19 Vektoren

Betrachte

```
; Liste der Nachfolger eines Knotens
(: succ (graph node -> (list node)))
(define succ
  (lambda (g n)
    (if (empty? g)
        (violation "illegal node")
        (let ((nodelist (first g)))
          (if (equal? (first nodelist) n)
              (rest nodelist)
              (succ (rest g) n))))))
```

- `find-path` ruft `succ` sehr oft auf.
- Jeder Aufruf durchläuft die Liste g:
  maximale Anzahl der Schritte = Anzahl der Knoten in g

Effizientere Datenstruktur dafür: Vektor

- auch Reihung, Feld, Array
- zusammengesetzte Datenstruktur, die einer Zahl (*Index*) in $[0, l)$ in konstanter Zeit einen Wert zuordnet
- l ist die *Länge* des Vektors, d.h. die Anzahl der Werte, die in ihm abgelegt sind

Teachpack `vector.ss` (siehe Folien)

Operationen auf Vektoren

1. Konstruktor (`vector` $V_0 \ \ldots \ V_{n-1}$) $n \geq 0$
   erzeugt Vektor der Länge n, der jeweils $0 \leq i < n$ den Wert V_i zuordnet
2. Konstruktor (`make-vector` $n \ V$) $n \geq 0$
   erzeugt Vektor der Länge n; alle Elemente mit V initialisiert
3. (`vector-ref (vector` $V_0 \ \ldots \ V_{n-1}$) i) $= V_i$
   falls $0 \leq i < n$; in konstanter Zeit
4. (`vector-length (vector` $V_0 \ \ldots \ V_{n-1}$)) $= n$
5. Typprädikat `vector?`
6. Vektoren als Zustandsvariable:
   (`vector-set! (vector` $V_0 \ \ldots \ V_{n-1}$) $i \ V$)
   ersetzt V_i im Vektor durch V, falls $0 \leq i < n$

Alternative Repräsentation von Graphen

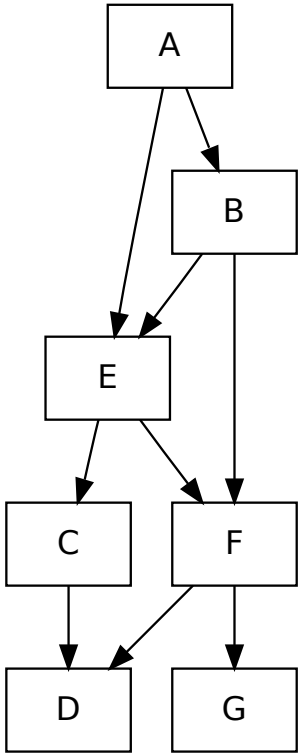
- Um Vektoren zu verwenden müssen die Graphknoten als Zahlen kodiert werden:
- Ein Knoten ist eine natürliche Zahl kleiner als N (Anzahl der Knoten des Graphen).
- Ein Graph ist ein Vektor, der Listen von Knoten enthält.

Mit $A \mapsto 0$, $B \mapsto 1$, ... ergibt sich

```
(define graph-as-list
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

```
(define graph-as-vector
  '#((1 4)
     (4 5)
     (3)
     ()
     (2 5)
     (3 6)
     ()))
```

Visuelle Darstellung des Graphen



Nachfolger eines Graphknotens

```
; Liste der Nachfolger eines Knotens
(: vsucc (vgraph vnode -> (list vnode)))
(define succ
  (lambda (g n)
    (vector-ref g n)))
```

Typische Operationen auf Vektoren

- Alle Berechnungen sind indexbasiert, d.h. nicht der Datentyp selbst, sondern seine Indexmenge $\mathbf{N}$ bestimmt das Prozedurschema
- Beispiel: Summe der Elemente eines 3D Vektors

Summe der Elemente eines beliebig langen Vektors

```
(: vector-sum ((predicate vector?) -> number))
; summiere alle Einträge des Vektors v
(define vector-sum
  (lambda (v)
    ...))

(check-expect (vector-sum (vector -1 3/4 1/4)) 0)
(check-expect (vector-sum (vector .1 .1 .1 .1 .1 .1 .1 .1 .1 .1)) 10)
(check-expect (vector-sum (vector)) 0)
(check-expect (vector-sum (make-vector 42 0.5)) 21)
```

Hilfsfunktion

```
(: vector-sum ((predicate vector?) -> number))
; summiere alle Einträge des Vektors v
(define vector-sum
  (lambda (v)
    (vector-sum-helper v (vector-length v))))

(: vector-sum-helper ((predicate vector?) natural -> number))
; summiere die Einträge des Vektors v mit Index kleiner i
(define vector-sum-helper
  (lambda (v i)
    ...))
```

Umgedreht summieren

```
(: lr-vector-sum-helper ((predicate vector?) natural -> number))
; summiere die Einträge des Vektors v mit Index größer gleich i
(define lr-vector-sum-helper
  (lambda (v i)
    ...))
```

Summer zweier Vektoren

```
(: vector-add ((predicate vector?) (predicate vector?) -> (predicate vector?
; berechne die Vektorsumme von zwei Vektoren
(define vector-add
  (lambda (v w)
    ...))
```