Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Teil 7: Datentypen

9.6.2021

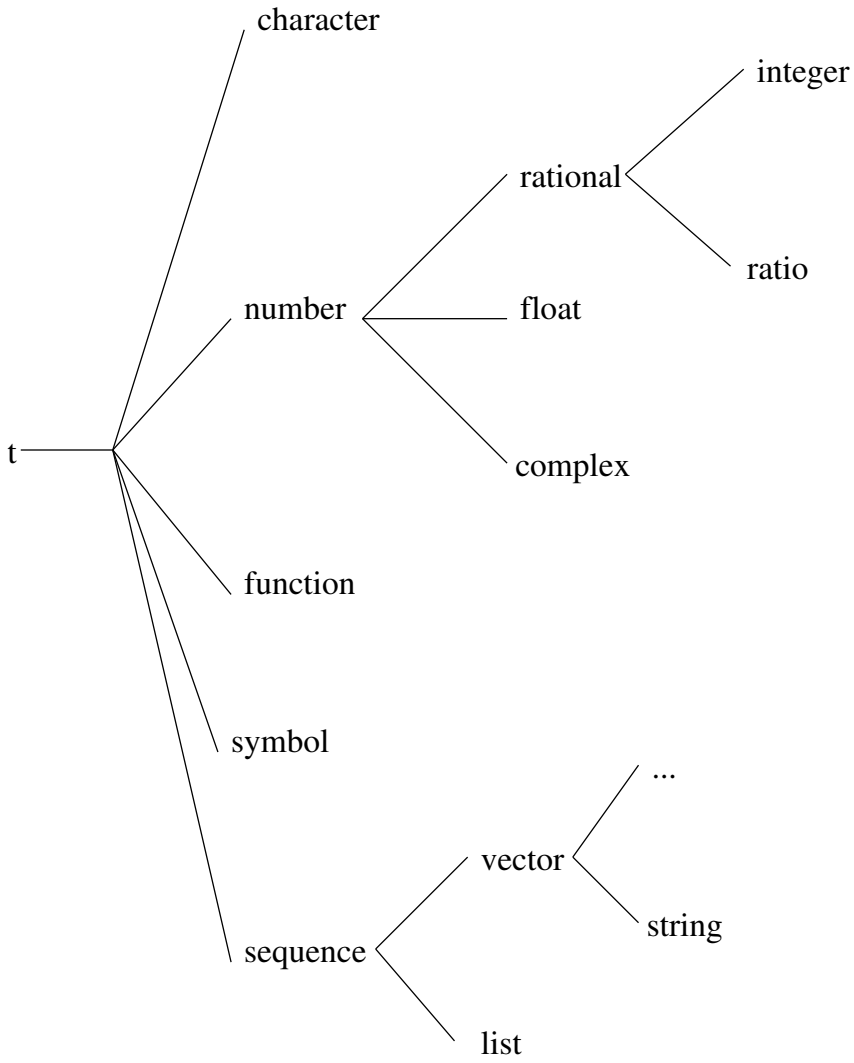
LISP-Intensivkurs

Übersicht:

- Datentypen
- Projekt: Tic-Tac-Toe Spiel

Datentypen

Datentypen in Lisp sind bezüglich der Mengeninklusion hierarchisch angeordnet; folgendes ist ein Ausschnitt der COMMONLISP Typhierarchie:



Datentypen

Für den Umgang mit Typen stellt Lisp Prädikate und Funktionen zur Verfügung:

`(typep obj typ)` überprüft, ob `obj` ein Element in `typ` ist.

```
(typep #\a 'character)  ==> T
```

Typspezifische Prädikate, z.B:

```
(listp '(1 2 3)) ==> T
(numberp 23)     ==> T
(symbolp 'a)     ==> T
(ratiop   2/3)   ==> T
```

`(type-of obj)` liefert den spezifischsten Typ, in dem `obj` enthalten ist, z.B:

```
(type-of #c(5 1))  ==> COMPLEX
(type-of 'nil)     ==> NULL
(type-of '(1 2 3)) ==> CONS
```

Datentypen

Bemerkung: Lisp erlaubt es also, i*n der Sprache selbst* Aussagen über Typen zu machen, und unterscheidet sich darin wesentlich von den meisten anderen Sprachen.

Im folgenden stellen wir eine Reihe von in COMMONLISP eingebauten Datentypen vor.

Statische vs. dynamische Typisierung

Viele Hochsprachen sind *statisch* typisiert, d.h. Typkorrektheit wird zur Compilezeit (z.B. Java, C, Pascal, Eiffel) oder Definitionszeit (z.B. ML) überprüft.

Demgegenüber ist Lisp *dynamisch* typisiert, d.h. Typinformation wird zur Laufzeit überprüft.

