Background

Sokoban is a well-known puzzle video game that was first introduced in Japan in 1980. The word *Sokoban* means "warehouse keeper" in Japanese which refers to the main character of the game. In this game, the player controls the warehouse keeper (sokoban) who is placed in a closed warehouse with a number of boxes that need to be put in some predefined goal locations. The keeper can walk around the warehouse and push boxes around in order to get them into goal positions. The goal of this game is to put all boxes into goal positions in the fewest number of moves. The initial states of two instances of this game are shown in Figure 1 below.

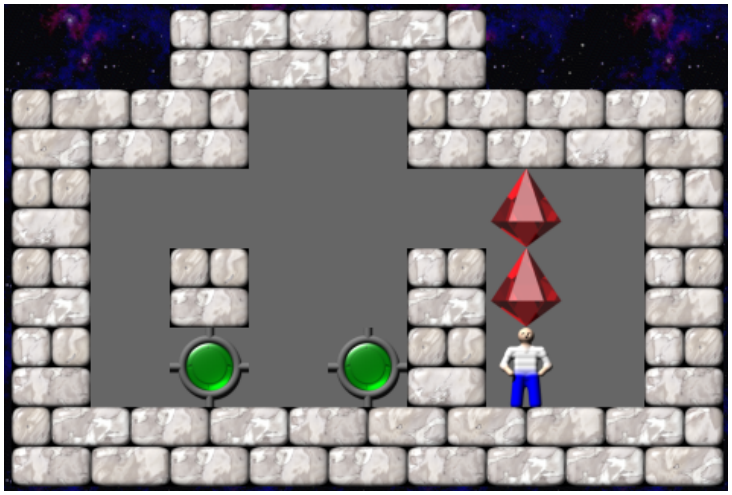

Figure 1: A Sokoban game. Boxes are presented by gems and goals are represented by circles.

How to play

The whole game (the warehouse) is on a grid. In each step, the keeper can move in any of the four basic directions (up, down, left, right). The keeper cannot walk into a wall or into a box. However, the keeper can push a box if the square next to the box (in the pushing direction) is either a goal or an empty square. Only one box can be pushed in any given step. In the case of multiple goals, there is no specific goal that a box has to be in. Boxes can be placed in goals in any order. A box can also be pushed out of a goal if needed (to make way for other moves). The game ends as soon as every box is in a goal position (even if there are more goals than boxes). In general, the minimum number of moves is not strictly required by the actual Sokoban game, because even finding a solution to the problem is already hard for a human player. Nevertheless, it is an objective of this assignment.

Sokoban as a search problem

In this assignment, we will turn this game into a search problem. As you have seen from class, a number of components are required for defining a search problem. In this case, they are

1) A state representation
2) A successor function that indicates which states can be reached from a given state in one step
3) A goal test
4) A cost function

Once all these components are defined (in practice as functions), the problem can be solved by a generic search engine that implements any search algorithm (e.g., DFS, BFS, A*, etc). Of course, the efficiency of the program and the quality of the solution depends on the type of search algorithm used.

Because of the time constraint, in this assignment, we will provide you with a search space formulation. You will then write some LISP functions to implement the above components based on the given formulation. Also, we will assume a uniform cost function. Every move in this game will have cost 1.

Game state representation

Each state of the game will be represented by a list of lists. The content of each square in the game will be represented by an integer. Each row of the game can then be represented by a list of integers that represent the content of each square in the row (left to right). Finally, a state is simply a list of rows (top to bottom). For simplicity, in this assignment we will assume that every row contains the same number of columns. We will assume that each state contains **exactly** one keeper, at least one box, and that the number of goals is greater than or equal to the number of boxes in the state.

Let us now introduce the representation of square contents. Each square in Sokoban can only contain so many things. A square may contain one of the following:

- Nothing (empty floor)
- A wall
- A box
- The keeper
- A goal
- A box on top of a goal
- The keeper on top of a goal.

Table 1 shows a mapping between square contents and integers that we will use to represent them. An ASCII representation of each type of content is also shown in this table. This will be discussed later.

Table 1: Mapping of square contents.

Content	Integer	ASCII
Blank	0	‘ ‘ (white space)
Wall	1	‘#’
Box	2	‘$’
Keeper	3	‘@’
Goal	4	‘.’
Box + goal	5	‘*’
Keeper + goal	6	‘+’

With this convention, we can represent the game state in Figure 1 in LISP (and assign it to variable p1) as:

```
(setq  p1  '((0 0 1 1 1 1 0 0 0)
             (1 1 1 0 0 1 1 1 1)
             (1 0 0 0 0 0 2 0 1)
             (1 0 1 0 0 1 2 0 1)
             (1 0 4 0 4 1 3 0 1)
             (1 1 1 1 1 1 1 1 1)) .
```

What is provided to you

For this assignment, we have provided you with an A* search engine in the file "a-star.lsp". You do not need to understand the code in this file. You will call the top level A* function defined in it. To call A*, after loading the file, type at the LISP prompt

(a* start-state #'goal-test #'next-states #'heuristic).

