Übungen zu Allgemeine Informatik I
WS 1998/99

Andreas Franz Borchert
Universität Ulm

27. Januar 1999

Aufgabe 13: No way out?

```
thales$ cat little_maze
XXXXXXX
X     X
X XXXXX

X XXXXX
X     X
XXXXXXX
thales$ pathfinder little_maze
XXXXXXX
X*****X
X*XXXXX
*******
X XXXXX
X     X
XXXXXXX
thales$
```

- Für ein vorgegebenes Labyrinth soll ein Weg gefunden werden.
- Der Eingang ist auf der linken Seite und der Ausgang auf der rechten.
- Alles außer Leerzeichen wird als Wand interpretiert.
- Der eigene Weg wird mit * markiert.

Fehlersituationen

```
thales$ pathfinder /dev/null
Labyrinth ist zu klein!
thales$ echo X | pathfinder
Labyrinth ist zu klein!
thales$ perl -e 'print "X" x 1000, "\n"' | pathfinder
Das Labyrinth ist zu gross!
thales$ perl -e 'print "XXX\n" x 1000' | pathfinder
Das Labyrinth ist zu gross!
thales$ cat bad1
XXX
  X
XXX
thales$ pathfinder bad1
Keinen Ausgang gefunden!
thales$ cat bad2
XXX
  X
X X
thales$ pathfinder bad2
Bitte keinen Ausgang in der letzten Zeile!
thales$
```

- Zahlreiche Vorbedingungen sind zu überprüfen.

Sonderfälle

```
turing$ cat locked1
XXX
 X
XXX
turing$ pathfinder locked1
Eingang ist verbarrikadiert!
turing$ mv in3 locked2
turing$ pathfinder locked2
XXXXXXXXXXXXXXXXXXXXXXXXXXX
X***********X             X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
**         *X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
X*         *X             X
X***********X             X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
turing$
```

- Nicht alle Labyrinthe erlauben ein Durchkommen...

Deklarationen

PathFinder.m2

```
CONST
   maxsize = 128;
   minsize = 3;
   space = " ";
   walk = "*";

TYPE
   MazeSize = INTEGER [0..maxsize];
   MazeIndex = INTEGER [0..maxsize-1];
   Maze = ARRAY MazeIndex, MazeIndex OF CHAR;
```

- Das Labyrinth wird genau in der Form abgelegt, wie es in der Eingabe vorgefunden wird.
- Nur Zeilentrenner werden dahingehend speziell interpretiert, das es in der nächsten Zeile der Matrix, die das Labyrinth repräsentiert, weitergeht.
- Mit `minsize` und `maxsize` werden die zulässigen Seitenlängen eingegrenzt. Labyrinthe, die eine kleinere Seitenlänge als `minsize` haben, sind nicht sinnvoll.
- Alle Zeichen ungleich `space` (und dem Zeilentrenner) werden als Wände betrachtet.
- Wegemarkierungen werden in der Eingabe nicht akzeptiert.

Globale Variablen

PathFinder.m2

```
VAR
   (* Kommandozeilenargumente *)
   stepwise: BOOLEAN;
   query: BOOLEAN;
   input: FILE;

   (* Resultat der Eingabe, das spaeter modifiziert wird *)
   maze: Maze;
   width, height: MazeSize;
   entry, exit: MazeIndex;
```

- Die Kommandozeilenparameter werden in globalen Variablen abgelegt:

stepwise	Option **-s** gesetzt, d.h. ist nach jedem Schritt eine Ausgabe gewünscht?
query	Option **-q** gesetzt, d.h. soll nach jedem Schritt zurückgefragt werden?
input	Eingabeverbindung: entweder *StdIO.stdin* oder eine erfolgreich geöffnete Datei.

- Das in der Eingabe vorgefundene Labyrinth wird in folgenden Variablen abgelegt (und teilweise nachher modifiziert):

maze	Matrix mit Wänden und freien Räumen.
width und *height*	Seitenlängen des Labyrinths.
entry und *exit*	Zeilenindex des Eingangs (auf der linken Seite) und des Ausgangs (auf der rechten Seite).

