$movecursor$: $squareaddrT \times \{UP, DOWN, RIGHT, LEFT\} \rightarrow squareaddrT$
$movecursor(x, y) = z$ where

Condition	$z =$
$y = UP \wedge \langle x.row - 1, x.col \rangle \in squareaddrT$	$\langle x.row - 1, x.col \rangle$
$y = DOWN \wedge \langle x.row + 1, x.col \rangle \in squareaddrT$	$\langle x.row + 1, x.col \rangle$
$y = LEFT \wedge \langle x.row, x.col - 1 \rangle \in squareaddrT$	$\langle x.row, x.col - 1 \rangle$
$y = RIGHT \wedge \langle x.row, x.col + 1 \rangle \in squareaddrT$	$\langle x.row, x.col + 1 \rangle$
$true$	$\langle x.row, x.col \rangle$

win: $boardT \rightarrow boolean$

$win(x)$ = all squares that contain mines are covered or marked $\wedge$
all squares that do not contain mines are uncovered

$lose$: $boardT \rightarrow boolean$

$lose(x)$ = a square that contains a mine is uncovered $\wedge$
a square that does not contain a mine is covered or marked

A.1.7 Expected changes

1. Allow the user to vary the size of the minefield and the number of mines in the minefield.
2. Add command line arguments to the invocation which would set the initial number of mines and grid size.
3. Allow the user to save a game while in progress and restore it later.
4. Add a timer which will monitor how long it takes a player to complete the game.
5. Add extra "lives" by allowing more than one mine to be uncovered before ending the game.

A.1.6 Functions

randomlocns: *integer* → *set of squareaddrT*
randomlocns$(x) = y$ where

$| y | = x \wedge$
each element of y is randomly selected from *squareaddrT*, all selection outcomes are equally likely

seedmines: *set of squareaddrT* → *boardT*
seedmines$(x) = y$ where

$(\forall \langle r,c\rangle \in squareaddrT)(y[r,c].mask = COVERED) \wedge$
$(\forall \langle r,c\rangle \in x)(y[r,c].hidden = MINE) \wedge$
$(\forall \langle r,c\rangle \notin x)(y[r,c].hidden = 0)$

minecounts: *boardT* → *boardT*
minecounts$(x) = y$ where

$(\forall \langle r,c\rangle \in squareaddrT \wedge x[r,c].hidden \neq MINE)$
$(y[r,c].hidden =$ the number of mines in $neighbourhood(\langle r,c\rangle))$

adjacent: *squareaddrT* × *squareaddrT* → *boolean*
adjacent$(x,y) = z$ where

Condition	$z =$
$\langle x.row\text{-}1, x.col\rangle = y$	*true*
$\langle x.row\text{-}1, x.col\text{+}1\rangle = y$	*true*
$\langle x.row, x.col\text{+}1\rangle = y$	*true*
$\langle x.row\text{+}1, x.col\text{+}1\rangle = y$	*true*
$\langle x.row\text{+}1, x.col\rangle = y$	*true*
$\langle x.row\text{+}1, x.col\text{-}1\rangle = y$	*true*
$\langle x.row, x.col\text{-}1\rangle = y$	*true*
$\langle x.row\text{-}1, x.col\text{-}1\rangle = y$	*true*
true	*false*

neignbourhood: *squareaddrT* → *set of squareaddrT*
neighbourhood$(x) = \{y \in squareaddrT \mid adjacent(x,y)\}$

frontier: *squareaddrT* → *set of squareaddrT*
frontier$(x) =$

$(board[x.row, x.col].hidden \neq 0 \Rightarrow \{\} \mid$
$true \Rightarrow \{y \in neighbourhood(x) \mid y.mask \in \{COVERED,\ MARKED\}\})$

when $c = MARK$
    if $board[cursquare.row, cursquare.col].mask \neq UNCOVERED$ then
        $board[cursquare.row, cursquare.col].mask := MARKED$
    else $MSG := MARKERRORMSG$
when $c = UNMARK$
    if $board[cursquare.row, cursquare.col].mask = MARKED$ then
        $board[cursquare.row, cursquare.col].mask := COVERED$
    else $MSG := UNMARKERRORMSG$
when $c \notin commandT$
    $MSG := INVALIDKEYMSG$