Notice that goal-test, next-states, and heuristic have #' in front of them. In LISP, you must put #', called "sharp-quote", in front of functions that you pass as arguments to other functions. You will need to define these functions in your submission. 'heuristic' must be substituted by the name of the heuristic function you want to use.

The above call to A* will return a solution of the search problem defined by the initial state and functions in the argument. The solution is returned in the form of a list of states along the solution path. So, if the solution returned takes 10 moves, the list returned will contain 11 elements.

In addition to "a-star.lsp", we have also provided you with some helper functions that you may need for completing the assignment. These functions are defined in "hw3.lsp".

What you have to do

1) Write a function called **goal-test** that takes a state as the argument and returns true (t) if and only if the state is a goal state of the game. A state is a goal state if it satisfies the game terminating condition described in "How to play" section.

2) Write a function called **next-states** that takes a state as the argument and returns the list of all states that can be reached from the given state in one move. This function may require several helper functions. It will be explained in more details in the next section. Your score for this component will depend solely on the correctness. Efficiency will not be evaluated (each function call should never take more than a second anyway. Otherwise there might be a problem).

3) Write a heuristic function called **h0** that returns the constant 0. This is a trivial admissible heuristic.

4) Write a heuristic function called **h1** that returns the number of boxes which are not on goal positions in the given state. Is this heuristic admissible? (Put your answer in the header comment of the function)

5) Write an admissible heuristic to make the A* search engine perform better. Name this function **hUID**, where UID is your student ID (for example, **h123456789** if 123456789 is your UID). This heuristic function takes in a state and returns an integer (>= 0). This function will be evaluated based on its relative performance when it is integrated by our implementation of the search space. The entry of each student will enter the competition against submissions by the rest of the class. The results will be based on the running time of different heuristics (minimizing number of expanded nodes is not enough). Top performers will be awarded with the full score and the worst entry will receive little or no credit for this part.

The remaining 5 percent of the score will be based on comments. All we look for is a brief description of and the logic behind each function you write.

Generating next states

Given a state of the game, each next state can be generated by moving the keeper in one of the valid directions. According to this formulation, each state can have at most four children. The following is one possible strategy for generating next states. You **do not** need to follow this, as long as your next-states function works as specified.

Implementation strategy

1) Write a function called **get-square** that takes in a state S, a row number r, and a column number c. It should return the integer content of state S at square (r,c). If the square is outside the scope of the problem, return the value of a wall.

2) Write a function called **set-square** that takes in a state S, a row number r, a column number c, and a square content v (integer). This function should return a new state S' that is obtained by setting the square (r,c) to value v. This function should not modify the input state.

(For the above two functions, how to index rows and columns is totally internal to your program. However, one reasonable scheme would be to index the top left corner of the game with (0,0) and increase the row and column indices as you move down and move to the right respectively.)

3) Write a function **try-move** that takes in a state S and a move direction D. This function should return the state that is the result of moving the keeper in state S in direction D. NIL should be returned if the move is invalid (e.g., there is a wall in that direction). How you represent a move direction is up to you. Remember to update the content of every square to the right value. Refer to Table 1 for the values of different types of square.

You will probably need several small helper functions to make your job easier. In any case, the high-level plan is that your **next-states** function can call **try-move** in each of four directions from the current position of the keeper (some of them will return NIL) and collect all returned values into a list. Helper functions are provided for identifying the keeper square and for removing NILs from a list.

The function **try-move** has to check whether the move is legal. If it is, it will need to update some squares of the current state to reflect the changes. Use **set-square** to help you do this. According to this strategy, *at most* three squares need to be updated for any single move (make sure you agree).

More information about each helper function can be found in the comment in "hw3.lsp".

Visualizing solutions

To make your experience more enjoyable, we have provided in "hw3.lsp" a printing function that will print out a state using the ASCII mapping in Table 1. In particular, you might find it useful to be able to visualize the solution returned by A*. To do so, type at the LISP prompt

(printstates (a* start-state #'goal-test #'next-states #'heuristic) 0.2).

The last argument of printstates is the delay (in seconds) between successive state display. For example, calling (printstates (a* start-state #'goal-test #'next-states #'heuristic) 0.2) with start-state set to p1 (defined above) results in something like Figure 2 (only the solution visualization part). An individual state can also be displayed using the printstate function. This might be useful for visualizing initial states.