Hauptprogramm

PathFinder.m2

```
(*
   Allgemeine Informatik I / Programmieren I    WS 1998/1999
   Musterloesung fuer das Blatt 11, Aufgabe 13
   Andreas Borchert, 21. Januar 1998
*)
MODULE PathFinder;

   FROM Arguments IMPORT InitArgs, GetFlag, GetArg, AllArgs,
      Usage;
   FROM ASCII IMPORT nl;
   FROM FtdIO IMPORT FwriteString, FwriteLn;
   FROM InOut IMPORT WriteString, Write, WriteLn, ReadString;
   FROM StdIO IMPORT FILE, MODE, Fopen, Fgetc, stdin, stderr;
   FROM SysExit IMPORT Exit;
   FROM SysPerror IMPORT Perror;

   (* ... *)

BEGIN
   ProcessArgs;
   IF ReadMaze(input, maze, width, height, entry, exit) THEN
      SolveMaze(maze, width, height, entry, 0);
      WriteMaze(maze, width, height);
   END;
END PathFinder.
```

- Nach dem Bearbeiten der Argumente und dem erfolgreichen Einlesen des Labyrinths, wird von `SolveMaze` nach einem Lösungsweg gesucht und dieser dann von `WriteMaze` ausgegeben.

Argumentverarbeitung

PathFinder.m2

```
PROCEDURE ProcessArgs;
   VAR
      flag: CHAR;
      filename: ARRAY [0..511] OF CHAR;
BEGIN
   query := FALSE; stepwise := FALSE;
   InitArgs("[-q] [-s] [maze]");
   WHILE GetFlag(flag) DO
      CASE flag OF
      | "q":   query := TRUE; stepwise := TRUE;
      | "s":   stepwise := TRUE;
      ELSE
         Usage;
      END;
   END;
   IF GetArg(filename) THEN
      IF ~Fopen(input, filename,
            read, (* buffered = *) TRUE) THEN
         Perror(filename); Exit(1);
      END;
   ELSE
      input := stdin;
      IF query THEN
         Error("-q vertraegt sich nicht mit ...!");
         Exit(1);
      END;
   END;
   AllArgs;
END ProcessArgs;
```

- `Error` gibt (analog zu *Arguments.Usage* eine Fehlermeldung auf *StdIO.stderr* aus.

Das Eröffnen einer Datei

StdIO.d

```
TYPE
   FILE; (* hidden *)
   MODE = (read, write, append);

PROCEDURE Fopen(VAR f: FILE; name: ARRAY OF CHAR; mode: MODE;
                buffered: BOOLEAN) : BOOLEAN;
```

PathFinder.m2

```
FROM StdIO IMPORT FILE, MODE, Fopen, Fgetc, stdin, stderr;
FROM SysExit IMPORT Exit;
FROM SysPerror IMPORT Perror;

(* ... *)

IF ~Fopen(input, filename, read, (* buffered = *) TRUE) THEN
   Perror(filename); Exit(1);
END;
```

- Der dritte Parameter bestimmt den Öffnungsmodus: Entweder zum Lesen, Schreiben oder zum Anhängen am Ende der Datei. Achtung: Wer eine Datei zum Schreiben eröffnet, zerstört den alten Dateiinhalt, falls es die Datei vorher schon gab. Der Anhäng-Moduls *append* verfährt da schonender.

- Der vierte Parameter entscheidet, ob die Ein- oder Ausgabe gepuffert werden soll. Pufferung sorgt für mehr Effizienz, ist aber bei interaktiver Ein- oder Ausgabe möglicherweise unpraktisch.

Ausgabe von Fehlermeldungen

PathFinder.m2

```
FROM FtdIO IMPORT FwriteString, FwriteLn;
FROM StdIO IMPORT FILE, MODE, Fopen, Fgetc, stdin, stderr;

(* ... *)

PROCEDURE Error(msg: ARRAY OF CHAR);
BEGIN
   FwriteString(stderr, msg); FwriteLn(stderr);
END Error;
```

- *FtdIO* exportiert eine Reihe von Prozeduren, die – abgesehen von einem weiteren Parameter für die Dateiverbindung – denen von *InOut* gleichen.