A.1.4 Constants

Name	Value
NUMROWS	10
NUMCOLS	10
NUMMINES	8
DEFEATMSG	"Game Over! You Lose!"
VICTORYMSG	"Game Over! You Win!"
INVALIDKEYMSG	"Invalid keystroke."
MARKERRORMSG	"Unable to mark square."
UNMARKERRORMSG	"Unable to unmark square."
MINE	'*'
LEFT	'h'
RIGHT	'l'
UP	'k'
DOWN	'j'
UNCOVER	'u'
MARK	'm'
UNMARK	'w'
QUIT	'q'
CONCEALED	'X'
TAGGED	'M'

A.1.5 Types

$hiddenT = [0..8] \cup \{MINE\}$
$squareT = tuple(mask: \{UNCOVERED, COVERED, MARKED\}$
$\qquad hidden: hiddenT)$
$squareaddrT = tuple\ (row:[0..NUMROWS\text{-}1], col:[0..NUMCOLS\text{-}1])$
$commandT = \{LEFT, RIGHT, UP, DOWN, QUIT, UNCOVER, MARK, UNMARK\}$
$boardT = sequence[NUMROWS,NUMCOLS]\ of\ squareT$

Figure A.2: Game FSM

Inputs
Commands from *stdin*.
Outputs
All outputs are to *scn* and its fields.
States
board:*boardT*
cursquare:*squaraddrT*
minelocns: *set of squareaddrT*
Initial state
$minelocns := randomlocns(NUMMINES)$
$board := seedmines(randomlocns)$
$board := minecounts(board)$
$cursquare := \langle 0, 0 \rangle$
Transitions and outputs
$MSG :=$ ""
if $win(board)$ then
$OUTCOMEMSG := VICTORYMSG$
else
$OUTCOMEMSG :=$ ""
For each command input, c, from *stdin*
modify state as detailed in the Command semantics subsection.

Notes on screen updating:
1. Initially and after each keystroke:
($\forall \langle r, c \rangle \in squareaddrT$)
(if the square is uncovered then $G[r,c] = board[r,c].hidden$
else if $board[r,c].mask = MARKED$ then $G[r,c] = TAGGED$
else $G[r,c] = CONCEALED$)
2. $G[cursquare.row,cursquare.col]$ is always highlighted.
3. Initially and after each keystroke, the cursor is positioned at *scn*[0,0]

We divide *scn*, a 24-by-80 array, into parts, as shown in Figure A.1. The non-blank areas are divided into *screen fields*, either *fixed* or *varying*. The fixed fields are written when MINESWEEP execution begins and remain unchanged while MINESWEEP is running. The varying fields may change repeatedly during MINESWEEP execution. There are three varying fields referenced by the identifiers G, MSG and $OUTCOMEMSG$. The extent of each varying field on the screen is the character positions occupied by the field identifier, and the trailing -'s if present. When the value of a varying field is shorter than the extent shown, the field is left-justified and padded right with blanks. The G fields are referenced by row and column subscripts numbered zero relative top-down and left-to-right. For example, the G field $G[0,NUMCOLS\text{-}1]$ is in the top right corner. One of the the G fields is always highlighted.

