Funktionale Programmierung

```
( λ(x) . x + 2 )
      ( (λ(x) . x² )
            3)                    ⇒ 11
```

Scheme Wertesemantik

- Konstanten haben sich selbst als Wert: `2 ==> 2` `#t ==> #t`
- Variablen haben den Wert, der Ihnen (z.B. mit define) zugewiesen wurde

```
x ==> error: unbound variable
(define x 2) ==> #undefined
x ==> 2
```

- der Wert von s-Ausdrücken muss berechnet werden (z.B. durch funktionale Applikation)

Variablen

- Ein Name identifiziert eine Variable (Symbol), deren Wert ein (abstraktes) Objektist.
- Benennung als einfachste Form der Abstraktion: Man sieht einem Wert nicht mehr an, wie er entstanden ist.

```
(   define   <name>   <konstante>   )
(   define   <name>   <s-expression>   )
```

- define ist eine special form expression
- Wert des define-Aufrufs ist undefiniert

s-Ausdrücke

Funktionsaufrufe (in Präfix-Notation)

```
( <Funktionsname>   .   <Liste von Argumenten> )
(sqrt (+ (expt x 2) (expt y 2)))
```

lambda-ausdrücke

```
(lambda <Liste von Argumenten   .   <Liste von s-Ausdrücken> )
(lambda (x y) (sqrt (+ (expt x 2) (expt y 2))))
```

Verwendung von lambda-Ausdrücken zur Definition einer benannten Funktion

```
(define pythagoras (lambda (x y)
      (sqrt (+ (expt x 2) (expt y 2)))))
```

Kurzversion ohne lambda

```
(define (pythagoras x y)
      (sqrt (+ (expt x 2) (expt y 2))))
```

Funktionsauswertung

- eval: Auswerten aller Listenelemente
- apply: Anwenden des Evaluationsergebnisses für das erste Listenelement auf die Evaluationsergebnisse der Restliste

• Funktionsdefinition:

```
(define (square x) (* x x))
```

• Funktionsanwendung:

```
( square (+ 3 1 2))   => 36

square          => (lambda (x) (* x x) )
(+ 3 1 2)  => 6
```

Umgebungen

lokale Umgebungen können zusätzliche Variable einführen, die nicht formale Parameter sind

• (let (<Name-Wert-Paar$_1$> ... [<Name-Wert-Paarn>) <Funktionsaufruf>)

• falls ein Ausdruck in Name-Wert-Paar$_i$ auf einen Namen in einem Name-WertPaar$_j$ mit j < i Bezug nimmt, muss statt let let* verwendet werden.

Funktionale Applikation

• Funktionsaufruf: s-Ausdruck

(<Funktor> <Argument$_1$> ... <Argument$_n$>)

• funktionale Applikation

1. eval: Auswertung der Listenelemente
– erstes Element → Lambda-Ausdruck
– Auswertung der restlichen Elemente → aktuelle Parameter
2. apply: Anwendung des Lambda-Ausdrucks auf die Werte

Special Forms Expressions

• Fall 1: Argumente werden nicht ausgewertet

(quote <expression>)

• Fall 2: Argumente werden nur teilweise ausgewertet

(set! <variable> <expression>)
(define <name> <expression>)

nur <expression> wird ausgewertet

• Fall 3: Argumente werden bedingt ausgewertet

(if <condition> <then> <else>)

<condition> wird immer ausgewertet und in Abhängigkeit vom Resultat entweder der <then>- oder der <else>-Zweig

(and <expr1> <expr2> ...)
(or <expr1> <expr2> ...)

Auswertung der Argumente wird abgebrochen, sobald der Funktionswert ermittelt wurde

• Fall 4: Ausdruck wird transformiert

cond wird in geschachtelten if-Ausdruck umgewandelt