Hochschule München
Fakultät für Informatik und Mathematik
M. Ruckert

Theoretische Informatik II
Übungen

Aufgabe 23. Implementieren Sie einen Scheme Interpreter. Nach den Vorarbeiten aus den vorhergehenden Übungen bleibt nicht mehr viel zu tun.

- Primitive Funktionen.

  Einige Funktionen müssen in den Interpreter direkt eingebaut werden. Da auch Funktionen wie `if` implementiert werden müssen erhalten sie ihre Argumente *unevaluiert*. Zusätzlich brauchen sie noch eine gültige Umgebung um die Werte von Variablen zu bestimmen. Eine primitive Funktion wird natürlich auch im simulierten Memory gespeichert. Als Tag verwenden wir `primitiv` und daneben braucht es nur noch die Funktion selbst. Würde man den Interpreter in C schreiben wäre hier ein Zeiger auf eine Funktion, die Teil des Interpreters ist; in Scheme speichern wir eine entspechende Scheme Funktion.

  Wir implementieren die folgenden Funktionen:

  - `(new-primitive f)`
    Diese Funktion erzeugt eine neue primitive Funktion im simulierten Speicher.
  - `(primitive->f p)`
    Extrahiert aus einer primitiven Funktion den Ausführbaren Code.
  - `(add-primitive name f)`
    Eine nützliche Funktion, die eine primitive Funktion erzeugt und ein Symbol mit dem gegebenen Namen und die entsprechende Bindung der globalen Umgebung `i-environment` hinzufügt.
  - Schreiben Sie eine Reihe von primitiven Funktionen (`+`, `-`, `if`, `define`, `quote`, etc.) — `lambda` heben wir uns noch für später auf — und fügen Sie diese zur globalen Umgebung hinzu. Etwa so:

```
(define (i-plus env values) ; values ist eine i-scheme liste
  (new-number
    (let loop ((sum 0); Akumulator für die Summe
               (v values)) ; Iteration über die i-scheme liste
      (if (eq? (tag v) 'pair)
        (loop (+ sum (number->value (i-eval env (pair->car v))))
              (pair->cdr v))
        sum))))

(add-primitive '+ i-plus)
```

- Eval und Apply

  Diese Funktionen sind zentral für den Interpreter.

  - `(i-eval env exp)`
    Die Funktion `i-eval` bekommt eine Umgebung und einen Ausdruck (beides im simulierten Speicher) und berechnet daraus einen Wert (ebenfalls im simulierten Speicher). Die Möglichkeiten sind dabei eher beschränkt. Werte von Variablen werden in der Umgebung nachgeschlagen. Bei Listen muss man nur das erste Argument evaluieren, der Rest ist ein Fall für `i-apply`. Konstante (Zahlen, Symbole, etc.) evaluieren zu sich selbst.
  - `(i-apply env func arglist)`
    Die Funktion `i-apply` bekommt eine Umgebung eine Funktion und eine Liste von Argumenten und muss die Funktion auf die Argumente anwenden. Untersucht man das Tag der Funktion kann es sich (derzeit) nur um eine primitive Funktion handlen (Funktion die mit Lambda definiert wurden kommen später noch dazu). Bei einer primitiven Funktion wird der Code extrahiert und mit der Umgebung und der Argumentliste aufgerufen.
  - Nun können sie auch primitive Funktionen für eval und apply zum Interpreter hinzufügen.

- Vervollständigen Sie den Scheme interpreter mit einem `read-eval-print` loop, etwa so:

```
(define (read-eval-print return)
  (define (i-exit env values) (return 0))
  (add-primitive 'exit i-exit)
  (let loop ()
    (newline)
    (display "i-scheme> ")
    (i-display (i-eval i-environment (i-read)))
    (loop)))

(define (i-scheme)
  (display "This is i-scheme version 1.0")
  (call-with-current-continuation read-eval-print)
  (display "Bye!")
  (newline))
```