Statische vs. dynamische Typisierung

Beispiel: Die Funktion `test` kann nur auf einer Liste von Zahlen arbeiten:

```
> (defun test (lst)
     (mapcar #'(lambda (x) (+ x 3)) lst))
TEST
> (test '(1 2 3))
(4 5 6)
```

Daher ist die folgende Definition sicher falsch:

```
> (defun testcall (x)
     (cons x (test '((1) (2)))))
TESTCALL
```

Dies wird jedoch erst zur Laufzeit entdeckt:

```
> (testcall 4)
>>Error: The value of RESULT, (1), should be a NUMBER
```

Statische vs. dynamische Typisierung

In ML wird bereits die Definition einer solchen Funktion verhindert:

```
- fun test lst = map (fn x => x + 3) lst ;
val test = fn : int list -> int list
- test [1, 2, 3] ;
val it = [4,5,6] : int list
- fun testcall x = x :: test [[1], [2]] ;
Error: operator and operand don't agree
  operator domain: int list
  operand:         'Z list list
  in expression:
    test ((1 : 'Z) :: nil) :: ((2 : 'Z) :: nil) :: nil
```

Statische vs. dynamische Typisierung

Ebenso ist in Lisp die Konstruktion bestimmter Werte erlaubt:

```
> '((1) (2 3) (4))
((1) (2 3) (4))
> '(1 (2))
(1 (2))
```

… die in anderen Sprachen (z.B. ML) zu Fehlern führen würde:

```
- [[1], [2,3], [4]];
val it = [[1],[2,3],[4]] : int list list
- [1, [2]];
Error: expected numeric type
  found type: int list
```

Zahlen

Der Datentyp `number` besitzt mehrere Untertypen.

`rational` bezeichnet die rationalen Zahlen in Quotientendarstellung. Als Selektoren stehen numerator und denominator zur Verfügung.

Beispiele: 1, 5/7, -3/8.

`integer` ist ein Untertyp von `rational` und bezeichnet die ganzen Zahlen (ohne Größenbeschränkung). integer selber hat die Untertypen `fixnum` und `bignum`.

Neben der gewöhnlichen Dezimalschreibweise, können Objekte vom Typ rational in jeder beliebigen Basis zwischen 2 und 36 dargestellt werden. Dabei wird der eigentlichen Zahlendarstellung die Basis in der Form *#basis*R vorangestellt; übliche Abkürzungen: `#B ::= #2R`, `#O ::= #8R` und `#X ::= #16R`. Beispiel: `#X12 = #16R12 = 18` (Dezimal).

Zahlen

`float` bezeichnet Gleitpunktzahlen und besitzt die Untertypen `short-float`, `single-float`, `double-float` und `long-float`.

Beispiele: `2.3, -.123E-2`

Statt ‘`E`’ können auch ‘`S`’, ‘`F`’, ‘`D`’ und ‘`L`’ verwendet werden, die jeweils die Untertypen spezifizieren.

Zahlen

`complex` bezeichnet die komplexen Zahlen mit Real- und Imaginäranteil. Untertypen entstehen durch Typfestlegung für die Komponenten.

Als Konstruktor für komplexe Zahlen dient die Funktion `complex`. Die Komponenten können mit den Funktionen `realpart` und `imagpart` selektiert werden.

Beispiel: `(complex 3 4) ==> #C(3 4)`

Zahlen

Prädikate und Funktionen auf Zahlen

- Typerkenner `numberp`,`integerp`,`floatp`,`complexp`
- Relationen auf Zahlen =, `/=`, <, <=, >, >=
- weitere Prädikate: `zerop`,`plusp`,`minusp`,`oddp`,`evenp`
- Arithmetische Operatoren +, -, *, /
- Exponential-, logarithmische und trigonometrische Funktionen
- Typkonversionen etc.: `floor`, `ceiling`, `truncate`, `round`, `mod`, `div`
- Bitoperationen auf Zahlen.

Zahlen

Beispiele:

```
(< 1 1.000000000000001 2)                    ==> T
(< 1 1.0000000000000001 2)                   ==> NIL
(= 0 #C(0.0 0.0))                            ==> T
(typep #C(2.5 1.5) '(complex float))         ==> T
(numerator 2/6)                              ==> 1
(denominator 2/6)                            ==> 3
(realpart #C(1 2))                           ==> 1
(imagpart #C(1 2))                           ==> 2
(- 2/3 1/6)                                  ==> 1/2
(expt 2 170)
   ==> 1496577676626844588240573268701473812127674924007424
```

Zeichen

Zum Typ `character` gehören vornehmlich die druckbaren Zeichen; z.B. Buchstaben und Ziffern. Objekte vom Typ `character`:

```
#\a          ==> #\a
#\space      ==> #\Space
#\newline    ==> #\Newline
```

Prädikate und Funktionen auf `character`:

- Typerkenner `characterp`
- Relationen auf Zeichen `char=`, `char/=`, `char<`,...
- Weitere Prädikate: `alpha-char-p`, `digit-char-p`, `lower-case-p`, `upper-case-p`
- Konstruktoren und Selektoren: `code-char`, `char-code`
- Funktionen zur Transformation von Zeichen: `char-upcase`, `char-downcase`

