Prof. Dr. Peter Koepke, Regula Krapf
A. Pick, A. Ribelles, T. Weiß

Syntax und Semantik der Aussagenlogik

Lösungen zu den Aufgaben

Mit dem Befehl `ocaml` lässt sich der Interpreter starten. Anschließend sollte man über den Befehl (man beachte, dass das Zeichen # hier tatsächlich eingegeben werden muss)

```
#use "init.ml";;
```

alles Nötige einbinden. Um aussagenlogische Formeln parsen zu können, muss der standardmäßig auf *First Order Logic* eingestellte Parser zunächst umgestellt werden:

```
# let default_parser = parse_prop_formula;;
```

Aufgabe 1. Betrachte die Funktionen `odd` und `even` auf Seite 610:

```
# let rec even n = if n = 0 then true else odd (n-1)
  and     odd n = if n = 1 then true else even (n-1);;
```

Teste damit, ob die Zahl 3 gerade oder ungerade ist. Wie sind die Beobachtungen zu erklären?

Lösung 1. Die Rekursion enthält keine Abbruchbedingungen, für den Aufruf von `even 3` wird folgende infinite Rekursion gestartet:

$$\texttt{even 3} \to \texttt{odd 2} \to \texttt{even 1} \to \texttt{odd 0} \to \texttt{even -1} \to \dots$$

Man könnte dies beispielsweise durch Modifaktion der Funktion `odd` so verhindern. Diese Funktion gibt nun für positive ganze Zahlen den richtigen Wahrheitswert aus, für negative grundsätzlich `false`:

```
# let rec even n = if n = 0 then true else odd (n-1)
      and  odd n = if n = 1 then true else
                   if n <= 0 then false else even (n-1);;
```

Aufgabe 2. Schreibe eine rekursive Funktion, die `fib n`, die als Eingabe eine natürliche Zahl bekommt und die n-te Fibonacci-Zahl ausgibt.

Lösung 2. Die lösende Funktion findet sich auf Seite 610 im Buch:

```
# let rec fib n = if n <= 1 then 1 else fib(n-2) + fib(n-1);;
```

Aufgabe 3. Schreibe einen Destruktor `dest_imp` für Aussagen der Form $\varphi \to \psi$, der das **Paar** (φ, ψ) zurückgibt, wobei φ und ψ aussagenlogische Formeln sind. Schreibe anschließend jeweils einen Destruktor `antecedent` bzw. `consequent` für die Implikation, der zu einer Implikation das Antezedenz bzw. das Sukzedens ausgibt.
Schreibe nun einen Destruktor `conjuncts`, der zu einer Aussage der Form $\varphi_1 \wedge \varphi_2 \wedge \dots \varphi_n$ alle Konjunkte in einer Liste zurückgibt. Ist der Input konjunktionsfrei, so soll er direkt ausgegeben werden.
Hinweis: Nutze pattern-matching für diese Aufgaben. Die Funktion, die die Konjunkte liefert, sollte rekursiv implementiert werden.

Lösung 3. Der Destruktor für die Implikation `dest_imp`:

```
# let dest_imp fm =
  match fm with Imp(p,q) -> (p,q) | _ -> failwith "dest_imp";;
```

Antezedens und Sukzedens unter Benutzung der Funktionen `fst` und `snd`, die für ein Paar als Eingabe das erste bzw. zweite Argument zurückgeben:

```
# let antecedent fm = fst(dest_imp fm);;
# let consequent fm = snd(dest_imp fm);;
```

Rekursive Funktion für die Konjunkte:

```
# let rec conjuncts fm =
  match fm with And(p,q) -> conjuncts p @ conjuncts q | _ -> [fm];;
```

Aufgabe 4. Experimentiere mit den Funktionen `print_truthtable`, `dnf` und `cnf`. Alle bekommen eine Formel als Input. Warum könnte die Formel

$$p \wedge (q \vee r) \vee q \wedge r$$

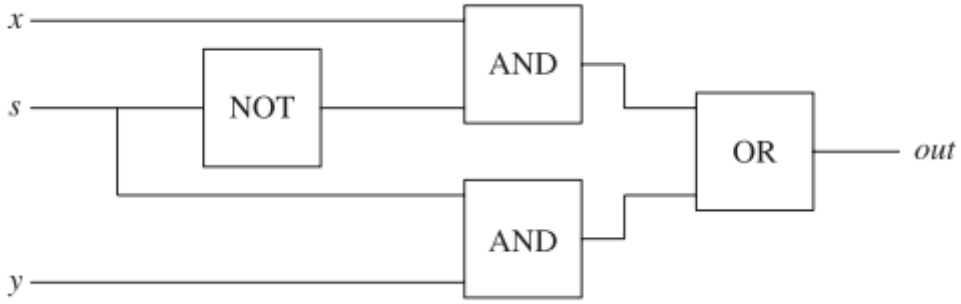
auch als *Majorität* oder *Mehrheitsfunktion* bezeichnet werden?