- Die drei von *StdIO* exportierten Standardverbindungen *stdin*, *stdout* und *stderr* sind mit der Standardeingabe, -ausgabe und dem Fehlerkanal verbunden.

- Die Fehlerausgabe läßt sich von der normalen Ausgabe beim Aufruf auf der Kommandozeile trennen:

  ```
  pathfinder maze >ausgabe 2>fehler
  ```

  Hinweis: *stderr* hat die Kanalnummer (*file descriptor*) 2.

Einlesen eines Labyrinths

PathFinder.m2

```
PROCEDURE ReadMaze(input: FILE;
                   VAR maze: Maze;
                   VAR width, height: MazeSize;
                   VAR entry, exit: MazeIndex) : BOOLEAN;
   VAR
      line, column: MazeSize;
      ch: CHAR; (* zuletzt eingelesenes Zeichen *)
BEGIN
   line := 0; column := 0; width := 0;
   WHILE Fgetc(ch, input) DO
      IF ch = nl THEN
         INC(line); column := 0;
      ELSE
         maze[line, column] := ch; INC(column);
      END;
   END;
   height := line; width := column;
   RETURN TRUE
END ReadMaze;
```

- So einfach könnte das Einlesen eines Labyrinths sein, wenn auf Überprüfungen verzichtet wird.

- Ein Teil der Überprüfungen und die Ermittlung von Ein- und Ausgang könnte wirklich auf später verschoben werden. Einige Überprüfungen (wie die Einhaltung der maximalen Seitenlängen) müssen jedoch in jedem Fall beim Einlesen durchgeführt werden.

- Dieser erster Entwurf verzichtet noch auf die Bestimmung von `entry` und `exit`.

Einlesen eines Labyrinths

PathFinder.m2

```
WHILE Fgetc(ch, input) DO
   IF ch = nl THEN
      (* ... *)
      INC(line); column := 0;
   ELSE
      IF ch = space THEN
         (* ... *)
         spaceSeen := TRUE;
      ELSIF column = 0 THEN
         spaceSeen := FALSE;
      END;
      IF (line >= maxsize) OR (column >= maxsize) THEN
         WriteString("Das Labyrinth ist zu gross!"); WriteLn;
         RETURN FALSE
      END;
      (* ... *)
      maze[line, column] := ch; INC(column);
   END;
END;
```

- Sehr häufig wie in diesem Fall gehen Einlese-Schleifen mit vielen Überprüfungen einher.

- Tip: Gestalten Sie die Schleifen-Bedingung so einfach wie nur irgend möglich (z.B. wie hier bis zum Erreichen des Eingabe-Endes).

- Wenn dann die Schleife beendet ist, brauchen Sie nicht zu untersuchen, *warum* die Schleife endete.

- Behandeln Sie Fehler- und Sonderfälle dadurch, daß sie eine Prozedur mit **RETURN FALSE** beenden.

Einlesen eines Labyrinths

PathFinder.m2

```
PROCEDURE ReadMaze(input: FILE;
                   VAR maze: Maze;
                   VAR width, height: MazeSize;
                   VAR entry, exit: MazeIndex) : BOOLEAN;
   VAR
      line, column: MazeSize;
      ch: CHAR; (* zuletzt eingelesenes Zeichen *)
      entryFound, exitFound: BOOLEAN;
      spaceSeen: BOOLEAN;
BEGIN
   line := 0; column := 0; width := 0;
   entryFound := FALSE; exitFound := FALSE;
   spaceSeen := FALSE;
   (* ... *)
END ReadMaze;
```

- Die Musterlösung führt sämtliche Prozeduren in der Eingabeprozedur zum jeweils frühestmöglichen Zeitpunkt durch.
- In den Variablen `entryFound` und `exitFound` wird notiert, ob Ein- und Ausgang bereits gefunden worden sind, da wir davon jeweils genau einen erwarten.
- `spaceSeen` ist genau dann wahr, wenn in der *letzten* eingelesenen Zeile mindestens ein Leerzeichen gesehen worden ist. Damit läßt sich am Ende sofort überprüfen, ob ein “Loch” in der untersten Zeile vorhanden ist (was nicht erlaubt ist).

