3. Übungsblatt „Programmiersprachen“ WS 97/98

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 13. November 1997 **Abgabe:** 20. November 1997

Aufgabe III.1: Linearisieren von Ausdrücken (5 Punkte)

In dieser Aufgabe sollen Scheme-Werte, die ausschließlich aus echten Listen und Symbolen bestehen, in ihre externe Darstellung als Wortfolge überführt werden. Diese sogenannten F-Ausdrücke sind eine Teilmenge der E-Ausdrücke und wie folgt definiert:

$$
\begin{array}{lcll}
F & ::= & S & \\
 & \mid & (F_1 \ldots F_k) & \quad \text{für } k \geq 0
\end{array}
$$

Die Worte sollen wie in folgender Tabelle gezeigt repräsentiert werden:

Wort	Darstellung
`(`	(open-paren)
`)`	(close-paren)
S	(symbol S)

Schreiben Sie eine Prozedur `unparse`, die dieses leistet. Beispielsweise soll gelten:

```
(unparse 'x)         ⇒  '((symbol x))
(unparse '(a (b)))   ⇒  '((open-paren) (symbol a) (open-paren)
                           (symbol b) (close-paren) (close-paren))
```

Aufgabe III.2: Syntaxanalyse von Ausdrücken (7 Punkte)

Schreiben Sie eine Prozedur `parse`, die sich invers zu der obigen Prozedur `unparse` verhält. Eingabe ist also eine Wortfolge, die einen F-Ausdruck beschreibt; Ausgabe soll der zugehörige Wert sein. Entspricht die Wortfolge keinem F-Ausdruck, so soll ein Fehler gemeldet werden. Hierzu gibt es in Scheme die vordefinierte Prozedur `error`:

```
(error "parse error")
```

Hinweise: Verwenden Sie einen globalen Zeiger in die Wortfolge, den Sie bei Bedarf über `set!` weiterrücken lassen. Entsprechend der in Aufgabe 1 angegebenen Grammatik für F-Ausdrücke werden Sie zwei verschränkt rekursive Prozeduren benötigen.

Aufgabe III.3: Klassen (5 Punkte)

Implementieren Sie eine Klasse `new-queue` nach dem in der Vorlesung angegebenen Schema. Objekte dieser Klasse realisieren den abstrakten Datentyp *Schlange*. Stellen Sie dabei folgende Methoden zur Verfügung:

Methodenname	Stelligkeit	Bedeutung
`enqueue`	1	hängt ein Element an die Schlange an (die Rückgabe ist unspezifiziert)
`dequeue`	0	liefert das erste Element der Schlange und entfernt es dabei aus dieser
`is-empty`	0	`#t`, falls die Schlange leer ist, sonst `#f`

Hinweis: Diese Repräsentation für Klassen ist in Abelson und Sussman, Structure and Interpretation of Computer Programs, ausführlich beschrieben.

Aufgabe III.4: Metaprogrammierung (12 + 4 Punkte)

Betrachten Sie die Teilsprache von Scheme, in der Ausdrücke nach folgenden Regeln gebildet werden können:

$$
\begin{array}{rcl}
A & ::= & L \\
 & | & X \\
 & | & (\texttt{lambda}\ (X)\ A) \\
 & | & (\texttt{if}\ A_1\ A_2\ A_3) \\
 & | & (A_1\ A_2) \\
L & ::= & \texttt{\#t} \mid \texttt{\#f} \\
 & | & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots
\end{array}
$$

Beachten Sie, daß diese Sprache nur einstellige Prozeduren und Applikationen erlaubt. Die Semantik ist durch die unten angegebenen Transitionen definiert.

$$
\begin{array}{ll}
\text{(lit)} & L, U \to L \\[1ex]
\text{(var)} & X, U \to W, \text{ falls } W = U(X) \\[1ex]
\text{(abs)} & (\textbf{lambda}\ (X)\ A), U \to (\textbf{lambda}\ (X)\ A)\ []\ U \\[1ex]
\text{(if-t)} & \dfrac{A_1, U \to \texttt{\#t} \quad A_2, U \to W}{(\textbf{if}\ A_1\ A_2\ A_3), U \to W} \\[2ex]
\text{(if-f)} & \dfrac{A_1, U \to \texttt{\#f} \quad A_3, U \to W}{(\textbf{if}\ A_1\ A_2\ A_3), U \to W} \\[2ex]
\text{(op)} & \dfrac{A_1, U \to O \quad A_2, U \to W_2}{(A_1\ A_2), U \to W} \qquad W = O(W_2) \\[2ex]
\text{(app)} & \dfrac{A_1, U \to (\textbf{lambda}\ (X)\ A)\ []\ V \quad A_2, U \to W_2 \quad A, V[W_2/X] \to W}{(A_1\ A_2), U \to W}
\end{array}
$$

