10.7 Auswertung von `letrec`

Der Wert von

`(letrec ((`x_1 e_1`)`
`...`
`(`x_n e_n`))`
e`)`

wird durch folgende Schritte bestimmt:

1. Werte $e_1, \ldots, e_n$ zu Werten $v_1, \ldots, v_n$ aus. Dabei dürfen $x_1, \ldots, x_n$ in $e_1, \ldots, e_n$ vorkommen (z.B. in rekursivem Aufruf).

2a. Ergebnis ist der Wert von e, wobei lokal die Bindungen $x_1 = v_1, \ldots, x_n = v_n$ gelten.

2b. Alternativ: Ergebnis ist der Wert von e, wobei x_i jeweils durch `(letrec ((`x_1 v_1`) ...(`x_n v_n`)) `v_i`)` ersetzt wird.

11 Abstrakte Datentypen

- abstrakte Datentypen
- generische Implementierung
  - datengesteuerte Programmierung
  - Operationstabelle

11.1 Abstrakte Datentypen

Bisher: *Konkrete Datentypen*

- Menge von Elementen
- Operationen auf den Elementen (Konstruktoren, Selektoren, Typprädikate)

Jetzt: *Abstrakte Datentypen* (ADT)

- nur Operationen vorgegeben
- Menge von Elementen (Repräsentation) *uninteressant*,
  jede Implementierung der Operationen kann eigene Repräsentation wählen
  abhängig von verfügbaren Implementierungen, nicht-funktionalen
  Anforderungen, etc

11.1.1 Definition: Abstrakter Datentyp (ADT)

Ein *abstrakter Datentyp* A ist gegeben durch

- eine Menge von Operationen auf A
  (durch ihre Verträge)
- eine Menge von Eigenschaften der Operationen
  (ausgedrückt durch Gleichungen).

Bemerkung: Die Operationen können eingeteilt werden in

- Konstruktoren (Konstruktion eines Werts vom Typ A),
- Selektoren (Zugriff auf Komponente eines Werts vom Typ A) und
- Observatoren (Eigenschaften eines Werts vom Typ A, z.B. Länge einer Liste)
- Transformatoren (Umformung eines Werts vom Typ A, z.B. Listenverkettung)

11.1.2 Mengen als ADT: Operationen

Der Datentyp `set(X)` sei gegeben durch die Verträge der Operationen

```
make-empty-set : (X X -> boolean) (X X -> boolean) -> set(X)
set-empty?     : set(X) -> boolean
set-insert     : X set(X) -> set(X)
set-remove     : X set(X) -> set(X)
set-member     : X set(X) -> boolean
```

Die Argumente von `(make-empty-set = <)` sind

- eine Gleichheitsrelation, `= : X X -> boolean`, und
- eine Kleiner-Relation, `< : X X -> boolean`, auf dem Datentyp `X`.

Dadurch kann der Elementdatentyp X parametrisch sein, obwohl auf X eine totale Ordnung definiert sein muss (die von den Operationen verwendet wird).

11.1.3 Mengen als ADT: Gleichungen

Zu den Verträgen der Operationen kommen noch Eigenschaften der Operationen. Z.B. gelten für alle Elemente `x` und Mengen `s` die beiden Gleichungen

```
(set-member x (set-insert x s)) == #t
(set-member x (set-remove x s)) == #f
```

Die Verträgen und die Eigenschaften zusammengenommen definieren den abstrakten Datentyp *Menge*.

Weitere Gleichungen:

```
(set-insert x (set-insert x s)) == (set-insert x s)
(set-remove x (set-remove x s)) == (set-remove x s)
(set-empty? (make-empty-set = <)) == #t
(set-empty? (set-insert x s)) == #f
```

11.1.4 Implementierung eines ADT

Eine Implementierung eines ADT A besteht aus

1. einer Menge (Sorte, konkreter Datentyp) M, deren Elemente die Elemente von A repräsentieren und
2. Implementierungen der Operationen des ADT für M, so dass die Eigenschaften/Gleichungen erfüllt sind.