Lösung 4. Wertet man die Aussage aus, so entspricht ihr Wahrheitswert der *Mehrheit* der Wahrheitswerte ihrer atomaren Aussagen.

```
# let fm = <<p /\ (q \/ r) \/ q /\ r>>;;
val fm : prop formula = <<p /\ (q \/ r) \/ q /\ r>>
# print_truthtable fm;;
p     q     r     | formula
---------------------------
false false false | false
false false true  | false
false true  false | false
false true  true  | true
true  false false | false
true  false true  | true
true  true  false | true
true  true  true  | true
---------------------------
- : unit = ()
# dnf fm;;
- : prop formula = <<p /\ q \/ p /\ r \/ q /\ r>>
# cnf fm;;
- : prop formula = <<(p \/ q) /\ (p \/ r) /\ (q \/ r)>>
```

Aufgabe 5. Übertrage den Schaltkreis auf Seite 64 (s. Abb. 1) in eine logische Formel in interner Darstellung (`Or(p,q)`, `And(p,q)`, usw.), also ohne die Symboldarstellung (mit $\vee, \wedge, \ldots$) zu benutzen. Gib die Wahrheitstabelle aus. Welche Rolle spielt s?

Abbildung 1: Schaltkreis zu Aufgabe 5. S. 64 im Buch.



Lösung 5. Bevor die Member-Funktionen des aussagenlogischen Formeltyps `Or`, `And` und `Not` benutzt werden können, müssen wir Variablen vom Typ `prop formula` erstellen:

```
# let s = Atom(P"s") and x = Atom (P"x") and y = Atom (P"y");;
```

Der Schaltkreis entspricht dann der zusammengesetzten Formel

$$\texttt{Or}(\texttt{And}(\texttt{Not}(\texttt{s}), \texttt{x}), \texttt{And}(\texttt{s}, \texttt{y})) \text{ oder } (\neg s \wedge x) \vee (s \wedge y)$$

Der Schaltkreis stellt einen sogenannten *Multiplexer* dar: Die Variable `s` ist heißt *Schaltvariable*: Sie entscheidet, ob entweder `x` weitergeleitet wird oder `y`. Dementsprechend hat die obige Formel den gleichen Wert wie `x`, wenn `s = false` ist, und den Wert von `y`, falls `s = true`.

Aufgabe 6. Betrachte die Funktion `tautology`:

```
# let tautology fm = onallvaluations (eval fm) (fun s -> false)
  (atoms fm);;
```

Diese Funktion überprüft, ob die übergebene Formel eine Tautologie ist. Schreibe nun Funktionen, die für eine gegebene Formel überprüfen, ob diese erfüllbar bzw. unerfüllbar ist.
Hinweis: Es ist einfacher, zunächst auf Unerfüllbarkeit zu testen und dafür den schon bekannten Code zu modifizieren.

Lösung 6. Folgende Funktionen lösen die Aufgabe:

```
# let unsatisfiable fm = tautology(Not fm);;
# let satisfiable fm = not(unsatisfiable fm);;
```

Aufgabe 7. Experimentiere mit der Funktion `psubst`. Was geschieht, wenn man Formel substitutieren möchte, die nicht atomar sind?

```
# let psubst subfn = onatoms (fun p -> tryapplyd subfn p (Atom p));;
```

Lösung 7. Wegen der `onatoms`-Funktion als Subroutine wirkt `psubst` nur auf Atome:

```
# psubst (P"p" |=> <<p /\ q>>) <<p /\ q /\ p /\ q>>;;
- : prop formula = <<(p /\ q) /\ q /\ (p /\ q) /\ q>>
```

Wenn man versucht, eine Formel zu substituieren und diese entsprechend mit französischen Anführungszeichen im `prop formula` Typ eingibt, wird eine entsprechende Fehlermeldung wegen falschen Datentyps zurückgegeben und das Programm nicht ausgeführt:

```
# psubst (<<~p>> |=> <<p /\ q>>) <<~p /\ ~ ~p>>;;
```

produziert diese Fehlermeldung:

```
This expression has type (prop formula, prop formula) func
but is here used with type (prop formula, prop formula formula) func
```

Möchte man allerdings dem Parser vorgaukeln, die komplexe Formel sei ein Atom, so wird diese Irreführung auf Userseite schlicht ignoriert, und die Formel, in der substituiert werden sollte, ohne jede Substitution, aber auch ohne jede Fehlermeldung ausgegeben:

```
# psubst (P"~p" |=> <<p /\ q>>) <<~p /\ ~ ~p>>;;
- : prop formula = <<~p /\ ~(~p)>>
```

Aufgabe 8. Sei `fm` eine aussagenlogische Formel. Die zu `fm` *duale* Formel entsteht, indem man die Wahrheitswerte aller Atome umdreht und jedes $\wedge$ durch ein $\vee$ und umgekehrt simultan ersetzt. Implementiere eine Funktion, die zu einer gegebenen Formel die duale Formel ausgibt.
Hinweis: pattern-matching könnte hilfreich sein.

Lösung 8. Folgende Funktion transformiert eine übergebene aussagenlogische Formel in die zu ihr duale Formel.

```
# let rec dual fm =
  match fm with
    False -> True
  | True -> False
  | Atom(p) -> fm
  | Not(p) -> Not(dual p)
  | And(p,q) -> Or(dual p,dual q)
  | Or(p,q) -> And(dual p,dual q)
  | _ -> failwith "Formula involves connectives ==> or <=>";;
```