Übungen zu Allgemeine Informatik I WS 1998/99

Andreas Franz Borchert
Universität Ulm

3. Februar 1999

Typenkompatibilität in Modula-2

```
oberon$ cat Error.m2
MODULE Error;

   VAR
      int: INTEGER;
      card: CARDINAL;

BEGIN
   IF int > card THEN
   END;
END Error.
oberon$ m2c Error.m2
errors in Error.m2
   8       IF int > card THEN
                        ^ type incompatible operands
oberon$
```

- Auf den ersten Blick wirken die Kompatibilitätsregeln in Modula-2 etwas sehr streng und eigenwillig.

- Dennoch sind sie wohlbegründet, wenn der Hintergrund bekannt ist. So gibt es zum Beispiel zwei unterschiedlich definierte Vergleiche für ganze Zahlen (**INTEGER**) und nicht-negative ganze Zahlen (**CARDINAL**). Welcher sollte also in diesem Beispiel genommen werden?

Identität

```
TYPE
   Rec1 = RECORD a, b: CARDINAL END;
   Rec2 = Rec1;
   Rec3 = Rec1;
   Rec4 = RECORD a, b: CARDINAL END;
```

- Jeder Typ ist zu sich selbst identisch.
- Zwei Typen sind genau dann zueinander identisch, wenn sie beide von der gleichen Typdefinition her stammen.

  Obiges Beispiel illustriert dies: *Rec2* und *Rec3* sind identisch, weil sie beide von *Rec1* abgeleitet sind. Obwohl *Rec4* die gleiche Struktur wie *Rec1* besitzt, ist es nicht identisch zu *Rec1*, *Rec2* oder *Rec3*.

Kompatibilität

Ein Typ *T1* ist kompatibel zu einem Typ *T2* genau dann, wenn mindestens eine der folgenden Bedingungen gilt:

- *T1* ist identisch zu *T2*.
- *T1* ist ein Unterbereich von *T2*.
- *T1* und *T2* sind Unterbereiche identischer Typen.
- *T1* ist identisch zu **CHAR** oder einem Unterbereich davon und *T2* ist der Typ einer Zeichenkettenkonstanten der Länge eins.
- *T1* ist identisch zu **SYSTEM.ADDRESS** und *T2* identisch zu **CARDINAL**.
- *T1* ist der Typ der Konstanten **NIL** und *T2* ein beliebiger Zeigertyp.
- *T2* ist kompatibel zu *T1*.
- Es existiert ein Typ *T3*, der zu *T1* und *T2* kompatibel ist.

Kompatibilität

```
TYPE
   Int10 = INTEGER [1..10];
   Int5 = Int10 [1..5];
   Card10 = [1..10]; (* Basistyp CARDINAL! *)
TYPE
   Colours = (red, green, blue, white, black);
   RGBColours = [red..blue];
   BWColours = [white..black];
```

- *Int10* ist kompatibel zu **INTEGER** und *Int5*, jedoch nicht zu **CARDINAL** oder *Card10*.
- Die Kompatibilitäts-Relation ist transitiv: *Int5* ist kompatibel zu **INTEGER**.
- *RGBColours* ist kompatibel zu *BWColours*, obwohl der Wertebereich sich nicht überschneidet.

Zuweisungskompatibilität

Ein Typ *T1* ist genau dann zuweisungskompatibel zu *T2*, wenn mindestens eine der folgenden Bedingungen zutrifft:

- *T1* und *T2* sind zueinander kompatibel.
- *T1* und *T2* sind jeweils kompatibel zu einer der folgenden Typen: **CARDINAL**, **INTEGER**, **LONGCARD** und **LONGINT**.
- *T1* und *T2* sind jeweils identisch zu **REAL** oder **LONGREAL**.
- *T1* ist vom Typ **SYSTEM.ADDRESS** und *T2* ein beliebiger Zeigertyp (oder umgekehrt).

