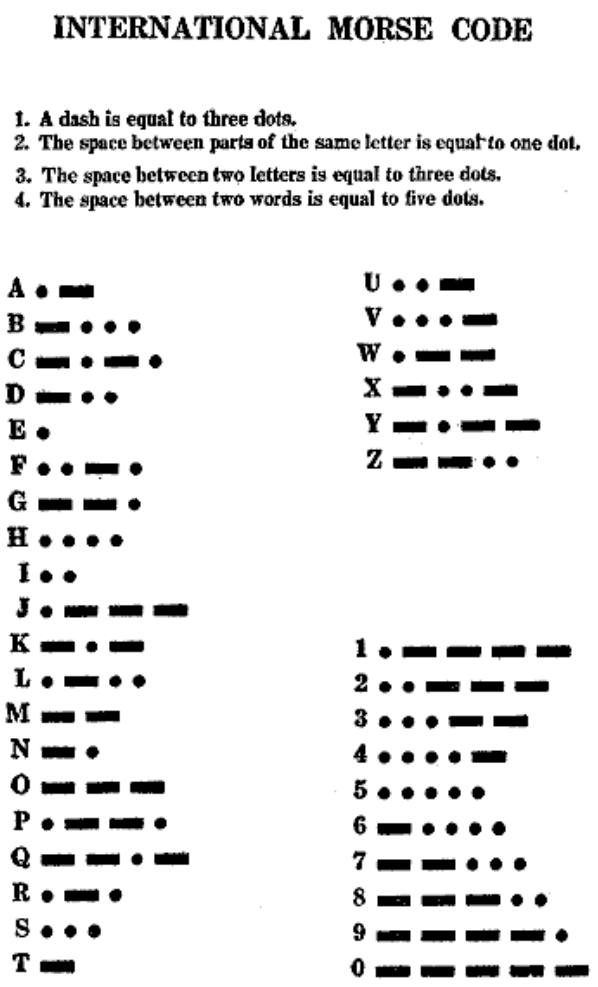
7.3 Huffman-Bäume

- Anwendung von Binärbäumen
- Aufgabe: Platz-effiziente Kompression von Textdaten
- Standardcodierungen von Textdaten
  - ISO-8859-1: 8 Bit pro Zeichen
  - UTF-16: 16 Bit pro Zeichen

  Codierungen mit *fester Länge*: Zeichen $\mapsto$ Bitfolge fester Länge
- Beobachtung: In Texten kommen Zeichen mit unterschiedlichen Häufigkeiten vor, trotzdem verbrauchen alle gleich viel Platz.
- Ziel: Codierung mit *variabler Länge*, so dass häufige Zeichen wenig Platz benötigen.

7.3.1 Beispiel: Morse-Code



INTERNATIONAL MORSE CODE

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to five dots.

A • —
B — • • •
C — • — •
D — • •
E •
F • • — •
G — — •
H • • • •
I • •
J • — — —
K — • —
L • — • •
M — —
N — •
O — — —
P • — — •
Q — — • —
R • — •
S • • •
T —

U • • —
V • • • —
W • — —
X — • • —
Y — • — —
Z — — • •

1 • — — — —
2 • • — — —
3 • • • — —
4 • • • • —
5 • • • • •
6 — • • • •
7 — — • • •
8 — — — • •
9 — — — — •
0 — — — — —

