $\mathbb{N}$
`z`	0
`s(z)`	1
`s(s(z))`	2
⋮	⋮

Exercise 1 (10 pts)

Define a predicate `to_nat/2` over a Prolog integer and an inductively-defined natural number. The predicate should be true if and only if the integer expresses the natural number and should be able to be used to convert back-and-forth between Prolog integers and inductively-defined natural numbers.
The only arithmetic-related predicates/functions you should use are `is` and `+`, and you should not use any negative integers.
You should have the following results for the queries below:

```
?- to_nat(3,s(s(s(z)))).
true
```

```
?- to_nat(0,s(z)).
false
```

```
?- to_nat(0,s(z)).
false
```

```
?- to_nat(X,s(s(s(z)))).
X = 3
```

```
?- to_nat(3,N).
N = s(s(s(z)))
```

Exercise 2 (5 pts)

Define a predicate `nat_plus/3` that is `true` if and only if the sum of the first and second arguments are equal to the third argument, where all arguments are inductively defined natural numbers (i.e., `nat` should hold for them).
For example, you should have the following results for the queries below:

```
?- nat_plus(s(z),s(s(z)),s(s(s(z)))).
true
```

```
?- nat_plus(z,s(z),z).
false
```

Exercise 3 (5 pts)

Write a query that uses `nat_plus` and `to_nat` to discover the difference `X` of 40 and 3. The result of the query should discover `X` such that `X` = 37.

```
% ?- your query here
```

Exercise 4 (10 pts)

Define a predicate `nat_mul/3` that is `true` if and only if the product of the first and second arguments are equal to the third argument, where all arguments are inductively defined natural numbers (i.e., `nat` should hold for them).
For example, you should have the following results for the queries below.

```
?- nat_mul(s(z),s(s(z)),s(s(z))).
true
```

```
?- nat_mul(s(z),s(z),z).
false
```

Exercise 5 (20 pts)

Consider the following grammar of arithmetic expressions involving our inductive definition of natural numbers, where *nat* ranges over all terms for which `nat(`*nat*`)` holds:

$$e \in S ::= nat \mid (\texttt{plus}, S, S) \mid (\texttt{minus}, S, S) \mid (\texttt{times}, S, S)$$

We will refer to the language generated by this grammar as NatArith.
We will use strings in NatArith to represent arithmetic expressions over natural numbers involving addition, subtraction, and multiplication. For example, the following NatArith string represents the expression $(2 \times 2 + 3) - 1$:

```
(minus, (plus, (times, s(s(z)), s(s(z))), s(s(s(z)))), s(z))
```

Define a predicate `eval/2` that takes an expression in NatArith and outputs the result of evaluating it so that Prolog can prove the goals for the following queries:

```
?- to_nat(2, Two), to_nat(3, Three), to_nat(1, One), eval((minus, (plus, (
    times, Two, Two), Three), One), Six), to_nat(6,Six).
Two = s(s(z)),
Three = s(s(s(z))),
One = s(z),
Six = s(s(s(s(s(s(z))))))
```

```
?- to_nat(5, Five), to_nat(3, Three), to_nat(4, Four), eval((times, Five,
    (minus, Four, Three)), Five).
Five = s(s(s(s(s(z))))),
Three = s(s(s(z))),
Four = s(s(s(s(z))))
```

Note that `eval/2` should implement a big-step operational semantics for the language NatArith, where $e \Downarrow nat$ if and only if `eval(`e, nat`)` for arbitrary NatArith strings e and inductively-defined natural numbers *nat*.

Part 2: Nested Queue Processing (50 pts)

In this part of the assignment, you will be implementing an evaluator for two different semantics of a variation of the Queue language (which you may have seen before here: `https://forms.gle/c9MqJfoe87MnAHTi6`).