Regeln in Modula-2

- Typenidentität wird verlangt bei **VAR**-Parametern.
- Kompatibilität wird benötigt bei der Mehrheit der Operatoren. Ebenfalls muß die Laufvariable einer **FOR**-Anweisung kompatibel zum Zielwert sein und bei **CASE**-Anweisungen müssen die Konstanten der einzelnen Fälle jeweils kompatibel zum Selektions-Ausdruck sein.
- Zuweisungskompatibilität reicht aus bei Zuweisungen, normalen Parameterübergaben (*call by value*), bei Indizes, **RETURN**-Anweisungen und bei dem Startwert einer **FOR-Anweisung**.

Konvertierungen: ORD

```
IF ORD(int) > card THEN
   (* ... *)
END;
```

- **ORD** konvertiert sein Argument in eine **CARDINAL**-Zahl.
- **ORD** akzeptiert alle ganzzahligen Typen, **CHAR** einschließlich Unterbereichen davon, **BOOLEAN** und beliebige Aufzählungstypen.
- Wenn das Resultat nicht als **CARDINAL** darstellbar ist (z.B. weil die übergebene Zahl negativ ist), dann kommt es zu einem Laufzeitfehler.

Konvertierungen: VAL

```
IF int > VAL(INTEGER, card) THEN
   (* ... *)
END;
```

- **VAL** konvertiert sein zweites Argument in den Typ, der als erstes angegeben ist.
- Der angegebene Typ muß skalar sein, d.h. entweder ganzzahlig oder ein beliebiger Aufzählungstyp oder **CHAR** oder **BOOLEAN**.
- Der Typ des zweiten Arguments muß zu **CARDINAL**, **INTEGER**, **LONGCARD** oder **LONGINT** kompatibel sein.
- Das zweite Argument wird dann in den angegebenen Typ konvertiert, wobei zur Laufzeit überprüft wird, ob es innerhalb des Bereiches des Ergebnistyps darstellbar ist.

```
oberon$ cat ValCrash.m2
MODULE ValCrash;
   TYPE Colours = (red, green, blue, white, black);
   VAR colour: Colours; card: [1..10];
BEGIN
   card := 10;
   colour := VAL(Colours, card);
END ValCrash.
oberon$ m2c -o valcrash ValCrash.m2 && valcrash

runtime error in module ValCrash at line 6:
CARDINAL value 10 out of [0..4]
Abort(coredump)
oberon$
```

Konvertierung durch Zuweisung

```
tmpcard := int;
IF tmpcard > card THEN
   (* ... *)
END;
```

- Da die Zuweisungskompatibilität großzügiger ist als die striktere Kompatibilität, können Kompatibilitätsprobleme auch durch vorbereitende Zuweisungen gelöst werden.
- Bei Zuweisungen wird (wie bei den anderen Konvertierungen) zur Laufzeit überprüft, ob der Wert des Ausdrucks im Typ der Ergebnisvariable repräsentierbar ist.

```
oberon$ cat Crash.m2
MODULE Crash;

   VAR
      negative: INTEGER;
      card: CARDINAL;

BEGIN
   negative := -1;
   card := negative;
END Crash.
oberon$ m2c -o crash Crash.m2 && crash

runtime error in module Crash at line 10:
CARDINAL/INTEGER conversion error
Abort(coredump)
oberon$
```

Konvertierungen zu und von REAL

```
FOR n := 0 TO m DO
   x := xmin + FLOAT(n) * xdelta;
   SetFuncParam(function, varname, x);
   Plot(x, EvalFunction(function));
END;
```

- Mit **FLOAT** können Argumente, die zu **INTEGER** zuweisungskompatibel sind, in **REAL** konvertiert werden.
- Mit **TRUNC** kann ein Argument vom Typ **REAL** in **INTEGER** konvertiert werden. Dabei wird der 0 zu gerundet.