Bemerkungen

- Ein abstrakter Datentyp A kann mehrere Implementierungen haben.
- Ein Klient eines ADT
  - verwendet nur die ADT Operationen mit den festgelegten Verträgen und Eigenschaften, aber
  - weiß nicht, welche Implementierung verwendet wird.

11.1.5 Implementierungen von `set(X)`

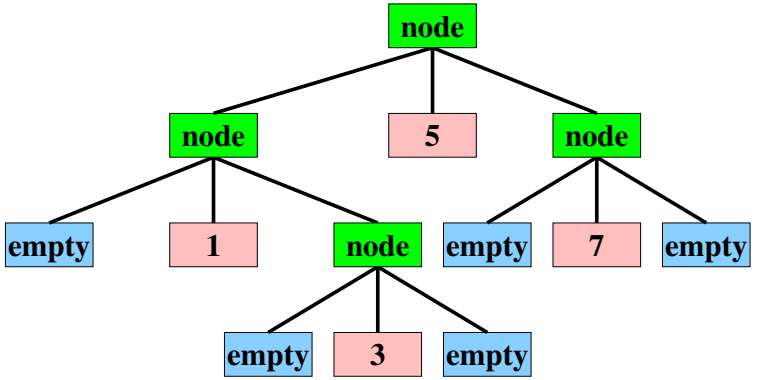
Für Elemente des Datentyps `set(X)` gibt es zum Beispiel folgende Möglichkeiten, demonstriert mit der Repräsentation der Menge $\{1, 3, 5, 7\}$.

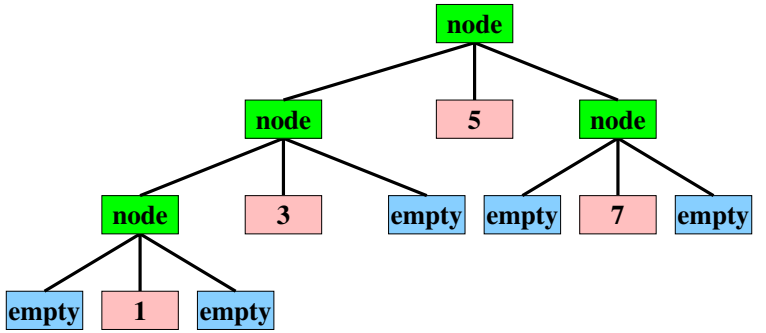
`list-set`: Liste der Elemente
`(list 1 3 5 7)`, `(list 3 5 3 7 3 1)`, `(list 7 7 3 3 1 5)`

`unique-list-set`: Liste der Elemente ohne Duplikate
`(list 1 3 5 7)`, `(list 3 5 7 1)`, `(list 7 3 1 5)`

`sorted-list-set`: Liste der Elemente aufsteigend sortiert, ohne Duplikate
`(list 1 3 5 7)`

`btree-set:` binärer Suchbaum





`function-set:` als charakteristische Funktion

```
(lambda (x)
  (or (= x 1) (= x 3) (= x 5) (= x 7)))
```

```
(lambda (x)
  (and (odd? x) (< 0 x 8)))
```

11.1.6 Implementierung: Menge durch ungeordnete Liste mit Wiederholungen

Ein `list-set(X)` ist ein Wert der Form

`(make-list-set eq? rep)`

wobei `eq? : X X -> boolean` und `rep : list(X)` sind.
Dabei ist = eine Gleichheitsrelation auf `X`.

Leere Menge

```
(define make-empty-list-set
  (lambda (= <)
    (make-list-set = empty)))
```

Element einfügen

```
(define list-set-insert
  (lambda (x s)
    (make-list-set (list-set-eq? s)
                   (make-pair x (list-set-rep s)))))
```

11.1.7 Implementierung: Mengen durch Listen ohne Wiederholung

Ein `unique-list-set(X)` ist ein Wert der Form

```
(make-unique-list-set eq? rep)
```

wobei `eq? : X X -> boolean` und `rep : list(X)` eine Liste ohne wiederholte Elemente ist.
Dabei ist = eine Gleichheitsrelation.