Einlesen eines Labyrinths

PathFinder.m2

```
WHILE Fgetc(ch, input) DO
   IF ch = nl THEN
      IF line = 0 THEN
         IF column < minsize THEN
            WriteString("Labyrinth ist zu klein!");
            WriteLn;
            RETURN FALSE
         END;
         width := column;
      ELSIF column # width THEN
         WriteString("Das Labyrinth ist nicht rechteckig!");
         WriteLn;
         RETURN FALSE
      END;
      INC(line); column := 0;
   ELSE
      (* ... *)
   END;
END;
```

- Beim Zeilenende ist darauf zu achten, daß alle Zeilen gleich lang sind.
- Bei der ersten Zeile fällt diese Überprüfung weg. Dafür wird hier auf die Mindestlänge von `minsize` geachtet.
- Entgegen dem Entwurf wird gleich in der 1. Zeile `width` gesetzt (damit für die weiteren Zeilen der Vergleichswert vorliegt).

Einlesen eines Labyrinths

PathFinder.m2

```
WHILE Fgetc(ch, input) DO
   IF ch = nl THEN
      (* ... *)
   ELSE
      IF ch = space THEN
         (* ... diverse Ueberpruefungen ... *)
         spaceSeen := TRUE;
      ELSIF column = 0 THEN
         spaceSeen := FALSE;
      END;
      IF (line >= maxsize) OR (column >= maxsize) THEN
         WriteString("Das Labyrinth ist zu gross!");
         WriteLn;
         RETURN FALSE
      END;
      IF ch = walk THEN
         WriteString("Bitte keine Markierungszeichen ");
         WriteString("verwenden!"); WriteLn;
         RETURN FALSE
      END;
      maze[line, column] := ch; INC(column);
   END;
END;
```

- Bei Leerzeichen müssen diverse Überprüfungen vorgenommen werden, um unzulässige Ein- und Ausgänge abzufangen (siehe folgende Folie).

Einlesen eines Labyrinths

PathFinder.m2

```
WHILE Fgetc(ch, input) DO
   IF ch = nl THEN
      (* ... *)
   ELSE
      IF ch = space THEN
         IF line = 0 THEN
            WriteString("Bitte keinen Eingang in der");
            WriteString("1. Zeile!"); WriteLn;
            RETURN FALSE
         ELSIF column = 0 THEN
            IF entryFound THEN
               WriteString("Mehr als einen Eingang ");
               WriteString("gefunden!"); WriteLn;
               RETURN FALSE
            ELSE
               entry := line; entryFound := TRUE;
            END;
         ELSIF column = width-1 THEN
            IF exitFound THEN
               WriteString("Mehr als einen Ausgang ");
               WriteString("gefunden!"); WriteLn;
               RETURN FALSE
            ELSE
               exit := line; exitFound := TRUE;
            END;
         END;
         spaceSeen := TRUE;
      ELSIF column = 0 THEN
         spaceSeen := FALSE;
      END;
      (* ... *)
   END;
END;
```

Einlesen eines Labyrinths

PathFinder.m2

```
IF (width < minsize) OR (height < minsize) THEN
   WriteString("Labyrinth ist zu klein!"); WriteLn;
   RETURN FALSE
ELSIF spaceSeen THEN
   WriteString("Bitte keinen Ausgang in der letzten Zeile!");
   WriteLn; RETURN FALSE
ELSIF ~entryFound THEN
   WriteString("Keinen Eingang gefunden!"); WriteLn;
   RETURN FALSE
ELSIF ~exitFound THEN
   WriteString("Keinen Ausgang gefunden!"); WriteLn;
   RETURN FALSE
ELSIF maze[entry, 1] # space THEN
   WriteString("Eingang ist verbarrikadiert!"); WriteLn;
   RETURN FALSE
END;
IF column # 0 THEN
   WriteString("Das Ende der letzten Zeile wird vermisst!");
   WriteLn;
   RETURN FALSE
END;
RETURN TRUE
```

- Eine Reihe weiterer Kontrollen sind nach dem Eingabe-Ende erforderlich.
- Insbesondere wird überprüft, ob auch ein minimaler Bewegungsspielraum vorhanden ist, damit `SolveMaze` den Rand auch beim Start nicht besonders berücksichtigen muß.