a. Schreiben Sie einen Interpreter, der einen Ausdruck A und eine Umgebung U erhält. U sei als Assoziationsliste dargestellt. Literale werden intern durch die entsprechenden Scheme-Literale repräsentiert, Abstraktionen durch Paare aus Lambda-Ausdruck und Umgebung, Operationen durch Scheme-Prozeduren. Beispielsweise evaluiert der Scheme-Ausdruck

```
(cons (cons 'x 10)
      (cons (cons 'y (cons '(lambda (x) x) ()))
            (cons (cons '+ (lambda (x) (lambda (y) (+ x y))))
                  ())))
```

zu einer Umgebung, in der die Variable `x` an den Integer-Wert 10 gebunden ist, `y` an eine benutzerdefinierte Abstraktion mit leerer Umgebung und `+` an die Operation „Addition" (denn der Lambda-Ausdruck war nicht gequotet und wird daher ausgewertet!). Der Ausdruck

```
((+ 7) (y x))
```

beispielsweise würde in dieser Umgebung zu dem Integer-Wert 17 ausgewertet.

Zusätzlich zu dem Resultat soll der Interpreter noch den zugehörigen Beweisbaum ausgeben: Regeln ohne Prämissen bilden dessen Blätter, die einfach aus dem Namen der Regel als Symbol bestehen; Regeln mit Prämissen sind Knoten, die mit den Namen der Regel markiert sind und als Unterbäume die Beweisbäume für die Prämissen enthalten. Das Resultat der Auswertung obigen Ausdrucks etwa wäre folgende Liste mit zwei Elementen (dem Ergebnis und dem Beweisbaum):

```
(17 (op (op var lit)
        (app var var var)))
```

Das Grundgerüst des Interpreters ist vorgegeben (siehe nächste Seite oder die Seite zur Vorlesung im WWW). Beispielhaft ist die (lit)-Regel implementiert; die anderen Regeln sollen Sie implementieren. Im Gerüst mit enthalten sind Prozeduren für den Aufbau des Beweisbaumes – diese verlangen lediglich, daß *nach* Auswertung der Prämissen der Regelname auf einen Stack gelegt wird:

`(proof-push '`⟨Regelname⟩`)`

b. Schreiben Sie einen Ausdruck in der oben definierten Sprache, der die Fakultätsfunktion implementiert und diese auf die Eingabe „3" anwendet. Stellen Sie den zugehörigen Beweisbaum dar.

```scheme
;; Aufbau des Beweisbaums

(define proof-stack ())

(define proof-push
  (lambda (rule)
    (set! proof-stack (cons rule proof-stack))))

(define proof-stack-to-tree
  (lambda ()
    (let ((rule (car proof-stack)))
      (begin
        (set! proof-stack (cdr proof-stack))
        (case rule
          ((var lit abs)
           rule)
          ((if-f if-t op)
           (let ((subtree2 (proof-stack-to-tree)))
             (let ((subtree1 (proof-stack-to-tree)))
               (list rule subtree1 subtree2))))
          ((app)
           (let ((subtree3 (proof-stack-to-tree)))
             (let ((subtree2 (proof-stack-to-tree)))
               (let ((subtree1 (proof-stack-to-tree)))
                 (list rule subtree1 subtree2 subtree3))))))))))

;; Transitionsregeln

(define literal?
  (lambda (a)
    (or (integer? a) (boolean? a))))
(define interpret-literal
  (lambda (a u)
    (proof-push 'lit)
    a))

(define variable? ...)
(define interpret-variable ...)

(define conditional? ...)
(define interpret-conditional ...)

(define abstraction? ...)
(define interpret-abstraction ...)

(define application? ...)
(define interpret-application ...)

;; Hauptfunktionen

(define interpret
  (lambda (a u)
    (cond ((literal?     a) (interpret-literal     a u))
          ((variable?    a) (interpret-variable    a u))
          ((conditional? a) (interpret-conditional a u))
          ((abstraction? a) (interpret-abstraction a u))
          ((application? a) (interpret-application a u))
          (else
           (error "illegal␣expression␣syntax")))))

(define global-environment
  (list (cons '+ (lambda (x) (lambda (y) (+ x y))))
        (cons '- (lambda (x) (lambda (y) (- x y))))
        (cons '* (lambda (x) (lambda (y) (* x y))))
        (cons '< (lambda (x) (lambda (y) (< x y))))))

(define start
  (lambda (a)
    (set! proof-stack ())
    (let ((res (interpret a global-environment)))
      (list res (proof-stack-to-tree)))))
```