Leere Menge

```
(define make-empty-unique-list-set
  (lambda (= <)
    (make-unique-list-set = empty)))
```

Element suchen, Element entfernen

```
(define unique-list-set-member ...)
```

11.1.8 Implementierung: Mengen durch sortierte Listen ohne Wiederholung

Ein `sorted-list-set(X)` ist ein Wert der Form

```
(make-sorted-list-set eq? lt? rep)
```

wobei `eq? : X X -> boolean` ein Gleichheitsprädikat, `lt? : X X -> boolean` ein Kleiner-Prädikat und `rep : list(X)` eine aufsteigend sortierte Liste ohne wiederholte Elemente ist.

```
(define make-empty-sorted-list-set
  (lambda (= <)
    (make-sorted-list-set = < empty)))

(define sorted-list-set-insert (lambda (x s) ...))
(define sorted-list-set-member (lambda (x s) ...))
```

11.1.9 Implementierung: Mengen durch binäre Suchbäume

Ein `search-tree-set(X)` ist ein Wert der Form

```
(make-search-tree eq? lt? rep)
```

wobei `eq? : X X -> boolean` ein Gleichheitsprädikat, `lt? : X X -> boolean` ein Kleiner-Prädikat und `rep : btree(X)` ein binärer Suchbaum ist.

```
(define make-empty-search-tree-set
  (lambda (= <)
    (make-search-tree = < the-empty-tree)))

(define search-tree-set-insert
  search-tree-insert)
(define search-tree-set-member
  search-tree-member?)
```

11.2 Generische Implementierungen

- Klientenprogramme eines ADT dürfen sich nicht auf eine spezifische Implementierung beziehen, sondern müssen *unabhängig* davon sein.

$\Rightarrow$ Benötigen *generische Schnittstelle*, die direkt die Operationen des abstrakten Datentyps benutzt.

11.2.1 Konstruktion bricht Abstraktion

Problem: Bei Konstruktion wird die Repräsentation erwähnt.

$\Rightarrow$ Bruch der Abstraktion!

Beispiel:

```
; work-with-set : ... -> ...
(define work-with-set
  (lambda (...)
    ... make-empty-search-tree-set ...
    ... set-empty? ... set-insert ...))
```

ADT Fabriken

Lösung: Parametrisiere/abstrahiere über den Konstruktor

⇒ in OO-Sprachen ist das ein **Entwurfsmuster** (*design pattern*),
das **Fabrikmuster** (*factory pattern*)

Beispiel:

```
; work-with-set : ((X X -> boolean) (X X -> boolean) -> S(X)) ... -> ...
;      S(X) muss set(X) implementieren
(define work-with-set
  (lambda (make-empty-set ...)
    ... make-empty-set ...
    ... set-empty? ... set-insert ...))
```

11.2.2 Datengesteuerte Programmierung

- Bei der Verwendung von datengesteuerter Programmierung wählt jede Funktion anhand der Repräsentation die richtige Implementierung aus.
- Implementierung entsprechend dem Muster für gemischte Typen.

Element einfügen

```
(define set-insert
  (lambda (x s)
    (cond
     ((list-set? s)
      (list-set-insert x s))
     ((unique-list-set? s)
      (unique-list-set-insert x s))
     ((sorted-list-set? s)
      (sorted-list-set-insert x s))
     ((search-tree-set? s)
      (search-tree-set-insert x s)))))
```

Element suchen

```
(define set-member
  (lambda (x s)
    (cond
     ((list-set? s)
      (list-set-member x s))
     ((unique-list-set? s)
      (unique-list-set-member x s))
     ((sorted-list-set? s)
      (sorted-list-set-member x s))
     ((search-tree-set? s)
      (search-tree-set-member x s)))))
```

Nachteile der datengesteuerten Programmierung:

- mühsame, uninteressante Implementierung
  boilerplate code
- unflexibel: schlecht erweiterbar

Grund für die Nachteile:

- Jede Operation muss sämtliche Implementierungen kennen.