Zeichen

Beispiel:

```
(char= #\a #\a)     ==> T
(alpha-char-p #\a) ==> T
(lower-case-p #\a) ==> T
(char-code #\a)     ==> 97
(code-char 97)      ==> 97
```

Strings

Strings sind Sequenzen von Zeichen, die in " … " eingeschlossen werden. Formatierte Strings lassen sich unter anderem mit format erzeugen.

```
> "This is a string"
"This is a string"
> (format nil "a ~A created by format" "string")
"a string created by format"
```

Strings

Strings sind, ebenso wie Listen, spezielle **Sequenzen**. Einige der bereits für Listen vorgestellten Funktionen arbeiten ganz generell auf Sequenzen.

```
> (length '(a b c d e))
5
> (length "abcde")
5
> (reverse '(a b c d e))
(E D C B A)
> (reverse "abcde")
"edcba"
> (elt '(a b c d e) 2)
C
> (elt "abcde" 2)
#\c
```

Dies sind erste Vorteile des Subtyp-Konzepts, das sich in erweiterter Form in der objekt-orientierten Programmierung wiederfindet.

Symbole

Man kann Symbole als Bezeichner für Variablen **und** Funktionen ansehen; tatsächlich sind sie *strukturierte Datenobjekte*, die zum Datentyp `symbol` gehören:

```
> (inspect 'cons)
#<Symbol 65D607>

[0: NAME] "CONS"
[1: VALUE] Unbound
[2: FUNCTION] #<Compiled-Function CONS 522A3E>
[3: PLIST] NIL
[4: PACKAGE] #<Package "LISP" 66D06E>
```

Symbole

Symbole enthalten folgende Komponenten:

Print-Name	dient der externen Repräsentation eines Symbols.
Funktionsdefinition	enthält einen Zeiger auf den Speicherbereich, in dem eine dem Symbol
Wert	enthält einen Zeiger auf den Speicherbereich, der den aktuellen Wert
Property-Liste	enthält eine Liste von Eigenschafts-Werte-Paaren.
Package	enthält einen Zeiger auf das Package zu dem das Symbol gehört.

Symbole

Bemerkung: P-Listen werden nicht mehr allzu häufig verwendet; stattdessen nimmt man oft gewöhnliche Assoziationslisten oder Hash-Tabellen (s.h.). Prädikate und Funktionen auf symbol:

- Konstruktor: `intern`
- Typerkenner: `symbolp`
- Tests: `boundp`, `fboundp`, `functionp`
- Selektoren `symbol-name`, `symbol-value`, `symbol-function`, `symbol-package`

Symbole

Beispiel:

```
(intern "y")              ==> |y|
(setf x '(1 2 3))
(boundp 'x)               ==> T
(fboundp 'x)              ==> NIL
(symbol-name 'x)          ==> "X"
(symbol-value 'x)         ==> (1 2 3)
(symbol-function 'x)      ==> >>Error: ...
(defun x (y) y)
(symbol-value 'x)         ==> (1 2 3)
(symbol-function 'x)
   ==> #<Interpreted-Function
            (NAMED-LAMBDA X (Y) (BLOCK X Y)) 1221AAE>

(x x)                     ==> (1 2 3)
```

Packages

Packages in Lisp sind vergleichbar mit Modulen in anderen Sprachen. Sie dienen dazu,

- verschiedene Namensräume zu schaffen
- zu regeln, welche Namen wo sichtbar sind

Das aktuelle Package regelt, wie Namen gelesen werden:

```
USER(82): *package*
#<The COMMON-LISP-USER package>
USER(83): (describe *package*)
#<The #1=COMMON-LISP-USER package> is
       the package named #1#.
It has nicknames: USER, CL-USER.
It uses packages: COMPOSER, COMMON-LISP, EXCL, ...
```

Packages

- Das aktuelle Package ist gebunden an die Variable `*package*`
- Außer durch seinen offiziellen Namen kann es auch durch “Nicknames” benannt werden (z.B. mit `"USER"` oder `:user`)
- Ein Package kann andere Packages einschließen (benutzen)

Packages

Jedem Lisp-Symbol ist ein Package zugeordnet. In diesem Package kann der Name des Symbols unqualifiziert verwendet werden.

```
USER(84): (setq foobar 5)
5
USER(85): (describe 'foobar)
FOOBAR is a SYMBOL.
  Its value is 5
  It is INTERNAL in the COMMON-LISP-USER package.
USER(86): foobar
5
```

Packages

- Der Benutzer kann selbst Packages definieren

  mit `make-package`

- Man kann in einander es Package wechseln

  mit `in-package`

```
USER(87): (make-package "MY-PACKAGE"
              :nicknames '(my-one))
#<The MY-PACKAGE package>
USER(88): *package*
#<The COMMON-LISP-USER package>
USER(89): (in-package my-one)
#<The MY-PACKAGE package>
MY-ONE(90):
```