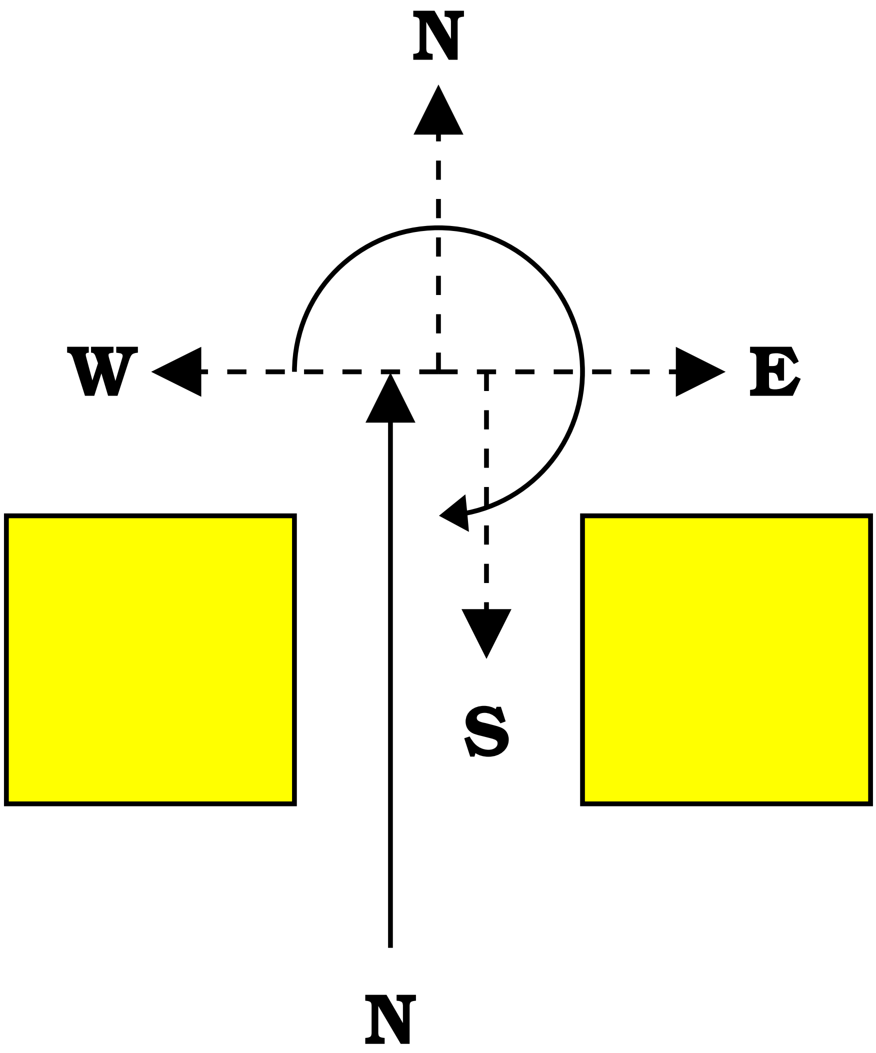
Wegsuche im Labyrinth

PathFinder.m2

```
PROCEDURE SolveMaze(VAR maze: Maze;
                    width, height: MazeSize;
                    startx, starty: MazeIndex);
   TYPE
      Direction = (east, south, west, north);
      Delta = [-1..1];
   VAR
      dx, dy: ARRAY Direction OF Delta;
      x, y: MazeIndex;
      direction: Direction;

BEGIN (* SolveMaze *)
   (* Initialisierung von dx und dy *)
   direction := MIN(Direction); x := startx; y := starty;
   REPEAT
      maze[x, y] := walk;
      IF stepwise THEN
         WriteMaze(maze, width, height);
         IF query & ~Weiter() THEN
            RETURN
         END;
      END;
      direction := Find(x, y, direction);
      INC(x, dx[direction]); INC(y, dy[direction]);
   UNTIL Exit(x, y);
   maze[x, y] := walk;
END SolveMaze;
```