Typfunktionen

```
oberon$ cat TypeFunction.m2
MODULE TypeFunction;
   FROM InOut IMPORT WriteInt, WriteLn;
   VAR int: INTEGER; card: CARDINAL;
BEGIN
   card := MAX(CARDINAL); int := INTEGER(card);
   WriteInt(int, 1); WriteLn;
END TypeFunction.
oberon$ m2c -o typef TypeFunction.m2 && typef
-1
oberon$
```

- Typfunktionen sind beliebig mächtig, beliebig gefährlich und völlig unportabel.
- Trotzdem werden sie leider auch dann verwendet, wenn es sichere Alternativen geben würde.
- Jeder Typname kann als Funktionsname verwendet werden. Das Resultat hat dann genau den Typ des Typnamens.
- An den Typ des Arguments wird nur die Forderung gestellt, daß er exakt die gleiche Größe in Bytes benötigt.
- Die Konvertierung erfolgt ohne Überprüfungen und ohne Veränderungen der Repräsentierung.

Tips zur Vermeidung von Problemen

```
TYPE
   WorldSize = INTEGER [0..maxsize];
   WorldIndex = INTEGER [-1..maxsize];
      (* schliesst einen unsichtbaren Rand ein, der immer
         unbelegt ist (bei der Option border)
      *)
   World = ARRAY WorldIndex, WorldIndex OF CHAR;
```

- Versuchen Sie, alle ganzzahlige Typen, die miteinander in Operationen verknüpft werden, entweder gemeinsam dem **INTEGER**- oder **CARDINAL**-Lager zuzuführen.
- Bei Unterbereichstypen läßt sich die "Lagerzugehörigkeit" mit der Angabe des zugehörigen Basistyps erzwingen: Zum Beispiel durch die Angabe von **INTEGER** statt der Voreinstellung **CARDINAL**.
- Nutzen Sie auf elegante Weise die Zuweisungskompatibilität, um von dem einen Lager ins andere zu wechseln. Dies ist insbesondere attraktiv bei Funktionen: Der Typ der Funktion muß nur zuweisungskompatibel zum Ausdruck bei **RETURN** sein.

```
VAR lines: INTEGER;

PROCEDURE Lines() : CARDINAL;
BEGIN
   RETURN lines
END Lines;
```

Mengen in Modula-2

GameOfLife.m2

```
TYPE
   NeighbourCount = [0..8];
   NeighbourCountSet = SET OF NeighbourCount;
CONST
   lonely = NeighbourCountSet{0, 1};
   birth = NeighbourCountSet{3};
   overpop = NeighbourCountSet{4..8};

(* ... *)

neighbours := Neighbours(i, j);
IF neighbours IN (lonely + overpop) THEN
   newWorld[i, j] := space;
ELSIF neighbours IN birth THEN
   newWorld[i, j] := inhabitated;
ELSE
   newWorld[i, j] := world[i, j];
END;
```

- Mit **SET OF** *Type* kann ein Mengentyp deklariert werden, wobei *Type* ein (fast) beliebiger skalarer Typ sein darf.

- Mengenkonstanten werden durch den Typnamen, gefolgt von einer durch geschweifte Klammern eingeklammerte Aufzählung der Elemente, spezifiziert.

- Bei Aufzählungen sind Bereiche zulässig.

- Mit dem **IN**-Operator kann überprüft werden, ob ein Element in einer Menge liegt.

Mengenoperatoren

`a + b`	Vereinigung zweier Mengen.
`a - b`	Differenz: Die Menge aller Elemente aus *a* abzüglich aller Elemente aus *b*.
`a * b`	Die Schnittmenge aus *a* und *b*.
`a / b`	Symmetrische Differenz: Die Vereinigung abzüglich der Schnittmenge.
`a = b`	Test, ob beide Mengen identisch sind.
`a # b`	Test, ob beide Mengen ungleich sind.
`a <= b`	Test, ob *a* eine Teilmenge von *b* ist.
`a >= b`	Test, ob *b* eine Teilmenge von *a* ist.
`x IN a`	Test auf Mitgliedschaft, wobei *x* kompatibel zu dem Basistyp von *a* sein muß.
`INCL(a, x)`	Entspricht `a := a + {x}`.
`EXCL(a, x)`	Entspricht `a := a - {x}`.

Hinweis: Bei Mengentypen entsprechen Kompatibilität und Zuweisungskompatibilität der Identität. Die Kompatibilität der Basistypen überträgt sich also nicht auf Mengentypen, die davon abgeleitet werden.