Packages

Der Name eines Symbols ist in anderen als dem Heimat-Package nicht sichtbar:

```
MY-ONE(98): foobar
Error: Attempt to take the value of
       the unbound variable ‘FOOBAR’.
  [condition type: UNBOUND-VARIABLE]

Restart actions (select using :continue):
 0: Try evaluating FOOBAR again.
 1: Use the value of USER::FOOBAR instead.
```

Durch Qualifizierung ist das Symbol dennoch erreichbar:

```
MY-ONE(100): user::foobar
5
```

Packages

Weiteres zu Packages:

- Deklaration von Symbolen eines Package als“extern”
- Import von externen Symbolen eines Package in ein anderes. Dadurch Verwendung ohne Qualifizierung möglich
- Schreibweise: Das externe Symbol ext aus Package pack kann als pack:ext (ein Doppelpunkt!) referenziert werden
- Wichtig: Eine Deklaration der Art `(in-package "USER")` sollte am Anfang jeder Lisp-Datei stehen!

Arrays

Arrays sind sinnvoll, wenn eine große Anzahl gleicher bzw. gleich zu behandelnder Daten verwaltet werden soll. Funktionen zur Konstruktion und für den Zugriff auf Arrays:

- `make-array` erzeugt ein Array mit Dimensionen dim. Anfangswerte können als Schlüsselwortargumente angegeben werden `(:initial-element :initial-contents)`. Dabei kann man auch spezifizieren, ob die Größe eines Arrays dynamisch veränderbar sein soll `(:adjustable, adjust-array)`. Vektoren können mit `vector` erzeugt werden.
- `(aref arr idx-1...idx-n)` greift in array auf die Komponente zu den gegebenen Indexwerten zu; Array-Indizes beginnen dabei immer mit 0.
- Wertzuweisungen an Arraykomponenten erfolgen mit dem Zuweisungsmakro `setf`.

Arrays

Beispiel:

```
(setf m (make-array '(3 4) :initial-element 0))
(aref m 1 2)            ==> 0
(setf (aref m 1 2) 5) ==> 5
(aref m 1 2)            ==> 5
m
          ==> #<Simple-Array T (3 4) 123385E>
(setf *print-array* t)
m
          ==> #2A((0 0 0 0) (0 0 5 0) (0 0 0 0))
(array-dimension m 0) ==> 3
(array-dimension m 1) ==> 4
```

Arrays

Bemerkung: Falls man ein eindimensionales Array mit bestimmtem Inhalt kreieren will, so benutzt man den Untertyp `vector` von `array`:

```
(setf *shapes* (vector :rectangle :circle))
(aref *shapes* 1)    ==> :circle
(svref *shapes* 1)   ==> :circle
```

Der auf einfache Vektoren zugeschnittene Selektor `svref` ist u.U. effizienter als `aref`.

Arrays

Bemerkung:

- Strings sind Vektoren von Zeichen; z.B:

```
(aref "hallo" 1)    ==> #\a
```

- Viele Funktionen, die wir bisher nur als Listenoperationen kennengelernt haben, funktionieren eigentlich auf dem Typ sequence, der sich aus Listen und Vektoren zusammensetzt:

```
(length "hallo")      ==> 5
(reverse "hallo")     ==> "ollah"
(elt "hallo" 1)       ==> #\a
(elt '(h a l l o) 1) ==> A
```

Hashtabelle

Zweck: effizientes Speichern und effizienter Zugriff auf *große* Datenmengen. Hashtabellen werden erzeugt mit `make-hash-table` und modifiziert mit einem `setf` auf `gethash`.:

Hashtabelle

Beispiel:

```
(setf *h-table*
      (make-hash-table :size 5000 :test #'eql))

(setf (gethash 'BW *h-table*) 'Baden-Wuerttemberg)
(setf (gethash 'BY *h-table*) 'Bayern)
(setf (gethash 'SL *h-table*) 'Saarland)

(gethash 'BY *h-table*)     ==> BAYERN
(gethash 'NRW *h-table*)    ==> NIL

(remhash 'BY *h-table*)     ==> T
(gethash 'BY *h-table*)     ==> NIL

(clrhash *h-table*)         ==> ...
```

Hashtabelle

Bemerkung: Als Hash-Funktion wird `sxhash` verwendet, das mit Objekten beliebigen Typs aufgerufen werden kann:

```
(sxhash "hallo")  ==> 536731615
(sxhash 2345)     ==> 9
(sxhash '(1 2 3)) ==> 392540170
```

Strukturen

Mit dem Makro `defstruct` werden neue Strukturen (Records) angelegt; z.B:

```
(defstruct cons-cell
   car
   cdr)
```

Bemerkung: Genaugenommen ist `defstruct` ein *Typkonstruktor*, mit dem Record-Typen generiert werden können.