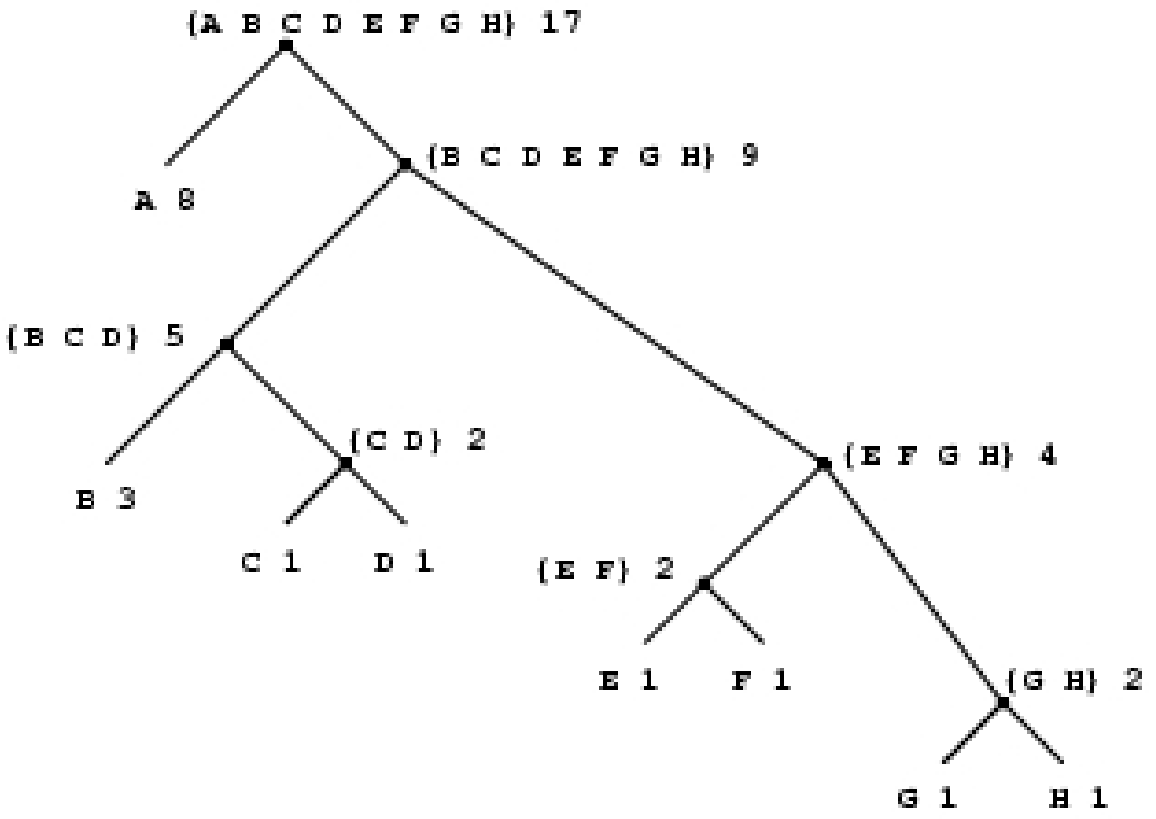
7.3.2 Präfix-Codierung

- Der Code eines Zeichens ist niemals Präfix des Codes eines anderen Zeichens.
- Bsp: Wenn 'E' mit '0' codiert wird, darf keine andere Codierung mit '0' beginnen.
- Problemstellung:
  Gegeben ein Zeichenvorrat mit Häufigkeiten für jedes Zeichen.
  Konstruiere eine Präfix-Codierung, die im Mittel die kürzeste Codierung liefert.

7.3.3 Huffman-Codierung

- Repräsentiert durch einen Binärbaum (den *Huffman-Baum*)
- Jedes Blatt repräsentiert ein Zeichen (mit Häufigkeit)
- Jeder innere Knoten repräsentiert eine Menge von Zeichen (mit Gesamthäufigkeit)
- Der Wurzelknoten repräsentiert den gesamten Zeichenvorrat
- Codierung eines Zeichens durch den Weg von der Wurzel zum Zeichen:
  - Suche das Blatt mit dem Zeichen beginnend von der Wurzel
  - Schreibe dabei '0'/'1' für jede Auswahl eines linken/rechten Teilbaums
- Decodierung eines Codes:
  - Beginne bei der Wurzel
  - Gehe zum linken/rechten Teilbaum über, je nachdem ob nächstes Bit '0'/'1' ist
  - Bei Ankunft am Blatt: Zeichen identifiziert

7.3.4 Beispiel: Huffman-Baum



Aus: Abelson and Sussman, Structure and Interpretation of Computer Programs, MIT Press

