5. Übungsblatt „Programmiersprachen“ WS 97/98

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 27. November 1997 **Abgabe:** 8. Dezember 1997

In den Aufgaben dieses Übungsblattes soll ein kleiner Teil von Scheme in SML eingebettet werden, nämlich einige der Standard-Datentypen von Scheme sowie Operationen auf diesen. Wir legen folgende Datentypdefinition für die SML-Repräsentation von Scheme-Werten zugrunde:

```
datatype SV =
    Int of int                      (* Ganze Zahl *)
  | Rat of (int * int)              (* Rationale Zahl *)
  | Bool of bool                    (* Boolescher Wert *)
  | Nil                             (* Leere Liste *)
  | Cons of (SV ref * SV ref)       (* Cons-Zelle *)
  | Func of (SV list -> SV) ref     (* Prozedur *)
  | Unspecified                     (* Unspezif. Ergebnis *)
```

Hier erlauben wir – im Gegensatz zu den in der Vorlesung vorgestellten Teilsprachen von Scheme – auch rationale Zahlen, jeweils dargestellt als Paar aus Zähler und Nenner, wobei Zähler und Nenner teilerfremd sein sollen und der Nenner echt größer als 1 sein soll. Werte der Typen `Int` und `Rat` können in arithmetischen Operationen beliebig kombiniert werden.

Scheme-Prozeduren (Konstruktor `Func`) werden in Referenzen gespeichert, damit der Vergleich zweier Prozeduren genau dann `true` liefert, wenn es sich um dieselben Speicherobjekte handelt.

Bemerkung: In allen Aufgaben soll gelten: Wird eine Operation auf Daten falschen Typs angewendet, so soll die Ausnahme `RunTimeError` geworfen werden.

Diese Aufgaben sind übrigens von Robert Harper gestellt worden.

Aufgabe V.1: Addition (8 Punkte)

Schreiben Sie eine Funktion `++` vom Typ `SV list -> SV`, die eine Liste von Scheme-Werten aufaddiert. Zulässig sind ganze und rationale Zahlen. Das Ergebnis muß exakt berechnet werden (also dürfen rationale Zahlen nicht dividiert und gerundet werden). Zum Beispiel soll gelten

`++ [Int 1, Rat (1, 2), Int 2]` $\Rightarrow$ `Rat (7, 2)`

Ist das Ergebnis eine rationale Zahl, so sollen die Invarianten ihrer Repräsentation wiederhergestellt werden (also soll sie gekürzt werden und der Nenner echt größer als 1 sein; ist sie eine ganze Zahl, so soll sie auch als solche dargestellt werden).

Aufgabe V.2: Listenkonstruktion (4 Punkte)

Schreiben Sie eine Funktion `cons` vom Typ `(SV * SV) -> SV` und Funktionen `null`, `car` und `cdr` vom Typ `SV -> SV`. Diese sollen sich genau wie die entsprechenden Operationen in Scheme-5 verhalten. Sie können die Funktionen mit folgendem Programm testen:

```
fun Srev L =
    let
        fun Srev' (L, acc) =
            case (null L)
                of Bool true => acc
               | _ => Srev' (cdr L, cons (car L, acc))
    in
        Srev' (L, Nil)
    end
val x1 = cons (Int 3, Nil);
val x2 = cons (Int 2, x1);
val x3 = cons (Int 1, x2);
val y = Srev x3
```

Aufgabe V.3: Destruktive Operationen auf Listen (4 Punkte)

Implementieren Sie Funktionen `setcar` und `setcdr` vom Typ `SV * SV -> SV`, die sich genau wie die entsprechenden Scheme-Operationen verhalten.

Aufgabe V.4: Vergleichsoperationen (6 Punkte)

Implementieren Sie die Vergleichsoperationen `eqv` und `equal` vom Typ `SV * SV -> SV`. Sowohl `eqv` als auch `equal` geben *wahr* zurück, wenn eine der folgenden Bedingungen erfüllt ist:

1. Beide Argumente sind boolesche Werte und beide sind entweder *wahr* oder *falsch*.
2. Beide Argumente sind Zahlen mit demselben Zahlenwert. (Es können insbesondere ganze Zahlen mit rationalen Zahlen verglichen werden.)
3. Beide Argumente sind die leere Liste.
4. Beide Argumente sind Prozeduren und stellen dasselbe Objekt im Speicher dar.

In dem Fall, daß beide Argumente cons-Zellen sind, liefert `eqv` genau dann *wahr*, wenn die Lokationen der cons-Zelle übereinstimmen, wohingegen `equal` dann *wahr* liefert, wenn die Werte, die in diesen Lokationen gespeichert sind, im Sinne von `equal` übereinstimmen. In allen anderen Fällen sollen `eqv` und `equal` den Wert *falsch* liefern.

Aufgabe V.5: Argumentübergabe (5 Punkte)

Bisher war es so, daß alle Funktionen (bis auf `++`) ihre Argumente als Tupel von Scheme-Werten erhielten. Dies soll jetzt dahingehend geändert werden, daß die Argumente als Listen übergeben werden. Jede Funktion muß selbst überprüfen, ob die Anzahl der ihr übergebenen Argumente stimmt, und gegebenenfalls eine `RunTimeError`-Ausnahme werfen.

Statt alle Funktionen umzuschreiben, sollen Sie in dieser Aufgabe zwei Funktionen

```
val onearg: (SV -> SV) -> (SV list -> SV)
val twoarg: (SV * SV -> SV) -> (SV list -> SV)
```

implementieren. Jede liefert zu einer gegebenen Funktion f eine Funktion f' zurück, die statt eines Tupels eine Liste von Scheme-Werten erhält, die Länge dieser Liste überprüft und f entsprechend aufruft. Beispielsweise sollen gelten:

```
(twoarg eq) [v]      ⇒  raise RunTimeError
(twoarg eq) [v, w]   ⇒  eq (v, w)
```

Mithilfe dieser Funktionen sollen Sie nun alle oben definierten einstelligen und zweistelligen Funktionen umdefinieren, so daß sie statt Tupeln Listen von Scheme-Werten erhalten. Die neue Version von `cons` beispielsweise soll dann den Typ `SV list -> SV` haben.

Aufgabe V.6: Applikation (6 Punkte)

In Scheme gibt es eine Prozedur `apply`, die zwei Argumente erhält, eine Prozedur und eine Liste, und die Prozedur auf die durch die Liste gegebenen Argumente anwendet. Wenn das erste Argument keine Prozedur oder das zweite Argument keine Liste ist, gibt es einen Laufzeitfehler.

Verwenden Sie `twoarg` und sonstige Hilfsfunktionen, die Sie eventuell benötigen, um die Funktion `apply` vom Typ `SV list -> SV` in SML zu definieren. Bedenken Sie, daß alle in SML modellierten Scheme-Prozeduren den Typ `SV list -> SV` haben, also eine SML-Liste als Eingabe erhalten, wohingegen das zweite Argument von `apply` eine Liste in der Scheme-Repräsentation ist.

Sie können Ihre Implementierung an folgendem Programm testen:

```
val l1 = cons [Int 3, Nil];
val l2 = cons [Int 2, l1];
val l3 = cons [Int 1, l2];
val c = Func (ref ++);
val z = apply [c, l3]
```