Strukturen

`defstruct` definiert außerdem einige neue Funktionen:

- Eine Konstruktorfunktion; hier: `make-cons-cell`.
- Selektorfunktionen; hier: `cons-cell-car`, `cons-cell-cdr`.
- Typerkenner; hier: `cons-cell-p`
- Eine Kopierfunktion; hier: `copy-cons-cell`

Strukturen

Beispiel:

```
(setf c (make-cons-cell :car 1 :cdr 2))
   ==> #S(CONS-CELL CAR 1 CDR 2)

(cons-cell-p c)   ==> T
(cons-cell-car c) ==> 1
(cons-cell-cdr c) ==> 2
```

Zur Erinnerung: Der *Inspektor* eignet sich besonders zum Betrachten von Strukturen.

Projekt: Tic-Tac-Toe-Spiel

Spielbeschreibung: Zwei abwechselnd ziehende Spieler versuchen, auf einem quadratischen Spielfeld eine Reihe, Spalte oder Diagonale jeweils mit den eigenen Steinen zu füllen.

Typischer Spielzustand:

```
    0   1   2
  -------------
0 | A |   | A |
  -------------
1 |   |   |   |
  -------------
2 |   |   | B |
  -------------
```

Projekt: Tic-Tac-Toe-Spiel

Sittlicher Nährwert der Untersuchung dieses Spiels:

Verwendung einiger KI-Techniken, z.B.:

- Aufbau eines Spielbaums bzw. Spielgraphen
- Anwendung einer einfachen Form von “α–β-Pruning”

Verwandte Techniken finden sich z.B. auch in

Implementierungen von Schach-Programmen.

TTT: Spielzustand

Ein Spielzustand besteht aus mehreren Komponenten. Er soll durch eine `Structure` beschrieben werden:

```
(defstruct state
  field
  last-mover
  visited?
  next-states
  winner)
```

TTT: Spielzustand

Eine Instanz von `state` hat also folgende Komponenten:

- `field`: Spielfeld(siehe unten)
- `last-mover`: Spieler, der zuletzt gezogen hat (läßt sich eigentlich aus field und dem ersten Spieler errechnen)
- `visited?`: Ist dieser Zustand schon einmal besucht worden? (siehe unten)
- `next-states:` Liste von Folgezuständen (also von Instanzen von state)
- `winner:` Gewinner dieses Zustands (siehe unten)

TTT: Spielfeld und Spieler

Das Spielfeld der Größe *n x n* wird dargestellt als eine Liste der Länge *n x n*. Typischerweise beträgt die Größe 3 x 3 Kästchen:

```
(defvar *field-length* 3)
```

Man möchte auf das Feld teilweise über Koordinaten zugreifen. Die Umrechnung geschieht über folgende Funktion:

```
(defun coord (i j)
  (declare (inline))
  (+ (* *field-length* i) j))
```

Zugriff auf das mittlere Feld dann mit

```
(nth (coord 1 1) field)
```

TTT: Spielfeld und Spieler

Es gibt zwei Spieler, A und B. Folgende Funktion abstrahiert von den Namen:

```
(defun adversary (player)
  (if (eq player 'a)
      'b
    'a))
```

Unbesetzte Positionen werden durch `'u` markiert. Das eingangs gezeigte Feld wird also repräsentiert als:

```
'(a u a  u u u  u u b)
```

TTT: Spielverlauf

Wir wollen in zwei Schritten vorgehen:

- Aufbauphase: Konstruktion eines vollständigen Spielbaums
- Spielphase: Möglichst “gutes” Spiel mit Hilfe des Spielbaums

Bemerkung: Die Konstruktion eines *vollständigen* Baums ist für komplexere Spiele ausgeschlossen ...

TTT: Spielverlauf

Der **Spielbaum** besteht aus den einzelnen Spielzuständen. Nachfolgezustände sind all die Zustände, die der gerade “aktuelle” Spieler erreichen kann. Für:

```
| A |   | A |
 -----------
|   |   |   |
 -----------
|   | B | B |
```

sind einige Nachfolgezustände (wenn A am Zug ist):

```
| A |   | A |        | A |   | A |        | A | A | A |
 -----------         -----------          -----------
|   | A |   |      |   |   |   |      |   |   |   |
 -----------         -----------          -----------
|   | B | B |      | A | B | B |      |   | B | B |
```

Der letzte Zustand ist ein *Gewinnzustand* für A.

TTT: Konstruktion des Spielbaums

Die Konstruktion des Spielbaums soll rekursiv durch folgende Funktion durchgeführt werden:

```
(defun play (st player)
  (let ((next-states (compute-next-states st (adversary player))))
    (setf (state-last-mover st) player)
    (setf (state-next-states st) next-states)
    (mapc #'(lambda (nst) (play nst (adversary player)))
          next-states)
    st))
```

Problem: Größe des Suchbaums: $9! > 300.000$.