Names

First, we introduce a predicate `names/1` to define a set of names *Names* = {`alice`, `bob`, `charlie`, `eve`}:

```
name(alice).
name(bob).
name(charlie).
name(eve).
```

Syntax

Exercise 6 (5 pts)

We consider a Prolog list to be a *nested queue* if and only if its elements are (1) either in the set of names or (2) also nested queues. A grammar for nested queues is shown below:

$$q \in \langle \textit{nested queue} \rangle ::= [\,] \mid [\textit{name}] \mid [\textit{name}|q] \mid [q_1|q_2]$$

Define a predicate `queue/1` that is `true` if and only if the predicate's argument is a nested queue. For example, we expect the following results for the following queries:

```
?- queue([alice, bob, charlie]).
true
```

```
?- queue([[[[]]]]).
true
```

```
?- queue([[[]] | [alice, bob]]).
true
```

```
?- queue([[[]] | [alice, 1, bob]]).
false
```

Semantics

Exercise 7 (10 pts)

In this exercise, we consider implementing a small-step operational semantics for our nested queue processing language. A configuration has the form q, p, where q ranges over a nested queues in Prolog (i.e., Prolog lists ℓ such that `queue`(ℓ) holds), and p ranges over a Prolog list of names representing the names processed so far. The set of values for our operational semantics is the singleton set of just the empty list [ ].
Our nested queue processing language has the below small-step operational semantics, where q ranges over nested queues, p ranges over a Prolog list of names, *name* ranges over the names in *Names*:

$$(\textsc{Empty-Head})\ ([[\,]|q], p) \to (q, p)$$

$$(\textsc{Name-Head})\ ([\textit{name}|q], p) \to (q, p') \quad \text{where } p' \text{ is the result of adding } \textit{name} \text{ to the end of } p$$

$$(\textsc{Queue-Head})\ \frac{(q_1, p) \to (q_1', p')}{([q_1|q_2], p) \to ([q_1'|q_2], p')}$$

Define a predicate `step/4` such that `step`(q, p, q', p') is `true` if and only if $(q, p) \to (q', p')$ by the above operational semantics rules.
For example, we expect the following results for the following queries:

```
?- step([[[]] | [alice,bob]], [charlie], Q, P).
P=[charlie],
Q=[[], alice, bob]
```

```
?- step([alice,bob], [charlie], Q, P).
P=[charlie, alice],
Q=[bob]
```

Exercise 8 (5 pts)

Add cuts to your definition of `step/4` so that there is exactly one result returned for every query `step`$(q, p,$ `Q`, `P`$)$.

Exercise 9 (5 pts)

Define a predicate `step_rtc/4` that defines the reflexive-transitive closure of $\to$. In particular, you should have that `step_rtc`(q, p, q', p') is `true` if and only if $(q, p) \to^* (q', p')$.
Proof rules that (when combined with the proof rules for the small-step operational semantics above) define the reflexive-transitive closure are below:

$$(\textsc{Reflexive})\ (q, p) \to^* (q, p)$$

$$(\textsc{Transitive})\ \frac{(q, p) \to (q', p') \qquad (q', p') \to^* (q'', p'')}{(q, p) \to^* (q'', p'')}$$

Exercise 10 (5 pts)

Define a predicate `big_step/4` that defines a big-step semantics for the query processing language that is consistent with the small-step semantics. That is, for any q, p and p', the query `?- step_rtc`$(q, p, [\,], p')$ should yield result `true` if and only if the query `?- big_step`$(q, p, [\,], p')$ also yields result `true`.

Exercise 11 (20 pts)

The small-step operational semantics defined in the previous exercises is *deterministic* and only allows for processing names in a left-to-right order in the nested queue.
In this exercise, we would like to consider a different *nondeterministic* semantics for the queue processing language that allows for processing the names in the queue in *any* order. Define a predicate `step_n/4` for this nondeterministic small-step semantics that allows for the processing of any name or removal of any empty inner queue in the next step.
We expect the following results for the following queries:

```
?- step_n([[[]] | [alice,bob]], [charlie], [[], alice, bob],[charlie]).
true
```

```
?- step_n([[[]] | [alice,bob]], [charlie], [[[]], bob],[charlie, alice]).
true
```

```
?- step_n([[[]] | [alice,bob]], [charlie], [[[]], alice],[charlie, bob]).
true
```