7.3.5 Implementierung von Huffman-Bäumen — Blätter

```
; Ein Huffman-Blatt ist ein Wert
;  (make-huffman-leaf s w)
; wobei s : string
; und   w : real (das Gewicht bzw die Häufigkeit von s)
(define-record-procedures huffman-leaf
  make-huffman-leaf huffman-leaf?
  (huffman-leaf-name huffman-leaf-weight))
```

7.3.6 Implementierung von Huffman-Bäumen — Knoten

```
; Ein Huffman-Knoten ist ein Wert
;  (make-huffman-node sl w l r)
; wobei sl : list(string)
;        w : real (kumulative Häufigkeit von sl)
;        l, r : huffman-tree
(define-record-procedures huffman-node
  really-make-huffman-node huffman-node
  (huffman-node-names huffman-node-weight
   huffman-node-left huffman-node-right))

; Ein Huffman-Baum ist einer der folgenden
; - ein Huffman-Blatt
; - ein Huffman-Knoten
(define-contract huffman-tree (mixed huffman-leaf huffman-node))
```

7.3.7 Erzeugung eines Huffman-Knotens

Die Information an einem inneren Knoten eines Huffman-Baums ergibt sich direkt aus seinen Teilbäumen, daher

```
; Huffman-Knoten aus zwei Huffman-Bäumen konstruieren
(: make-huffman-node (huffman-tree huffman-tree -> huffman-node))
(define make-huffman-node
  (lambda (l r)
    (really-make-huffman-node
     (append (huffman-tree-names l)
             (huffman-tree-names r))
     (+ (huffman-tree-weight l)
        (huffman-tree-weight r))
     l r)))
```

Nach Topdown-Design verbleibt zu definieren

```
(: huffman-tree-names (huffman-tree -> (list string)))
(: huffman-tree-weight (huffman-tree -> real))
```

7.3.8 Liste der Namen in einem Huffman-Baum

```
(: huffman-tree-names (huffman-tree -> (list string)))
(define huffman-tree-names
  (lambda (ht)
    (cond
     ((huffman-leaf? ht)
      (list (huffman-leaf-name ht)))
     ((huffman-node? ht)
      (huffman-node-names ht)))))
```

7.3.9 Gewicht eines Huffman-Baums

```
(: huffman-tree-weight (huffman-tree -> real))
(define huffman-tree-weight
  (lambda (ht)
    (cond
     ((huffman-leaf? ht)
      (huffman-leaf-weight ht))
     ((huffman-node? ht)
      (huffman-node-weight ht)))))
```

7.3.10 Decodierung eines Huffman-Codes

```
; Ein Bit ist entweder 0 oder 1
(define-contract bit (oneof 0 1))

; Huffman-codierte Bitfolge decodieren
(: huffman-decode ((list bit) huffman-tree -> (list string)))
(define huffman-decode
  (lambda (bits ht)
    ...))
```

- Neuigkeit in dieser Prozedur
  - `bits` ist Liste
  - `ht` ist Huffman-Baum
- Beide müssen zerlegt werden!
- Beim **Decodieren** obliegt die Steuerung den `bits`, daher verwende
  - das Muster für gemischte Datentypen für `ht` und
  - das Muster für Listenfunktionen für `bits`

7.3.11 Decodierung eines Huffman-Codes, II

Zunächst nur das erste Zeichen decodieren

```
; Erstes Zeichen einer Huffman-codierten Bitfolge decodieren
(: decode-1 ((list bit) huffman-tree -> (list string)))
(define decode-1
  (lambda (bits ht)
    (cond
     ((huffman-leaf? ht)
      (list (huffman-leaf-name ht)))
     ((huffman-node? ht)
      (cond
       ((empty? bits)
        empty)
       ((pair? bits)
        (if (= (first bits) 0)
            (decode-1 (rest bits) (huffman-node-left ht))
            (decode-1 (rest bits) (huffman-node-right ht))))))))))
```

```
; Huffman-codierte Bitfolge decodieren
(: huffman-decode ((list bit) huffman-tree -> (list string)))
(define huffman-decode
  (lambda (bits ht-root)
    (letrec ((decode-1
              (lambda (bits ht)
                (cond
                 ((huffman-leaf? ht)
                  (make-pair (huffman-leaf-name ht)
                             (decode-1 bits ht-root)))
                 ((huffman-node? ht)
                  (cond
                   ((empty? bits)
                    empty)
                   ((pair? bits)
                    (if (= (first bits) 0)
                        (decode-1 (rest bits) (huffman-node-left ht))
                        (decode-1 (rest bits) (huffman-node-right ht))))))))))
      (decode-1 bits ht-root)))
```