TTT: Konstruktion des Spielbaums

Beobachtungen:

- Es gibt nur $3^9 < 20.000$ verschiedene Zustände.
- Mehrere Zustände können den selben Nachfolger haben.
- Ein Spielzustand “hängt nicht von Vorgängerzuständen ab”.

Lösung: Konstruktion eines Spiel*graphen*. Dazu:

- play braucht einen Knoten nicht mehr zu bearbeiten, der schon besucht worden ist. Dafür wird der Slot `visited?` eingeführt.
- Die Funktion compute-next-states darf nicht einfach nur neue Zustände generieren, sondern muß evtl. schon betrachtete liefern. Dafür wird eine Tabelle angelegt.

TTT: Verbesserte Konstruktion des Spielbaums

Die Funktion play testet nun zuerst, ob ein Knoten schon besucht worden ist:

```
(defun play (st player)
  (if (state-visited? st)

      ;; current state has already been visited, all
      ;; relevant entries have already been computed
      st

    ;; current state has not yet been visited
    (let ((next-states
           (compute-next-states st (adversary player))))
      (setf (state-last-mover st) player)
      (setf (state-visited? st) T)
      (setf (state-next-states st) next-states)
      (mapc #'(lambda (nst)
                (play nst (adversary player)))
            next-states)
      st)))
```

TTT: Berechnung der Folgezustände

Die Berechnung der Folgezustände eines Zustandes st (wobei Spieler player als nächster zieht) erfolgt durch die Funktion `compute-next-states`, die ungefähr wie folgt arbeitet:

```
(defun compute-next-states (st player)
  (let ((winner (winner-of-state st)))
    (if winner
        ;; this state is a winning state
        ;; --> no follow-up-states
        NIL

      ;; compute free positions and place
      ;; 'player' on them
      .... )))
```

TTT: Berechnung der Folgezustände

Aufgabe: Füllen Sie die Stelle . . . aus. Gehen Sie folgendermaßen vor:

- Berechnen Sie eine Liste der noch unbesetzten Positionen des Felds von `st` (durch Funktion `compute-free-positions`)
- Schreiben Sie eine Funktion `put-player-on-field (field player pos)`, die den Spieler player in dem Feld field auf Position pos setzt (Tip: Verwenden Sie die Funktion `substitute`)
- Verwenden Sie auch die Funktion `create-state-for-field` (siehe unten) und kombinieren Sie die Funktionen in geeigneter Weise (z.B. durch mapcar)

TTT: Funktion `create-state-for-field`

Die Funktion berechnet einen neuen Zustand mit dem entsprechenden Feld-Eintrag, wenn dieser “noch nicht existiert”. Andernfalls gibt sie den bereits erzeugten Zustand zurück:

```
(defun create-state-for-field (field)
  (let ((state-from-table (get-from-table field)))
    (if state-from-table

        state-from-table

      (let ((new-state (make-state :field field
                                   :visited? nil)))
        (insert-into-table field new-state)
        new-state))))
```

TTT: Funktion `create-state-for-field`

Zum Nachschauen wird eine Hash-Tabelle verwendet:

```
(let ((table NIL))

(defun make-empty-table ()
  (setf table (make-hash-table :test #'equal)))
)
```

Aufgabe: Schreiben Sie die Funktionen `get-from-table` und `insert-into-table`.

TTT: Berechnung des Gewinners eines Zustands

Funktion `winner-of-state` berechnet für einen Zustand den Gewinner (A oder B). Im Fall einer unentschiedenen Situation wird `NIL` zurückgegeben.

```
(defun winner-of-state (st)
  (or (winner-of-field (state-field st) 'a)
      (winner-of-field (state-field st) 'b)))

(defun winner-of-field (field player)
  (or (check-row field player)
      (check-column field player)
      (check-diag field player)))
```

