12. Übungsblatt zu Informatik 1 WS 98/99

Andreas Rossberg, Prof. Gert Smolka

Abgabe: 5. Februar 1999 vor der Vorlesung

Hinweis: Alle hier angegebenen Signaturen finden Sie unter `http://www.ps.uni-sb.de/courses/info1-ws98/uebung/sml/blatt12/`.

Bitte alle Programme per Email abgeben.

Aufgabe 12.1: Schalter (5 Punkte)

(a) Schreiben Sie eine Prozedur `toggle: unit -> bool`, die bei jedem Aufruf die Negation des vorherigen Ergebnisses zurückliefert. Beispielsweise:

```
- (toggle(),toggle(),toggle(),toggle());
> val it = (true,false,true,false) : bool * bool * bool * bool
```

(b) Schreiben Sie eine Prozedur `newToggle: unit -> unit -> bool`, die bei jedem Aufruf eine neue Schalterprozedur wie in Teil (a) erzeugt.

Aufgabe 12.2: Schlange mit Prioritäten (8 Punkte)

Eine Schlange mit Prioritäten (*priority queue*) ist eine Schlange, in der jedes Element mit einer Priorität versehen ist. Die Prozedur `get` liefert dann nicht einfach das älteste Element aus der Schlange, sondern das älteste mit der höchsten Priorität. Neue Elemente dürfen sich also 'vordrängeln', wenn ihre Priorität höher ist.

Schreiben Sie ein Modul `PQueue`, das Schlangen mit Prioritäten mit folgender Signatur verwirklicht:

```
type 'a pqueue

exception Empty

val new :      unit -> 'a pqueue
val put :      'a pqueue * 'a * int -> unit
val get :      'a pqueue -> 'a                (* Empty *)
val isEmpty :  'a pqueue -> bool
```

Der Prozedur `put` wird neben dem einzufügenden Element auch dessen Priorität in Form einer ganzen Zahl übergeben. Hohe Zahlen sollen hohe Priorität bedeuten. Zum Beispiel:

```
- open PQueue;
> ...
- val p = new() : string pqueue;
> ...
- put(p,"Horf",7); put(p,"Gismalda",3); put(p,"Pazo",6);
> ...
- (get p, get p, get p);
> val it = ("Horf","Pazo","Gismalda") : string * string * string
- isEmpty p;
> val it = true : bool
```

Aufgabe 12.3: Zirkuläre Listen (5+4 Punkte)

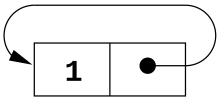
Mithilfe von Referenzen lassen sich zirkuläre Datenstrukturen aufbauen. Betrachten Sie dazu die Deklarationen

```
datatype 'a rlist = RNil | RCons of 'a * 'a rlist ref

fun head(RCons(x,_)) = x
fun tail(RCons(_,r)) = r
```

Nun kann man eine zirkuläre Liste wie folgt erzeugen:

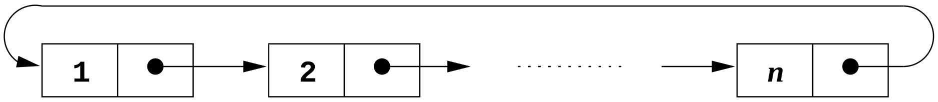
```
val l = RCons(3,ref RNil)
val _ = tail l := l
```



Die nebenstehende Abbildung veranschaulicht den Aufbau dieser Liste.

(a) Geben Sie Deklarationen an, die zwei Listen `l1` und `l2` erzeugen mit der Eigenschaft `!(tail l1) = l2` und `!(tail l2) = l1`.

(b) Schreiben Sie eine Prozedur `cycle: int -> int rlist`, so dass `cycle` n für $n > 0$ eine zirkuläre Liste mit den Elementen 1 bis n liefert:



(c∗) Schreiben Sie eine Prozedur `isCyclic: 'a rlist -> bool`, die ermittelt, ob ihr Argument eine zirkuläre Liste ist. Die Prozedur arbeitet wie folgt: sie iteriert mit zwei Zeigern (Referenzen) über die Liste. Der eine Zeiger wird bei jedem Rekursionsschritt um ein Element weitergesetzt, der andere um zwei. Erreicht man das Listenende, so ist die Liste offensichtlich nicht zirkulär. Stellt man hingegen irgendwann fest, dass beide Zeiger auf dieselbe Zelle verweisen, so ist die Liste zirkulär (der 'schnelle' Zeiger hat den langsamen 'überrundet').

Aufgabe 12.4: Parser mit rekursivem Abstieg (12 Punkte)

Die Grammatik von MML-Ausdrücken mit `let` und `val` sei wie folgt gegeben:

$$\begin{array}{rcl} e & ::= & c \\ & \mid & x \\ & \mid & x\,(es) \\ & \mid & \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & \texttt{let}\ ds\ \texttt{in}\ e\ \texttt{end} \\ d & ::= & \texttt{val}\ x\ \texttt{=}\ e \\ & \mid & \texttt{fun}\ x\,(xs)\ \texttt{=}\ e \end{array}$$

$$\begin{array}{rcl} b & ::= & e_1\ \texttt{=}\ e_2 \\ & \mid & e_1\ \texttt{<}\ e_2 \\ & \mid & e_1\ \texttt{<=}\ e_2 \\ xs & ::= & x \mid x\,\texttt{,}xs \\ es & ::= & e \mid e\,\texttt{,}es \\ ds & ::= & d \mid d\ ds \end{array}$$

Schreiben Sie ein Modul `Parser` mit folgender Signatur

```
exception Error

val parse: Token.token list -> Phrase.expression    (* Error *)
```

das die syntaktische Analyse für diese Grammatik mittels rekursivem Abstieg realisiert. Die Module `Token` und `Phrase`, die die Typen für Token und für die zu generierende abstrakte Syntax definieren, finden sie im Web.