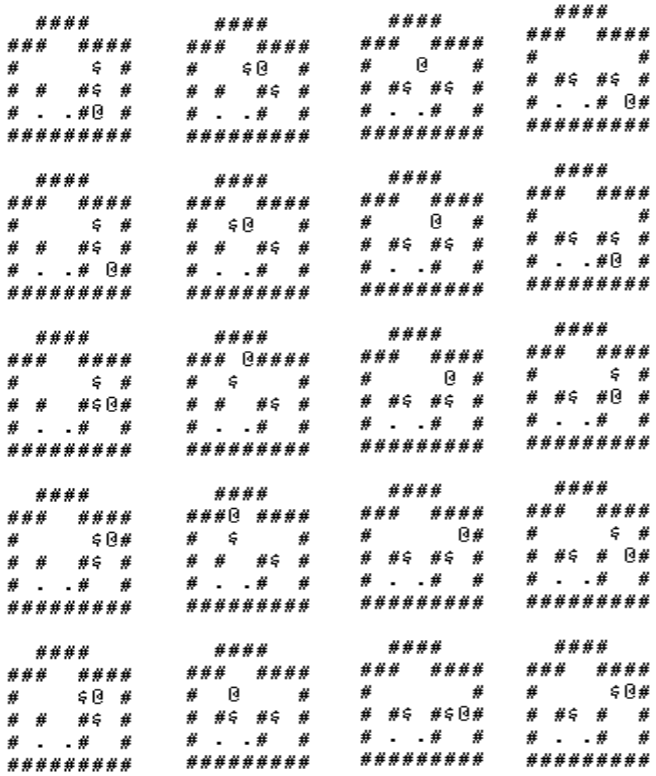

Figure 2: The first 20 states of an optimal solution of p1. States are ordered top to bottom and then left to right.

Sample outputs of next-states

In this section, we provide some sample outputs of the function **next-states.** Using printstates or printstate to visualize these inputs/outputs may be useful.

Example 1

```
(setq s1 '( (1 1 1 1 1)
            (1 0 0 4 1)
            (1 0 2 0 1)
            (1 0 3 0 1)
            (1 0 0 0 1)
            (1 1 1 1 1)
            ))
```

(next-states s1) should return

(((1 1 1 1 1) (1 0 2 4 1) (1 0 3 0 1) (1 0 0 0 1) (1 0 0 0 1) (1 1 1 1 1))
((1 1 1 1 1) (1 0 0 4 1) (1 0 2 0 1) (1 0 0 3 1) (1 0 0 0 1) (1 1 1 1 1))
((1 1 1 1 1) (1 0 0 4 1) (1 0 2 0 1) (1 0 0 0 1) (1 0 3 0 1) (1 1 1 1 1))
((1 1 1 1 1) (1 0 0 4 1) (1 0 2 0 1) (1 3 0 0 1) (1 0 0 0 1) (1 1 1 1 1))).

All four moves are possible in this case. Note how the upward move results in box pushing.

Example 2

```
(setq s2 '((1 1 1 1 1)
           (1 0 0 4 1)
           (1 0 2 3 1)
           (1 0 0 0 1)
           (1 0 0 0 1)
           (1 1 1 1 1)
           ))
```

(next-states s2) should return

```
(((1 1 1 1 1) (1 0 0 6 1) (1 0 2 0 1) (1 0 0 0 1) (1 0 0 0 1) (1 1 1 1 1))
 ((1 1 1 1 1) (1 0 0 4 1) (1 0 2 0 1) (1 0 0 3 1) (1 0 0 0 1) (1 1 1 1 1))
 ((1 1 1 1 1) (1 0 0 4 1) (1 2 3 0 1) (1 0 0 0 1) (1 0 0 0 1) (1 1 1 1 1))).
```

Only three moves are possible in this case: up, left, down. Note that the upward move puts the keeper on a goal square (keeper+goal is represented by the number 6, according to Table 1).

Example 3

```
(setq s3 '((1 1 1 1 1)
           (1 0 0 6 1)
           (1 0 2 0 1)
           (1 0 0 0 1)
           (1 0 0 0 1)
           (1 1 1 1 1)
           ))
```

(next-states s3) should return

```
(((1 1 1 1 1) (1 0 0 4 1) (1 0 2 3 1) (1 0 0 0 1) (1 0 0 0 1) (1 1 1 1 1))
 ((1 1 1 1 1) (1 0 3 4 1) (1 0 2 0 1) (1 0 0 0 1) (1 0 0 0 1) (1 1 1 1 1))).
```

Note how the keeper was on a goal square in s3. In this case, there are two possible moves: left and down. Both moves take the keeper out of the goal square.

Example 4

```
(setq s4 '((1 1 1 1 1)
           (1 4 2 0 1)
           (1 0 0 0 1)
           (1 0 0 0 1)
           (1 0 5 3 1)
           (1 1 1 1 1)
           ))
```

(next-states s4) should return

(((1 1 1 1 1) (1 4 2 0 1) (1 0 0 0 1) (1 0 0 3 1) (1 0 5 0 1) (1 1 1 1 1))

((1 1 1 1 1) (1 4 2 0 1) (1 0 0 0 1) (1 0 0 0 1) (1 2 6 0 1) (1 1 1 1 1))).

A box started in a goal state in s4. There are two possible moves in this case: up and left. Moving the keeper to the left pushes the box out of the goal state and lands the keeper on the goal state.