TTT: Berechnung des Gewinners eines Zustands

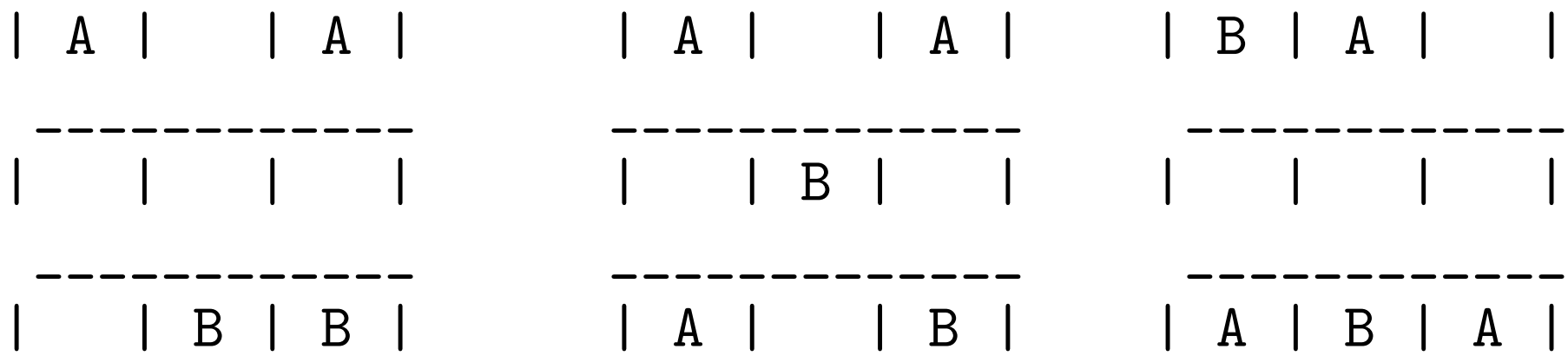
Die Funktion `check-row` überprüft, ob player eine ganze Reihe gefüllt hat:

```
(defun check-row (field player)
  (loop for i from 0 to (- *field-length* 1)
        when (loop for j from 0 to (- *field-length* 1)
                   always (eq (nth (coord i j) field)
                              player))
        return player))
```

Aufgabe: Schreiben Sie in ähnlicher Weise Funktionen `check-column` und `check-diag`.

TTT: Berechnung des “sicheren” Gewinners (I)

Betrachten Sie folgende Zustände:

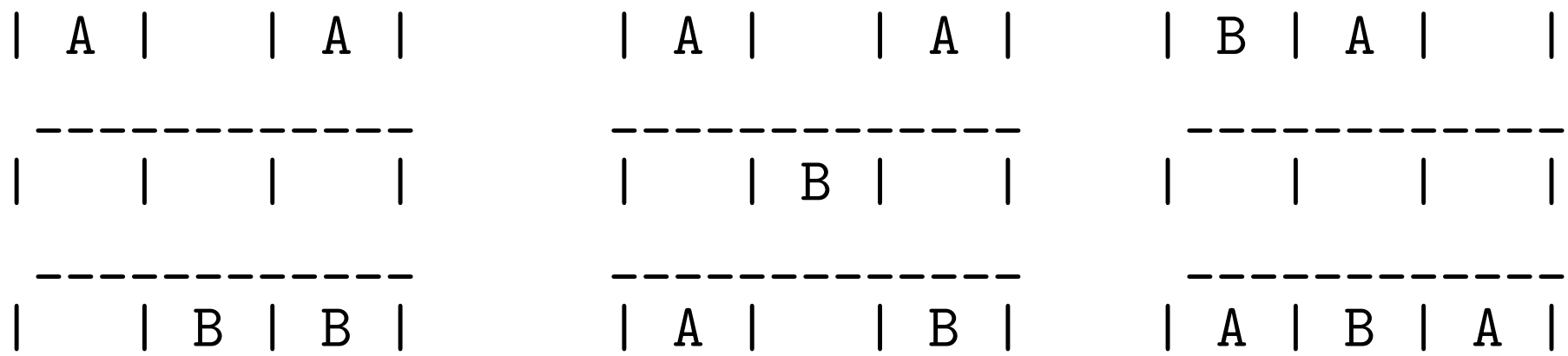


Zum ersten Zustand: Sei A als nächster am Zug. Dann gibt es für A eine sichere Gewinnstrategie, denn es *existiert* ein *Nachfolgezustand*, in dem A gewinnt (und A hat die Wahl).

Zum zweiten Zustand: Sei B als nächster am Zug. Auch hier gibt es für A eine sichere Gewinnstrategie, denn *für* alle *Nachfolgezustände* gilt, daß A das Spiel mit Sicherheit gewinnen kann (egal, wie sich B entscheidet).

TTT: Berechnung des “sicheren” Gewinners

Betrachten Sie folgende Zustände:



Zum dritten Zustand: Sei B als nächster am Zug. Hier gibt es für keinen Spieler eine sichere Gewinnstrategie, denn jeder kann durch einen geeigneten Zug den Gewinn des anderen abwenden.

Diese Form der Berechnung einer möglichst günstigen Gewinnstrategie ist einfacher Spezialfall des sogenannten α – β-Pruning

TTT: Berechnung des "sicheren" Gewinners (II)

Aufgabe: Schreiben Sie aufgrund dieser Überlegungen eine Funktion

```
(defun certain-winner (next-states player) ....)
```

die den Spieler, den Gegner oder NIL zurückgibt, je nachdem, wer die Zustände next-states sicher gewinnt.

Hinweis: Verwenden Sie dazu die Funktionen
(some predicate sequence) und
(every predicate sequence), die testen, ob zumindest ein bzw. alle Elemente einer Sequenz das einstellige *predicate* erfüllen.