Komplement

```
TYPE
   Colour = (red, green, blue, white, black);
   ColourSet = SET OF Colour;
CONST
   allcolours = ColourSet{MIN(Colour) .. MAX(Colour)};

PROCEDURE Complement(colours: ColourSet) : ColourSet;
BEGIN
   RETURN allcolours - colours
END Complement;
```

- In Modula-2 fehlt (im Gegensatz zu Oberon) ein Komplement-Operator. Der entsprechende Effekt läßt sich jedoch mit Hilfe einer Menge, die alle Elemente enthält, und der Differenz erreichen.

- Wie an diesem Beispiel zu sehen ist, ist es (wie sonst auch) sehr zu empfehlen, Basistypen auch dann einen Namen zu geben, wenn sie für Variablen nie verwendet werden. Hier ist der Name des Basistyps für die Verwendung von **MIN** und **MAX** nützlich.

Mengen und Bit-Operationen

```
PROCEDURE And(a, b: CARDINAL) : CARDINAL;
BEGIN
   RETURN CARDINAL(BITSET(a) * BITSET(b))
END And;

PROCEDURE Or(a, b: CARDINAL) : CARDINAL;
BEGIN
   RETURN CARDINAL(BITSET(a) + BITSET(b))
END Or;

PROCEDURE Xor(a, b: CARDINAL) : CARDINAL;
BEGIN
   RETURN CARDINAL(BITSET(a) / BITSET(b))
END Xor;
```

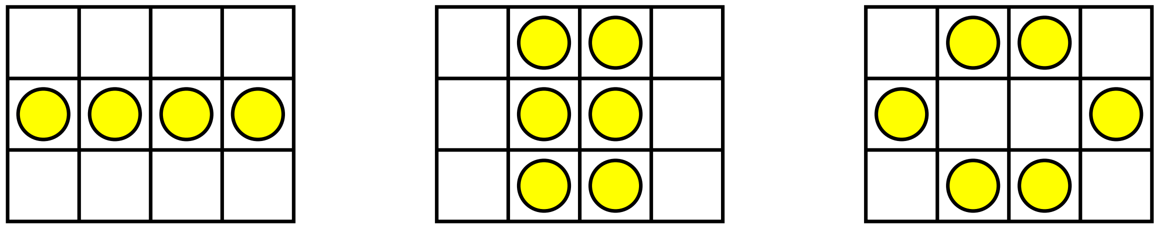
- Alle Mengen-Operationen entsprechen (in der Implementierung) Bit-Operationen. Das liegt daran, daß jedem Element in einer Menge genau ein Bit zugeordnet wird.

- Der Typ **BITSET** entspricht dem Typ `SET OF [0..31]`, falls **CARDINAL** aus 32 Bits besteht. In jedem Fall ist **BITSET** genauso groß wie **CARDINAL** oder **INTEGER**.

- Damit lassen sich sehr leicht Bit-Operationen für **CARDINAL** durchführen, wenngleich die unseligen Typfunktionen dabei verwendet werden müssen.

- Für Bit-Schiebe-Operationen empfiehlt sich die Verwendung von **DIV** und **MOD** unter Verwendung entsprechender Zweierpotenzen.

Aufgabe 14: Conway's Game of Life

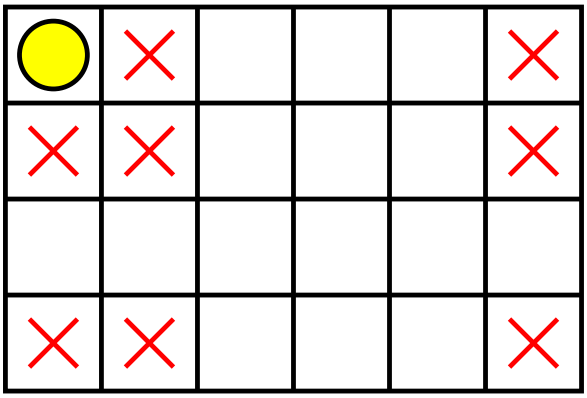


- Bei Conway's Game of Life handelt es sich um einen zellulären Automaten in der Ebene, bei der jedes Feld zwei Zustände besitzt: gefüllt (1) oder leer (0).

