Functions implemented in simple.pl

Question 1.1: List Processing

Write a predicate `sumsq_even(Numbers, Sum)` that sums the squares of only the even numbers in a list of integers.

Example:

```
?- sumsq_even([1,3,5,2,-4,6,8,-7], Sum).
Sum = 120
```

Note that it is the element of the list, not its position, that should be tested for oddness. (The example computes 2*2 + (-4)*(-4) + 6*6 + 8*8). Think carefully about how the predicate should behave on the empty list — should it fail or is there a reasonable value that Sum can be bound to?

To decide whether a number is even or odd, you can use the built-in Prolog operator N mod M, which computes the remainder after dividing the whole number N by the whole number M. Thus a number N is even if the goal 0 is N mod 2 succeeds. Remember that arithmetic expressions like X + 1 and N mod M are only evaluated, in Prolog, if they appear after the is operator. So 0 is N mod 2 works, but N mod 2 is 0 doesn't work.

Question 1.2: List Processing

Write a predicate `log_table(NumberList,ResultList)` that binds `ResultList` to the list of pairs consisting of a number and its log, for each number in `NumberList`. For example:

```
?- log_table([1, 3.7, 5], Result).
Result = [[1, 0.0], [3.7, 1.308332819650179], [5,
1.6094379124341003]].
```

Note that the Prolog built-in function `log` computes the natural logarithm, and that it needs to be evaluated using `is` to actually compute the log:

```
?- X is log(3.7).
X = 1.308332819650179.
?- X = log(3.7).
X = log(3.7).
```

Question 1.3: List Processing

Any list of integers can (uniquely) be broken into "parity runs" where each run is a (maximal) sequence of consecutive even or odd numbers within the original list. For example, the list `List=[8,0,4,3,7,2,-1,9,9]` can be broken into `[8,0,4]`, `[3,7]`, `[2]` and `[-1,9,9]` Write a predicate `paruns(List,RunList)` that converts a list of numbers into the corresponding list of parity runs. For example:

```
?- paruns([8,0,4,3,7,2,-1,9,9], RunList).
RunList = [[8, 0, 4], [3, 7], [2], [-1, 9, 9]]
```

Note: you can find out how to test if a number is even or odd from the Prolog Dictionary

Question 1.4: Prolog Terms

Arithmetic expressions can be written in *prefix* format, e.g 1+2*3 can be written as `add(1, mul(2, 3))`. If the operators available are `add`, `sub`, `mul`, `div`, write a Prolog program, `eval(Expr, Val)`, that will evaluate an expression, e.g.

```
?- eval(add(1, mul(2, 3)), V).
V = 7
?- eval(div(add(1, mul(2, 3)), 2), V).
V = 3.5
```

pronounResolution.pl

This assignment requires extending a definite clause grammar to perform reference resolution. You can find the grammar for a subset of English here . Download it as **ass3.pl** . You are required to write a Prolog predicate **process(LogicalForm, Ref1, Ref2)** , which you will find at the bottom of the file.

DON'T PANIC!!! It's a complicated program but most of the work has already been done for you.

You run the program by calling the predicate **run (Sentence, [RefList])** :

```
run(S, Refs) :-
    sentence(X, S, []), !,
    writeln(X),
    process(X, [], Refs),
    listing(history/1).
```

This takes an input sentence, **S** , and outputs, **Refs** , which is a list of the objects that each pronoun refers to. For example:

```
?- run([john,lost,his,wallet], Refs).
event(lost,[actor(thing(john,[])),object(possessive(his,thing(wallet,[]))])
history(thing(john, [isa(person), gender(masculine), number(singular)])).
history(thing(wallet, [isa(physical_object), gender(neutral), number(singular)])).
history(event(lost, [actor(thing(john, [])), object(possessive(his, thing(wallet, [])))])).
Refs = [john]
```

- This first line of the output is the parse of the sentence.
- You don't have to do anything to get this. It's already written for you, but you have to wrote the **process** predicate.
- This takes the logical form of the parsed sentence (i.e. event(lost, ...)) and goes through the logical form, looking for pronouns and finding which previous phrases they refer to.
- Each time you encounter a noun phrase or a verb phrase, you should use **assert** to add a new entry to the **history** in Prolog's database.
- A history entry looks like **history(H)** , where **H** represents either an **event** for a verb phrase, or a **thing** for a noun phrase.
- The next three lines in the output are the history entries that the program should create as a result for processing the logical form in this example.
- The last line of the output is the list of resolved references. In this case, there's only one: **his** resolves to **John.**

Process takes three arguments. The first is the logical form, e.g.

```
?- process(event(lost,[actor(thing(john,[])),object(possessive(his,thing(wallet,[])))]), [], Refs).
Refs = [John]
```

Your job is to write the **process** program. You can write as many additional predicates to help as you want, and you can even change the grammar, it you really want to dive into the code.

- You will have to work your way, recursively through the logical form and when you see a **thing** term or an **event** tern, assert a new history entry.
- When you see either a **personal** pronoun, or a **possessive** pronoun, you should use the history to find a suitable match.
- In the example above, the possessive pronoun, **his** , refers to something in the same sentence, i.e. **john** .
- You should also be able to resolve references across sentences:

```
?- run([he,looked,for,it], Refs).
Refs =[john, wallet]
```

This is why we use **assert** . It allows us to store the history so that can resolved references in subsequent sentences.

If you want to reset the history, use the **abolish** built-in predicate:

```
?- abolish(history/1).
```

This will clear the database of the history entries. You'll need to use this when you are testing because **assert** because if you re-run the same code, you'll end up with duplicates in the database.

Resolving a Pronoun

When you come across a pronoun, use the following strategy to find the object it is referring to:

- You'll see in the code that all words have properties associated with them, e.g. number (singular, plural), gender (masculine, feminine, neutral), and a type (person, physical object, abstract object, place). You search the history for **things** that match the properties of the **pronoun** .
- Here is a hint for who to do that for a personal pronoun:

```
personal(Pronoun, Props1),
member(gender(Gender), Props1),
member(number(Number), Props1),
history(thing(Ref, Props2)),
member(gender(Gender), Props2),
member(number(Number), Props2).
```


```
personal(Pronoun, Props1),
member(gender(Gender), Props1),
member(number(Number), Props1),
history(thing(Ref, Props2)),
member(gender(Gender), Props2),
member(number(Number), Props2).
```


The first line of this code looks up the personal pronoun entry, e.g. the entry for **she** , is:

```
personal(she, [number(singlar), gender(feminine)]).
```

so a call:

```
?- personal(she, X).
X = [number(singlar), gender(feminine)]
```

returns the properties of the pronoun. Calling

```
?- member(gender(Gender), [number(singlar), gender(feminine)]).
Gender = feminine
```

returns the gender. The call

```
history(thing(Ref, Props2))
```

will find an entry on the history list and the following calls to **member** , check if this history entry has the same number and gender. if they don't match, Prolog will backtrack until it finds successful match, or else fail. This is the way you can search the history list.

WARNING: This is not a perfect method for resolving references. In fact, there is no perfect method. All reference resolution methods involve heuristics that work a lot of the time, but can fail for some examples. **For the purposes of this assignment, we will accept any code that makes reasonable matches.**