Wegsuche im Labyrinth



PathFinder.m2

```
PROCEDURE Find(x, y: MazeIndex; dir: Direction) : Direction;
BEGIN
   dir := PreviousDirection(dir);
   LOOP
      IF ~Wall(x, y, dir) THEN
         RETURN dir
      END;
      dir := NextDirection(dir);
   END;
END Find;
```

Wegsuche im Labyrinth

PathFinder.m2

```
TYPE
   Direction = (east, south, west, north);

PROCEDURE PreviousDirection(dir: Direction) : Direction;
   (* nach rechts schauen *)
BEGIN
   IF dir = MIN(Direction) THEN
      RETURN MAX(Direction)
   ELSE
      DEC(dir); RETURN dir
   END;
END PreviousDirection;

PROCEDURE NextDirection(dir: Direction) : Direction;
   (* nach links schauen *)
BEGIN
   IF dir = MAX(Direction) THEN
      RETURN MIN(Direction)
   ELSE
      INC(dir); RETURN dir
   END;
END NextDirection;
```

- Bei der Linke- oder Rechte-Hand-Regel ist die Reihenfolge der Richtungen signifikant. Wenn alternativ `(east, north, west, south)` verwendet worden wäre, hätten wir die Rechte-Hand-Regel.

- **INC** und **DEC** funktionieren auch für Aufzählungs-Typen.

- Bei der Linke-Hand-Regel schaut `PreviousDirection` nach links und `NextDirection` nach rechts.

Wegsuche im Labyrinth

PathFinder.m2

```
TYPE
   Delta = [-1..1];
VAR
   dx, dy: ARRAY Direction OF Delta;

PROCEDURE Wall(x, y: MazeIndex; dir: Direction) : BOOLEAN;
   VAR
      ch: CHAR;
BEGIN
   ch := maze[x + dx[dir], y + dy[dir]];
   RETURN (ch # space) & (ch # walk)
END Wall;

(* ... *)

dx[east] := 0; dx[south] := 1;
dx[west] := 0; dx[north] := -1;
dy[east] := 1; dy[south] := 0;
dy[west] := -1; dy[north] := 0;

(* ... *)

INC(x, dx[direction]); INC(y, dy[direction]);
```

- Der Ermittlung der Koordinaten von Nachbarfeldern dienen die beiden Arrays `dx` und `dy`.
- Wenn wir auf dem Rand sind, ist (mit Ausnahme der Anfangssituation) das Ziel bereits erreicht. Daher kann dieser Fall vernachlässigt werden.

Wegsuche im Labyrinth

PathFinder.m2

```
PROCEDURE Weiter() : BOOLEAN;
   (* Implementierung der Option -q *)
   VAR
      answer: ARRAY[0..0] OF CHAR;
BEGIN
   WriteString("Weiter?"); ReadString(answer);
   RETURN answer[0] # "n"
END Weiter;

PROCEDURE Exit(x, y: MazeIndex) : BOOLEAN;
   (* Liegt (x,y) am Rand des Labyrinths? *)
BEGIN
   RETURN (x = 0) OR (x = height-1) OR
          (y = 0) OR (y = width-1) OR
          (x = startx) & (y = starty)
END Exit;
```

- `Weiter` verwendet *ReadString* anstelle von *Read*, um ein “j”, gefolgt von RETURN, genauso wie ein einfaches RETURN zu behandeln.