- 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.

Fehlerbehandlung

```
oberon$ live
Usage: live [-q] [-b] world
oberon$ perl -e 'print "X" x 1000' >long && live long
Die Eingabe ist zu gross!
oberon$ perl -e 'print "X\n" x 1000' >endless && live endless
Die Eingabe ist zu gross!
oberon$ live shooter3
Diese Welt ist zu gross fuer diesen Bildschirm!
oberon$
```

- Die Musterlösung akzeptiert alle Vorgaben, wenn sie noch auf den Bildschirm passen und das vorgegebene Limit der Seitenlängen einhalten.

- Alle Zeichen, die kein Leerzeichen sind, werden dabei als "lebend" betrachtet.

Programm-Rahmen

GameOfLife.m2

```
(*
   Allgemeine Informatik I / Programmieren I    WS 1998/1999
   Musterloesung fuer das Blatt 12, Aufgabe 14
   Ingo Melzer und Andreas Borchert
*)
MODULE GameOfLive;

   FROM Arguments IMPORT InitArgs, GetFlag, FetchString,
      AllArgs, Usage;
   FROM ASCII IMPORT nl;
   FROM FtdIO IMPORT FwriteString, FwriteLn;
   FROM MainWin IMPORT WriteString, Write, WriteLn, Read,
      Flush, SetPos, Clear, lines, columns;
   FROM StdIO IMPORT FILE, MODE, Fopen, Fgetc, stdin, stderr;
   FROM SysExit IMPORT Exit;
   FROM SysPerror IMPORT Perror;

   (* ... *)

BEGIN
   Clear; Flush;
   ProcessArgs;
   IF ReadWorld(input, world, noflines, nofcolumns) THEN
      continue := FALSE;
      REPEAT
         NextGeneration(world, noflines, nofcolumns,
            newWorld);
         WriteWorld(world, noflines, nofcolumns);
         world := newWorld;
      UNTIL query & ~Weiter();
   END;
   SetPos(lines-1, 0); Flush;
END GameOfLive.
```

Deklarationen

GameOfLife.m2

```
CONST
   maxsize = 80; (* maximale Seitenlaenge *)
   space = " "; (* unbesetzt *)
   inhabitated = "X"; (* besetzt *)
TYPE
   WorldSize = INTEGER [0..maxsize];
   WorldIndex = INTEGER [-1..maxsize];
   World = ARRAY WorldIndex, WorldIndex OF CHAR;

VAR
   world, newWorld: World;
   noflines, nofcolumns: WorldSize;
```

- Auch wenn die Welt einen Rand besitzt, werden auf die Nachbarschaftsfelder zugegriffen. Damit dies ohne Probleme gelingt, wurde der Indexbereich von `World` vergrößert. Allerdings bleibt der Rand immer unbesetzt.

- `noflines` und `nofcolumns` geben die Zahl der genutzten Zeilen und Spalten an, die jeweils das Minimum von `maxsize` und der jeweiligen Bildschirmgröße sind.

Argumentverarbeitung

GameOfLife.m2

```
VAR
   (* Kommandozeilenargumente *)
   query: BOOLEAN; (* mit Rueckfragen? *)
   border: BOOLEAN; (* Torus oder mit Rand? *)
   input: FILE;
      (* von hier ist die Ausgangssituation einzulesen *)

PROCEDURE ProcessArgs;
   VAR
      flag: CHAR;
      filename: ARRAY [0..511] OF CHAR;
BEGIN
   InitArgs("[-q] [-b] world");
   query := FALSE; border := FALSE;
   WHILE GetFlag(flag) DO
      CASE flag OF
      | "q":   query := TRUE;
      | "b":   border := TRUE;
      ELSE
         Usage;
      END;
   END;
   FetchString(filename);
   IF ~Fopen(input, filename, read,
         (* buffered = *) TRUE) THEN
      Perror(filename); Exit(1);
   END;
   AllArgs;
END ProcessArgs;
```

