12. Übungsblatt „Programmiersprachen“ WS 97/98

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 5. Februar 1998 **Abgabe:** 16. Februar 1998

XII.1: Klassen in Oz (6 Punkte)

Schreiben Sie eine Oz-Klasse `Queue`, die eine Schlange implementiert. Diese Klasse soll die Methoden `put(X)` sowie `get(X)` verstehen; weiterhin soll die Methode `getLength(L)` ihr Argument an einen Integer binden, der die Anzahl der momentan in der Queue enthaltenen Elemente angibt.

Implementieren Sie die Schlange effizient durch einen Strom und zwei Zeiger auf dessen Kopf und Schwanz, so daß alle Operationen konstante Zeit brauchen.

XII.2: Beschränkter Puffer mit synchronen Kanälen (3 + 4 Punkte)

a. Implementieren Sie eine „Pumpe“, die zwei synchrone Kanäle verbindet, indem sie einfach die vom einen Kanal gelesenen Elemente in den anderen schreibt. Die Prozedur

```
proc {NewPump Get Put}
```

erhält die `Get`-Operation des ersten Kanals sowie die `Put`-Operation des anderen als Eingabe und startet in einem neuen Thread die entsprechende Endlosschleife.

b. Schreiben Sie eine Prozedur

```
proc {NewBoundedBuffer K Put Get}
```

die einen k-beschränkten Puffer mit den Zugriffsoperationen `Put` und `Get` erzeugt. Der Puffer soll dabei durch eine Folge von synchronen Kanälen realisiert sein, die durch Pumpen verbunden sind.

XII.3: Faule funktionale Programmierung (6 Punkte)

In dieser Aufgabe soll der Strom der *Hamming-Zahlen* erzeugt werden. Hierbei sollen alle ganzen Zahlen der Form $2^i 3^j 5^k$ in aufsteigender Reihenfolge aufgezählt werden, wobei i, j und k nichtnegative ganze Zahlen sind. Die ersten zehn Hamming-Zahlen sind also $1, 2, 3, 4, 5, 6, 8, 9, 10, 12$.

Man sieht leicht, daß die Menge der Hamming-Zahlen die eindeutige Lösung der rekursiven Gleichung

$$H = \{1\} \cup \{2h \mid h \in H\} \cup \{3h \mid h \in H\} \cup \{5h \mid h \in H\}$$

ist. Demzufolge ist die Liste der Hamming-Zahlen in aufsteigender Reihenfolge die eindeutige Lösung der Gleichung

```
Hs = 1|{Merge {Merge {Times Hs 2} {Times Hs 3}} {Times Hs 5}}
```

wobei `Merge` eine Funktion ist, die zwei Integer-Listen in aufsteigender Reihenfolge unter Vermeidung von Verdopplungen vereinigt, und `Times` eine Funktion, die alle Elemente einer Liste mit einer gegebenen Zahl multipliziert.

Implementieren Sie `Merge` und `Times`.

Hinweis: Verwenden Sie hierzu `Lazy.apply`. Dieses Primitiv ist nur in der Mozart-Implementierung von Oz verfügbar.

Beispiel: Die Eingabe

```
{Browse {Nth Hs 105}}
```

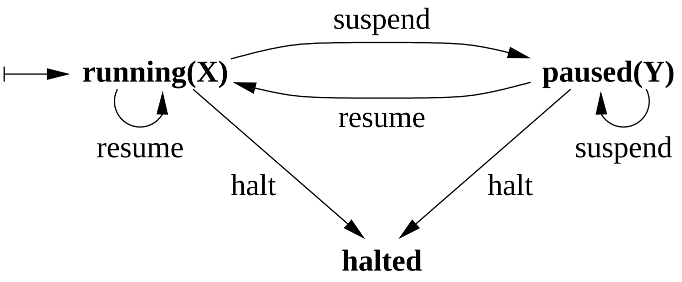
bewirkt, daß die Zahl 1875, also die 105te Hamming-Zahl, im Browser ausgegeben wird. (Die Prozedur `Nth` liefert das n-te Element der Eingabeliste zurück.)

XII.4: Wiederholer (14 Punkte)

Schreiben Sie eine Prozedur

```
proc {NewRepeater Action DelayTime Repeater}
```

die einen Wiederholer für die nullstellige Prozedur `Action` mit der Periode `DelayTime` erzeugt und `Repeater` an einen Record `repeater(suspend: ... resume: ... halt: ...)` bindet. Die Komponenten dieses Records sind nullstellige Prozeduren, die folgende Zustandsübergänge bewirken (die Bedeutung der Variablen `X` und `Y` wird später erläutert):



Beispielsweise soll nach der Eingabe

```
declare R in
thread
   R = {NewRepeater proc {$} {Browse action} end 1000}
   {Browse finished}
end
```

im Browser jede Sekunde `action` ausgegeben werden. Die Eingabe `{R.halt}` soll den Wiederholer terminieren, was die Ausgabe von `finished` bewirkt.

`DelayTime` bestimmt die Zeit, die zwischen dem Ausführungs*beginn* aufeinanderfolgender Aktionen verstreichen soll. Überschreitet die Ausführungsdauer einer Aktion `DelayTime`, soll sofort nach dieser Aktion mit der Ausführung der nächsten begonnen werden. Die Ausführung zweier Aktionen darf sich jedoch nie überlappen.

Das tatsächliche Verlassen eines Zustands als Reaktion auf eine der Operationen soll immer zum nächstmöglichen Zeitpunkt stattfinden, ohne dabei eine laufende Aktion zu unterbrechen. Ein eventuell aktives `Delay` darf jedoch nicht mehr abgewartet werden.

Hinweis: Um dieses Verhalten zu realisieren, haben wir im obigen Diagramm die Synchronisationsvariablen `X` beziehungsweise `Y` vorgesehen, die die betroffenen Zustände mit sich führen. Die Operationen von `Repeater` binden diese an **`unit`**, wenn der jeweilige Zustand zu verlassen ist.

Verwenden Sie das Primitiv `{WaitOr A B}`, das solange blockiert, bis mindestens eines der Argumente determiniert ist.