Ausgabe des Labyrinths

PathFinder.m2

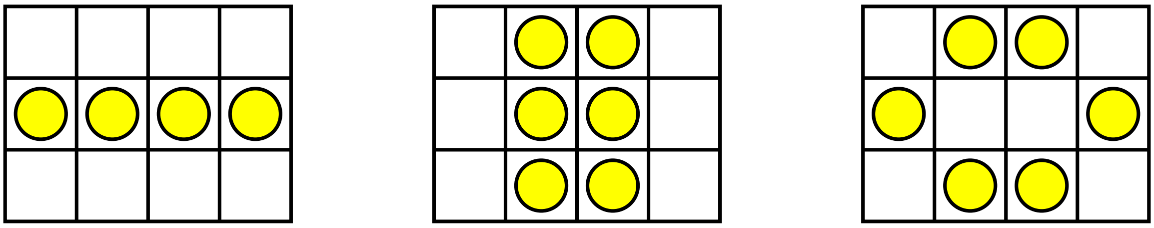
```
PROCEDURE WriteMaze(maze: Maze; width, height: MazeSize);
   (* Ausgabe des Labyrinths auf stdout *)
   VAR
      line, column: MazeIndex;
BEGIN
   FOR line := 0 TO height-1 DO
      FOR column := 0 TO width-1 DO
         Write(maze[line, column]);
      END;
      WriteLn;
   END;
END WriteMaze;
```

- Wenn Nullbytes jeweils am Ende der Zeilen wären, könnte die innere Schleife durch ein

  `WriteString(maze[line]);`

  ersetzt werden.

Tips zur 14. Aufgabe

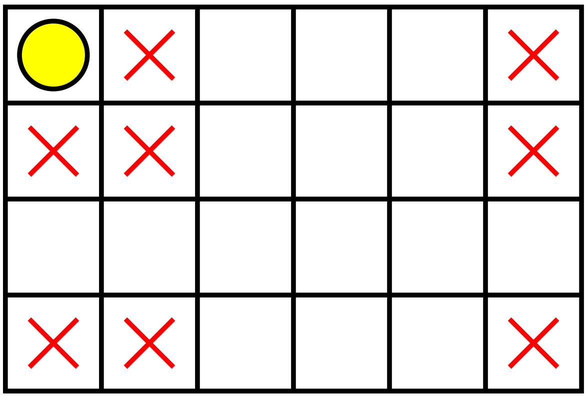


- Bei Conway's Game of Life handelt es sich um einen zellulären Automaten in der Ebene, bei der jedes Feld zwei Zustände besitzt: gefüllt (1) oder leer (0).

- Der Zustandsübergang eines jeden Feldes geschieht in Abhängigkeit des eigenen Zustandes und der Anzahl der gefüllten Nachbarn (0 - 8):

	Anzahl der Nachbarn			
	0-1	2	3	4-8
gefüllt (1)	0	1	1	0
leer (0)	0	0	1	0

Aufgabenstellung



- Das zu erstellende Programm soll ein Szenario aus einer (auf der Kommandozeile zu spezifierenden) Datei einlesen und mit Hilfe des Modules *MainWin* die Entwicklung bildschirmorientiert ausgeben.

- Leider steht nicht (wie im mathematischen Modell) beliebig Platz zur Verfügung, so daß ein Problem mit dem Rand gibt. Zwei Varianten sollen unterstützt werden:

  - Die zur Verfügung stehende Fläche wird als Torus behandelt (siehe Nachbarschaftsbeispiel im Diagramm).

  - Alle Felder des nicht darstellbaren Bereiches sind grundsätzlich 0.

- Analog zur letzten Aufgabe soll die Option **-q** jeweils rückfragen, ob es weitergehen soll. Auf diese Weise läßt sich die Entwicklung Schritt für Schritt verfolgen.

Tips

- Packen Sie den Zustandsübergang in eine Prozedur, die als Werteparameter den aktuellen Stand erhält und über einen **VAR**-Parameter den neuen Stand zurückliefert.

- Sie benötigen in jedem Fall *zwei* Matrizen beim Übergang, da die Zählung der Nachbarn nicht durch Änderungen beeinträchtigt werden darf.