Einlesen der Ausgangssituation

GameOfLife.m2

```
PROCEDURE ReadWorld(input: FILE; VAR world: World;
                    VAR noflines,
                    nofcolumns: WorldSize) : BOOLEAN;
   VAR
      line, column: WorldSize;
      ch: CHAR;
BEGIN
   InitWorld(world);
   line := 0; column := 0;
   WHILE Fgetc(ch, input) DO
      IF ch = nl THEN
         INC(line); column := 0;
      ELSE
         IF (line = maxsize) OR (column = maxsize) THEN
            WriteString("Die Eingabe ist zu gross!");
            WriteLn; RETURN FALSE
         END;
         world[line, column] := ch; INC(column);
      END;
   END;
   IF (ORD(column) >= columns) OR (ORD(line) >= lines) THEN
      WriteString("Diese Welt ist zu gross fuer ...!");
      WriteLn;
      RETURN FALSE
   END;
   nofcolumns := maxsize; noflines := maxsize;
   IF ORD(nofcolumns) > columns THEN
      nofcolumns := columns
   END;
   (* die letzte Zeile wird fuer die Eingabe reserviert *)
   IF ORD(noflines) > lines-1 THEN noflines := lines-1 END;
   RETURN TRUE
END ReadWorld;
```

Einlesen der Ausgangssituation

GameOfLife.m2

```
PROCEDURE InitWorld(VAR world: World);
   (* initialisiert die Welt fuer die Eingabe *)
   VAR i, j: WorldIndex;
BEGIN
   FOR i := MIN(WorldIndex) TO MAX(WorldIndex) DO
      FOR j := MIN(WorldIndex) TO MAX(WorldIndex) DO
         world[i, j] := space;
      END;
   END;
END InitWorld;
```

- Da beim Einlesen nur die Felder initialisiert werden, die in der Eingabe ihre Repräsentierung finden, ist es notwendig zuvor die gesamte Matrix zu initialisieren.

- Um flexibel gegenüber Programmänderungen zu sein, empfiehlt es sich bei solchen **FOR**-Schleifen immer, einen entsprechenden Bereichstyp zur Hand zu haben, um die **MIN**- und **MAX**-Funktionen dafür auszunutzen.

Die nächste Generation

GameOfLife.m2

```
PROCEDURE NextGeneration(world: World;
                         noflines, nofcolumns: WorldSize;
                         VAR newWorld: World);
   VAR
      i, j: WorldIndex;
      neighbours: NeighbourCount;

BEGIN (* NextGeneration *)
   IF border THEN
      (* den nicht sichtbaren Rand initialisieren *)
      FOR i := -1 TO noflines DO
         world[i, -1] := space;
         world[i, nofcolumns] := space;
      END;
      FOR j := -1 TO nofcolumns DO
         world[-1, j] := space; world[noflines, j] := space;
      END;
   END;
   FOR i := 0 TO noflines -1 DO
      FOR j := 0 TO nofcolumns -1 DO
         neighbours := Neighbours(i, j);
         IF neighbours IN (lonely + overpop) THEN
            newWorld[i, j] := space;
         ELSIF neighbours IN birth THEN
            newWorld[i, j] := inhabitated;
         ELSE
            newWorld[i, j] := world[i, j];
         END;
      END;
   END;
END NextGeneration;
```

Die nächste Generation

GameOfLife.m2

```
PROCEDURE Neighbours (l, c: WorldIndex) : NeighbourCount;
   (* Zaehlt die Anzahl der belegten Felder in
      der Nachbarschaft von world[l, c]
   *)
   VAR
      neighbours: NeighbourCount;
      i, j: [-1..1];
      nl, nc: WorldIndex;
BEGIN
   neighbours := 0;
   FOR i := -1 TO 1 DO
      FOR j := -1 TO 1 DO
         IF (i # 0) OR (j # 0) THEN
            (* eigenes Feld ignorieren *)
            IF border THEN
               nl := l + i; nc := c + j;
            ELSE (* Torus *)
               nl := (noflines + l + i) MOD noflines;
               nc := (nofcolumns + c + j) MOD nofcolumns;
            END;
            IF world[nl, nc] # space THEN
               INC(neighbours);
            END;
         END;
      END;
   END;
   RETURN neighbours
END Neighbours;
```

