Semantics of Programming Languages
Exercise Sheet 4

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

Exercise 4.1 Rule Inversion

Recall the evenness predicate $ev$ from the lecture:

```isar
inductive $ev :: \text{nat} \Rightarrow \text{bool}$ where
  $ev0$: "$ev\ 0" |
  $evSS$: "$ev\ n \Rightarrow ev\ (Suc\ (Suc\ n))""
```

Prove the converse of rule $evSS$ using rule inversion. Hint: There are two ways to proceed. First, you can write a structured Isar-style proof using the `cases` method:

```isar
lemma "ev\ (Suc\ (Suc\ n)) \Rightarrow ev\ n"
proof -
  assume "ev\ (Suc\ (Suc\ n))" then show "ev\ n"
  proof (cases)
    ...
  qed
qed
```

Optional: Alternatively, you can write a more automated proof by using the `inductive_cases` command to generate elimination rules. These rules can then be used with "auto elim". (If given the `[elim]` attribute, `auto` will use them by default.)

```isar
inductive_cases evSS_elim: "ev\ (Suc\ (Suc\ n))""
```

Next, prove that the natural number three ($Suc\ (Suc\ (Suc\ 0))$) is not even. Hint: You may proceed either with a structured proof, or with an automatic one. An automatic proof may require additional elimination rules from `inductive_cases`.

```isar
lemma "\neg\ ev\ (Suc\ (Suc\ (Suc\ 0)))"
```
Exercise 4.2  (Deterministic) labeled transition systems

A labeled transition system is a directed graph with edge labels. We represent it by a predicate that holds for the edges.

\[
\text{type synonym } (\text{'q,'l}) \text{lts} = \text{"'q } \Rightarrow \text{'l } \Rightarrow \text{ 'q } \Rightarrow \text{ bool"}
\]

I.e., for an LTS \(\delta\) over nodes of type \('q\) and labels of type \('l\), \(\delta \ p \ l \ q\) means that there is an edge from \(p\) to \(q\) labeled with \(l\).

A word from source node \(u\) to target node \(v\) is the sequence of edge labels one encounters when going from \(u\) to \(v\).

Define a predicate \(\text{word}\), such that \(\text{word } \delta \ p \ l \ q\) holds iff \(w\) is a word from \(u\) to \(v\).

\[
\text{inductive } \text{word} :: \text{"('q,'l) lts } \Rightarrow \text{ 'q } \Rightarrow \text{ 'l } \Rightarrow \text{ 'q } \Rightarrow \text{ bool"}
\]

For a deterministic LTS, the same word from the same source node leads to at most one target node, i.e., the target node is determined by the source node and the path.

\[
\text{lemma assumes } \text{det: } \text{"det } \delta\" \\
\text{shows } \text{"word } \delta \ p \ ls \ q\Rightarrow \text{ word } \delta \ p \ ls \ q' \Rightarrow q = q'\" 
\]

Exercise 4.3  Counting Elements

Recall the count function, that counts how often a specified element occurs in a list:

\[
\text{fun } \text{count} :: \text{"'a } \Rightarrow \text{ 'a } \text{list } \Rightarrow \text{ nat" where} \\
\text{"count } x \] = 0" \| \text{"count } x \ (y \# \ ys) = (if } x= y \text{ then Suc (count x } ys) \text{ else count } x \ ys)"
\]

Show that, if an element occurs in the list (its count is positive), the list can be split into a prefix not containing the element, the element itself, and a suffix containing the element one times less

\[
\text{lemma } \text{"count a } xs = \text{Suc } n \Rightarrow \exists ps \ ss. \ xs = ps \@ a \# \ ss \land \text{count a } ps = \text{0 } \land \text{count a } ss = n" 
\]

Homework 4.1  Paths in Graphs

Submission until Sunday, Nov 29, 23:59.

Give all your proofs in Isar, not apply style

A graph is specified by a set of edges: \(E :: ('v \times 'v) \text{set}\). A path in a graph from \(u\) to \(v\) is a list of vertices \([u_1, \ldots, u_n]\) such that \(u = u_1, (u_i, u_{i+1}) \in E,\) and \((u_n, v) \in E\). Moreover, the empty list is a path from any node to itself.
For example, in the graph: \( \{(i, i+1) \mid i \in \mathbb{N}\} \), we have that \([3, 4, 5]\) is a path from 3 to 6, and \([\ ]\) is a path from 1 to 1.

Note that not including the last node of the path into the list simplifies the formalization.

Formalize an inductive predicate \( \text{is\_path} \)

\[
\text{inductive \ is\_path :: "'(v \times 'v) set \Rightarrow 'v \Rightarrow 'v \Rightarrow bool"}
\]

Test your formalization for some examples:

```
lemma "\text{is\_path} \{(i, i+1) \mid i :: \text{nat}. \ True\} \ 3 \ [3, 4, 5] \ 6" 
lemma "\text{is\_path} \{(i, i+1) \mid i :: \text{nat}. \ True\} \ 1 \ [\ ] \ 1"
```

Prove the following two lemmas that allow you to glue together and split paths:

```
theorem \text{path\_appendI}: 
  "[\text{is\_path} \ E \ u \ p1 \ v; \text{is\_path} \ E \ v \ p2 \ w] \implies \text{is\_path} \ E \ u \ (p1 @ p2) \ w"

*Hint: For the next lemma, use induction on \( p1 \) and case analysis.
```

```
theorem \text{path\_appendE}:
  "\text{is\_path} \ E \ u \ (p1 @ p2) \ w \implies \exists \ v. \ \text{is\_path} \ E \ u \ p1 \ v \land \text{is\_path} \ E \ v \ p2 \ w"
```

**Bonus exercise (5 points)**

Bonus points are added to your total, but not to the maximum number of points.

Show that if there is a path from \( u \) to \( w \), then also there exists a path from \( u \) to \( w \) where all the nodes are distinct (using the pre-defined \( \text{distinct} \)).

*Hint: Reason over path length, using the \text{less\_induct} induction rule.

```
thm \text{less\_induct}
```

```
theorem \text{path\_distinct}: 
  "\text{is\_path} \ E \ u \ p \ v \implies \exists \ p'. \ \text{distinct} \ p' \land \text{is\_path} \ E \ u \ p' \ v"
```

Homework 4.2  Grammars

*Submission until Sunday, Nov 29, 23:59.*

Give all your proofs in Isar, not apply style

We define a grammar for strings of the form \( a^n b^n \), where \( a \) and \( b \) are defined via the type \( ab \):

```
datatype \text{ab} = a | b
```

We define the language of all strings of the form \( a^n b^n \) by means of the following rules:

\[
S \rightarrow aSb \mid \epsilon
\]

\text{inductive} \ S :: "\text{ab list} \Rightarrow \text{bool}" \text{ where}

add: “$S w \implies S (a \# w @ [b])$”
| nil: “$S []$”

Your task is to show that the grammar fulfills the informal specification of the language, i.e.

**theorem** $S_{\text{correct}}$:

“$S w \iff (\exists n. w = \text{replicate } n a @ \text{replicate } n b)$”

Here, *replicate* is a pre-defined function, with *replicate* $n x$ producing a list consisting of $n$ copies of $x$. 