- Denken Sie entweder in Zeilen und Spalten oder in X- und Y-Koordinaten und reflektieren Sie dies in den Variablennamen. Wenn Sie beide Notationen mischen, endet das fast immer im Desaster.

Hinweise zu *MainWin*

Hello.m2

```
MODULE Hello;

   FROM MainWin IMPORT SetPos, WriteString, WriteLn;

BEGIN
   SetPos(10, 10); WriteString("Hello World"); WriteLn;
END Hello.
```

- *MainWin* ist eine angenehm einfache Schnittstelle zu *Windows*, das bildschirmorientierte Ein- und Ausgaben unterstützt.
- Bei *InOut* findet Ein- und Ausgabe Zeile für Zeile statt. Wenn der Bildschirm gefüllt ist, dann wird der Inhalt nach oben geschoben, um unten Platz für eine neue Zeile zu schaffen.
- Bei *MainWin* wird der Bildschirminhalt nicht verschoben. Stattdessen gibt es die Möglichkeit, sich frei auf dem Bildschirm zu positionieren.

Hinweise zu *MainWin*

Chaos.m2

```
MODULE Chaos;

   FROM RandomGenerator IMPORT Random, Flip;
   FROM MainWin IMPORT SetPos, Write, lines, columns, Flush;

BEGIN
   LOOP
      SetPos(Random(0, lines-1), Random(0, columns-1));
      IF Flip() THEN
         Write("X");
      ELSE
         Write(" ");
      END;
      Flush;
   END;
END Chaos.
```

- *MainWin* kennt zwei Bildschirminhalte: den für den Benutzer sichtbaren und den internen.
- Alle Schreib-Operationen operieren zunächst nur auf dem internen Bildschirminhalt.
- Der sichtbare Bildschirm wird dann aktualisiert, wenn etwas gelesen wird, die Operation *Flush* ausgeführt wird oder das Programm endet.

Hinweise zu *MainWin*

WinPath.m2

```
FROM MainWin IMPORT SetPos, lines, columns, Write, Clear,
   Flush, WriteString, WriteLn, Read;

(* ... *)

PROCEDURE WriteMaze(maze: Maze; width, height: MazeSize);
   VAR
      line, column: MazeIndex;
BEGIN
   SetPos(0, 0);
   FOR line := 0 TO height-1 DO
      FOR column := 0 TO width-1 DO
         Write(maze[line, column]);
      END;
      WriteLn;
   END;
   Flush;
END WriteMaze;
```

- Selbst Programme, die ursprünglich auf *InOut* basierten, können rasch an *MainWin* angepaßt werden.

Hinweise zu *MainWin*

WinPath.m2

```
PROCEDURE Weiter() : BOOLEAN;
   (* Implementierung der Option -q *)
   VAR
      answer: CHAR;
BEGIN
   IF continue OR skip THEN
      RETURN TRUE
   ELSE
      SetPos(lines-1, 0);
      WriteString("Weiter?"); Read(answer);
      SetPos(lines-1, 0);
      WriteString("        "); Flush;
      CASE answer OF
      | "c":   continue := TRUE; RETURN TRUE
      | "n":   RETURN FALSE
      | "s":   skip := TRUE; RETURN TRUE
      ELSE
         RETURN TRUE
      END;
   END;
END Weiter;
```

- Für Benutzereingaben ist die letzte Zeile ein geeignetes Feld.
- Nach erfolgter Eingabe ist es möglicherweise sinnvoll, Frage und Eingabe wieder zu entfernen.

Hinweise zu *MainWin*

WinPath.m2

```
BEGIN
   Clear; Flush;
   ProcessArgs;
   IF ReadMaze(input, maze, width, height, entry, exit) THEN
      SolveMaze(maze, width, height, entry, 0);
      WriteMaze(maze, width, height);
   END;
   SetPos(lines-1, 0); Flush;
END WinPath.
```

- Damit die Fehlermeldungen bei der Abarbeitung der Kommandozeile nicht verschwinden, empfiehlt es sich vorher *Clear* und *Flush* aufzurufen.