Interaktion

GameOfLife.m2

```
PROCEDURE Weiter() : BOOLEAN;
   (* Implementierung der Option -q *)
   VAR
      answer: CHAR;
BEGIN
   IF continue 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
      ELSE
         RETURN TRUE
      END;
   END;
END Weiter;
```

Ausgabe

GameOfLife.m2

```
PROCEDURE WriteWorld(world: World; height, width: WorldSize);
   (* Ausgabe von world auf stdout *)
   VAR
      line, column: WorldIndex;
BEGIN
   FOR line := 0 TO height-1 DO
      SetPos(line, 0);
      FOR column := 0 TO width-1 DO
         Write(world[line, column]);
      END;
   END;
   Flush;
END WriteWorld;
```

- Bei dieser Variante wurde auf *WriteLn* verzichtet und stattdessen *SetPos* in die äußere Schleife gezogen.

- Das Problem liegt darin, daß beim Schreiben eines Zeichens in der 80. Spalte bereits implizit ein Zeilenvorschub vorgenommen wird und ein Aufruf von *WriteLn* dann zu einem weiteren Zeilenvorschub führen würde.

- So können jetzt auf sichere Weise 80 Spalten genutzt werden und nicht nur 79.

Tips zur 15. Aufgabe

World.d

```
DEFINITION MODULE World;

   TYPE Mode = (border, torus);

   PROCEDURE GetMaximumSize(): CARDINAL;

   PROCEDURE Init(columns, lines: CARDINAL; mode: Mode);

   PROCEDURE Lines(): CARDINAL;
   PROCEDURE Columns(): CARDINAL;
   PROCEDURE GetMode(): Mode;

   PROCEDURE Get(line, column: CARDINAL): CHAR;
   PROCEDURE Set(line, column: CARDINAL; char: CHAR);

   PROCEDURE NextGeneration;

END World.
```

- Die Lösung der 14. Aufgabe ist so zu modularisieren, daß die Datenstruktur des Spielfeldes unabhängig vom Rest des Programmes wird.

- Hierfür ist *World.d* genau in der gezeigten Form zu verwenden.

Die Schnittstelle

- *GetMaximumSize* liefert die maximale Seitenlänge zurück, die bei *lines* oder *columns* an *Init* übergeben werden darf.

- Mit *Init* wird ein Spielfeld der gegebenen Größe und Form initialisiert. Alle anderen Prozeduren (außer *GetMaximumSize* dürfen erst nach *Init* aufgerufen werden.

- Mit *Lines*, *Columns* und *GetMode* können die bei *Init* übergebenen Parameter abgefragt werden.

- *Get* und *Set* erlauben das Abfragen und Setzen einzelner Felder. Zulässig sind dabei die Indexbereiche `[0..Lines()-1]` (für Zeilen) und `[0..Columns()-1]` (für Spalten).

- Wie bisher stehen Leerzeichen für unbelegte Felder und alle andere Zeichen repräsentieren belegte Felder.

- *NextGeneration* führt das Spielfeld in die nächste Generation über.

Modularisiertes Beispiel

Maze.d

```
DEFINITION MODULE Maze;

   CONST
      space = " ";  (* freier Raum *)
      walk = "*";   (* Wegemarkierung *)

   PROCEDURE MinimalSize() : CARDINAL;
   PROCEDURE MaximalSize() : CARDINAL;

   PROCEDURE Init(lines, columns: CARDINAL);

   PROCEDURE Lines() : CARDINAL;
   PROCEDURE Columns() : CARDINAL;

   PROCEDURE Get(line, column: CARDINAL; VAR ch: CHAR);
   PROCEDURE Set(line, column: CARDINAL; ch: CHAR);

END Maze.
```

- Um ein funktionierendes Beispiel für die Modularisierung zu sehen, steht eine modularisierte Fassung der Musterlösung zur 13. Aufgabe zur Verfügung (Durchlaufen eines Labyrinths).

- Die Modularisierung ist bei diesem Beispiel noch weiter vorangeschritten: Auch die Ein- und Ausgabe und das Finden eines Weges wurden herausgetrennt.