TTT: Berechnung des “sicheren” Gewinners

Die Funktion `certain-winner` wird in play verwendet, um den Slot winner zu setzen:

```
(defun play (st player)
  (if (state-visited? st)
      st
    (let ((next-states
           (compute-next-states st (adversary player))))
      ...
      (mapc #'(lambda (nst)
                (play nst (adversary player)))
            next-states)
      (setf (state-winner st)
            (if next-states
                (certain-winner next-states player)
              (winner-of-state st)))
      st)))
```

TTT: Aufbauphase

Aufgabe: Schreiben Sie eine Funktion generate-game-graph, die den Spiel-Graphen generiert und dabei die bisher beschriebenen Funktionen verwendet.

Hinweis: Im wesentlichen muß nur die Tabelle der Spielzustände initialisiert und dann die Funktion play mit geeigneten Anfangswerten aufgerufen werden.
Belegen Sie eine globale Variable *top-state* mit der “Wurzel” des Spielgraphen. Dieser Zugang zum Graphen wird in der Spielphase verwendet.

Hinweis: Vor Aufruf der Funktion generate-game-graph sollten Sie sämtliche Funktionen kompilieren.

TTT: Spielphase

Die Spielphase wird durch Aufruf der Funktion `interactive-play` initiiert:

```
(defun interactive-play ()
  (format T "a is the first, B the second player~%")
  (format T "Who will play first, I (i) or you (y)? :")
  (if (eq (read) 'i)
      (my-move *top-state* 'a)
    (user-move *top-state* 'a)))
```

TTT: Spielphase

Die Funktionen `my-move` und `user-move` bewegen sich durch den vorher konstruierten Graphen. Sie rufen sich dabei gegenseitig rekursiv auf.

- `my-move` führt einen Zug der Maschine durch. Dabei wird mit der Funktion `my-favorite-state` ein bevorzugter Folgezustand ausgewählt.
- `user-move` liest die Koordinaten des Kästchen sein, auf das das der Benutzer seinen nächsten Stein platzieren möchte, und sucht den passenden Folgezustand des aktuellen Knotens.

Aufgabe: Schreiben Sie die Funktionen `my-move` und `user-move`.

TTT: Spielphase

Die Funktion `my-favorite-state` sucht in einer Liste von Nachfolgezuständen nach einem Zustand, in dem `player` sicher gewinnt oder zumindest nicht sicher verliert:

```
(defun my-favorite-state (state-list player)
  (or (find player state-list
            :test #'(lambda (x y)
                      (eq x (state-winner y))))
      (find nil state-list
            :test #'(lambda (x y)
                      (eq x (state-winner y))))))
```

TTT: Spielphase

Es fehlen jetzt nur noch Funktionen, die Ein- und Ausgabe komfortabler gestalten. `print-field` gibt ein Feld im *n x n*-Format aus:

```
(defun print-field (field)
  (loop for i from 0 to (- *field-length* 1) do
    (loop for j from 0 to (- *field-length* 1) do
      (format T " ~A "
          (if (eq (nth (coord i j) field) 'u)
              " "
              (nth (coord i j) field))))
    (format T "~%")))
```

Aufgabe: Schreiben Sie eine Funktion `pretty-print-field` für eine etwas schönere Ausgabe.

TTT: Erweiterung und Optimierung

Folgende Erweiterungen und Verbesserungen lassen sich vorstellen:

Abfangen von fehlerhaften Eingaben

- Im allgemeinen ist die Generierung eines ganzen Spielbaums nicht zu machen. überlegen Sie sich
- Techniken, mit denen nur ein Teilbaum (z.B. bis zu einer bestimmten Tiefe) konstruiert werden kann.
- Optimieren Sie die Auswahl des Folgezustands `(in my-favorite-state),` indem sie in dem winner-Slot vermerken, wie “wahrscheinlich” ein Gewinn in dem Zustand für einen jeden Spieler ist. Besonders die Funktion certain-winner muß hier angepaßt werden.
- Die Funktion `create-state-for-field` schaut explizit in einer Tabelle nach, um einen evtl. schon vorhanden Zustand zurückzugeben. Stattdessen kann auch Memoisierung angewendet werden. Welche Veränderungen sind hierzu notwendig?

Zusammenfassung

In Lisp gibt es:

- “Einfache”Datentypen wie ganze, rationale und *floating point* Zahlen, Characters, Symbole
- Sequenzen wie Listen,Strings
- Komplexe Datentypen wie Arrays und Hash-Tabellen
- Vom Benutzer definierbare, z.B. durch `defstruct`

Typen sind in einer Typhierarchie angeordnet, so sind z.B. Listen und Strings Untertypen von Sequenzen.

Mit `typep` kann überprüft werden, ob ein Objekt von einem bestimmten Typ ist, mit `type-of` kann der Typ eines Objekts berechnet werden.

Im Gegensatz zu vielen Sprachen ist Lisp dynamisch typisiert.

Ausblick: Typisierungsfragen sind auch relevant im Rahmen von objektorientierter Programmierung. ~> später