A.1.3 State machine

Command line invocation

MINESWEEP is invoked by typing

```
msweep
```

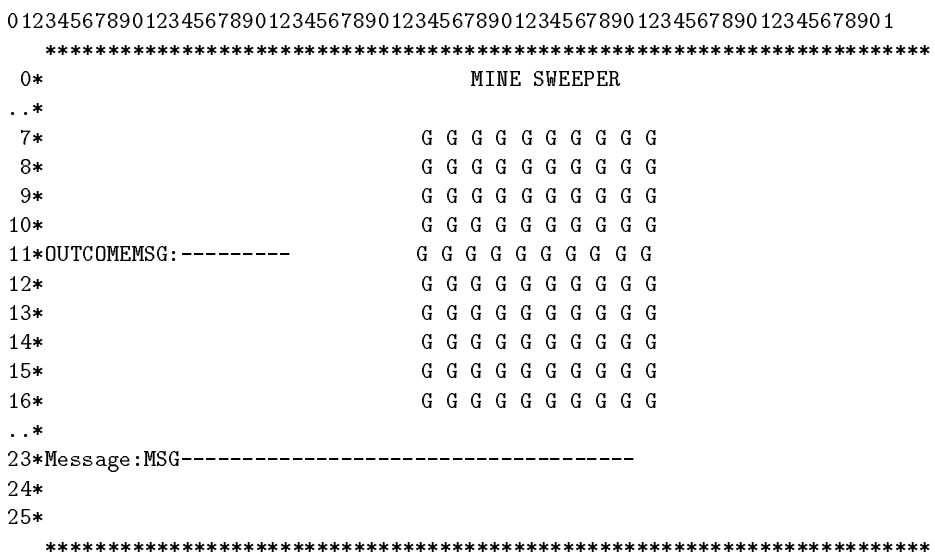
on the command line.

Figure A.2 shows the FSM that details MINESWEEP behaviour. The function *seedmines* stores mines in *board* squares determined by *randomlocns*. The function *minecount* initializes the *board* squares that do not contain mines.

Command semantics

$MSG :=$ ""
when $c = QUIT$
    clear *scn*
    halt MINESWEEP
when $c \in \{UP, DOWN, RIGHT, LEFT\}$
    $cursquare := movecursor(cursquare,c)$
when $c = UNCOVER$
    $board[cursquare.row, cursquare.col].mask := UNCOVERED$
    $y := frontier(cursquare)$
    while $y \neq \{\}$ do
        $x :=$ an element from y
        $y := y$ - $\{x\}$
        $board[x.row,x.col].mask := UNCOVERED$
        $y := y \cup frontier(x)$
    if $win(board)$ then
        $OUTCOMEMSG := VICTORYMSG$
    else if $lose(board)$ then
        $OUTCOMEMSG := DEFEATMSG$

Figure A.1: Screen format



Document overview

Section 2 declares the environment variables. Section 3 describes how to invoke MINESWEEP from the Unix Shell and contains the finite state machine that specifies how the MINESWEEP game is generated and played. Sections 4, 5 and 6 describe constants, type definitions and functions respectively. Section 7 lists expected changes.

A.1.2 Environment variables

Input variables

stdin : *string*
: Unix standard input

Output variables

scn : *sequence* $[24, 80]$ *of char*
: $scn[r, c]$ is the character at screen row r and column c,
with numbering zero-relative and beginning at the upper-left corner.

Appendix A

Requirements Specification

A.1 MINESWEEP Requirements Specification

A.1.1 Overview

System overview

MINESWEEP is a simplified version of the popular Windows based game called minesweep. Users are presented with a two-dimensional grid of squares. Some of the squares on this grid contain mines and some do not. The player must determine the locations of the mines. This is accomplished by selectively "uncovering" squares. If a mine is uncovered then the player loses. Otherwise, a number appears which indicates the number of adjacent mines. Player victory occurs when all squares without mines are uncovered.

Hardware and software environment

MINESWEEP runs on Sun/3 and Sun/4 workstations running SunOS. It is implemented in the C programming language and requires the standard Unix/C programming libraries. The curses function library is required to perform output to the terminal screen and the stdlib library is required to facilitate random number generation.

Notation

All identifiers are shown in *italics*. The names of constants and abbreviations are all upper case. The others are all lower case, except for types, whose names end in '*T*'.

Part I

Appendix