7.3.12 Huffman-Codieren

```
(: huffman-encode ((list string) huffman-tree -> (list bit)))
(define huffman-encode
  (lambda (names ht)
    (cond
     ((empty? names)
      empty)
     ((pair? names)
      (append (huffman-encode-name (first names) ht)
              (huffman-encode (rest names) ht))))))
```

Verbleibt zu definieren

```
(: huffman-encode-name (string huffman-tree -> (list bit)))
```

7.3.13 Huffman-Codieren eines Zeichens

```
(: huffman-encode-name (string huffman-tree -> (list bit)))
(define huffman-encode-name
  (lambda (name ht)
    (cond
     ((huffman-leaf? ht)
      empty)
     ((huffman-node? ht)
      (let ((left (huffman-node-left ht))
            (right (huffman-node-right ht)))
        (if (member name (huffman-tree-names left))
            (make-pair 0 (huffman-encode-name name left))
            (make-pair 1 (huffman-encode-name name right)))))))))
```

- Selbst: Definiere endrekursive Version der Codierungsfunktion!

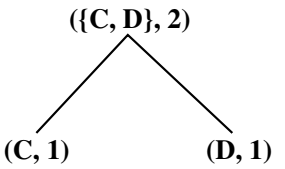
7.3.14 Aufbau eines Huffman-Baums

- Der Aufbau eines Huffman-Baums geschieht *bottom-up*, das heißt von den Blättern zur Wurzel.
- Vorbereitung: jeder Name wird mit seiner Häufigkeit in ein Huffman-Blatt verpackt.
- Eingabe ist die (nicht-leere) Liste dieser Blätter.
- Solange noch mindestens zwei Bäume in der Liste sind
  - Entferne die beiden Bäume mit niedrigstem Gewicht
  - Füge sie mit `make-huffman-node` zusammen
  - Lege den neuen Baum zurück in die Liste
- Der gesuchte Huffman-Baum ist das erste (einzige) Element der Liste.

7.3.15 Beispiel für den Aufbau

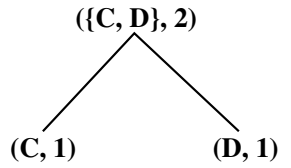
Beginne mit $(B, 3)$, $(C, 1)$, $(D, 1)$.

Neuer Baum aus $(C, 1)$ und $(D, 1)$:

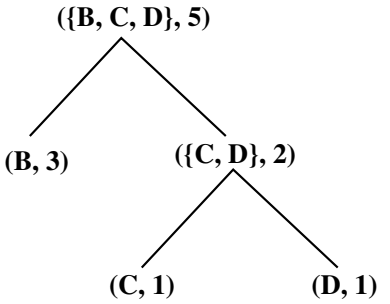


Weiter mit:

$(B, 3)$,



Neuer Baum daraus (Endergebnis):



Zusammenfassung

- Ein Huffman-Baum ist ein Binärbaum, der einen Präfixcode mit variabler Länge durch den Weg von der Wurzel zu einem Zeichen am Blatt repräsentiert.
- Die Zeichen sind so an den Blättern des Baums verteilt, dass häufige Zeichen kurze Codeworte besitzen.
- Diese Codierung ist optimal, d.h., sie liefert im Mittel die kürzeste Codierung einer Zeichenfolge.

7.4 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), *aber nicht verwendet werden*.

2. Ergebnis ist der Wert von e, wobei

   - lokal die Bindungen $x_1 = v_1, \ldots, x_n = v_n$ gelten
     oder alternativ
   - jedes Vorkommen von x_i durch
     `(letrec ((`x_1 v_1`) ...(`x_n v_n`)) `v_i`)` ersetzt wird.