11.2.3 Operationstabelle

- Vermeidung der Nachteile der datengesteuerten Programmierung
- Ansatz:
  - Jedes Element eines ADT wird mit seinen Operationen zusammengepackt
  - Implementierung des ADT = Operationen $\times$ Repräsentation

  $\Rightarrow$ Kapselung: Implementierung ist versteckt vor dem Programm
- Vgl. objekt-orientierte Programmierung:
  - jedes Objekt kennt seine Methoden
  - Auswahl der Methoden durch Methodenaufruf
  - dort: Methodentabelle (Vtable)

Operationstabelle und Mengenkapselung

```
; Eine Operationstabelle für set(X) ist ein Wert
;  (make-ops ins mem)
; wobei ins : X R(X) -> R(X)  die Einfügeoperation
;   und mem : X R(X) -> boolean der Elementtest ist.
; Name: ops(R,X)
(define-record-procedures ops
  make-ops ops?
  (ops-ins ops-mem))

; Eine Menge mit Elementen aus X ist ein Wert
;  (really-make-set ops rep)
; wobei ops : ops(R,X)
;   und rep : R(X) die Repräsentation der Menge ist
(define-record-procedures set
  really-make-set set?
  (set-ops set-rep))
```

Operationen über der generischen Repräsentation

```
; Generischer Aufruf der Einfügeoperation
; set-insert : X set(X) -> set(X)
(define set-insert
  (lambda (x s)
    (wrap-rep s
      ((ops-ins (set-ops s)) x (set-rep s)))))

; Generischer Aufruf des Elementtests
; set-member : X set(X) -> boolean
(define set-member
  (lambda (x s)
    ((ops-mem (set-ops s)) x (set-rep s))))
```

Einpacken in generische Repräsentation

```
; Ändere (den Zustand der) Repräsentation in einer Menge
; wrap-rep : set(X) R(X) -> set(X)
(define wrap-rep
  (lambda (this rep1)
    (really-make-set (set-ops this) rep1)))
```

Implementierung: Mengen durch Listen

Repräsentiere `set(X)` durch `R(X) = list-set(X)`.

```
; Konstruiere eine generische Menge als list-set(X)
; make-generic-list-set : (X X -> boolean) (X X -> boolean) -> set(X)
(define make-generic-list-set
  (lambda (= <)
    (really-make-set
     (make-ops list-set-insert
               list-set-member)
     (make-empty-list-set = <))))
```

Implementierung: Mengen durch sortierte Listen

Repräsentiere `set(X)` durch `R(X) = ordered-list-set(X)`.

```
; Konstruiere eine generische Menge als ordered-list-set(X)
; make-generic-ordered-list-set : (X X -> boolean) (X X -> boolean) -> set(X
(define make-generic-ordered-list-set
  (lambda (= <)
    (really-make-set
     (make-ops ordered-list-set-insert
               ordered-list-set-member)
     (make-empty-ordered-list-set = <))))
```

Beispiel: Anwendung der neuen Mengenoperationen

```
   (define ul-s (set-insert (make-generic-list-set = <) 1))
   (set-rep (set-insert ul-s 1)
=> (list 1 1)
   (set-member ul-s 7)
=> #f

   (define ol-s (set-insert (make-generic-ordered-list-set = <) 1))
   (set-rep (set-insert ol-s 1))
=> (list 1)
   (set-member ol-s 1)
=> #t
```

Interne Operationen

- Erweitere den ADT `set(X)` um
  `set-union : set(X) set(X) -> set(X)`
- Problem: Unterschiedliche Repräsentation der Mengen!
- Lösungsmöglichkeit: Erweitere ADT um Konversionsoperation `set->list`.

Signatur: `set->list : set(X) -> list(X)`

Erklärung: `(set->list s)` ist eine Liste, die die Elemente von s ohne Wiederholung enthält.

Definition:

```
(define set->list
  ...)
; DAMIT
(define set-union
  (lambda (s1 s2)
    (list-fold set-insert s1 (set->list s2))))
```

11.3 Zusammenfassung

- Abstrakte Datentypen spezifizieren
  - eine Menge von Operationen
  - Eigenschaften der Operationen (Gleichungen)
- ADTs lassen mehrere Implementierungen zu
- Klienten sind unabhängig von Implementierung
- Fabrikmuster
- Datengetriebene Implementierung möglich
- Generische Implementierung durch Paarung von
  Operationstabelle $\times$ Repräsentation
  (vgl. Objekt-Orientierung)