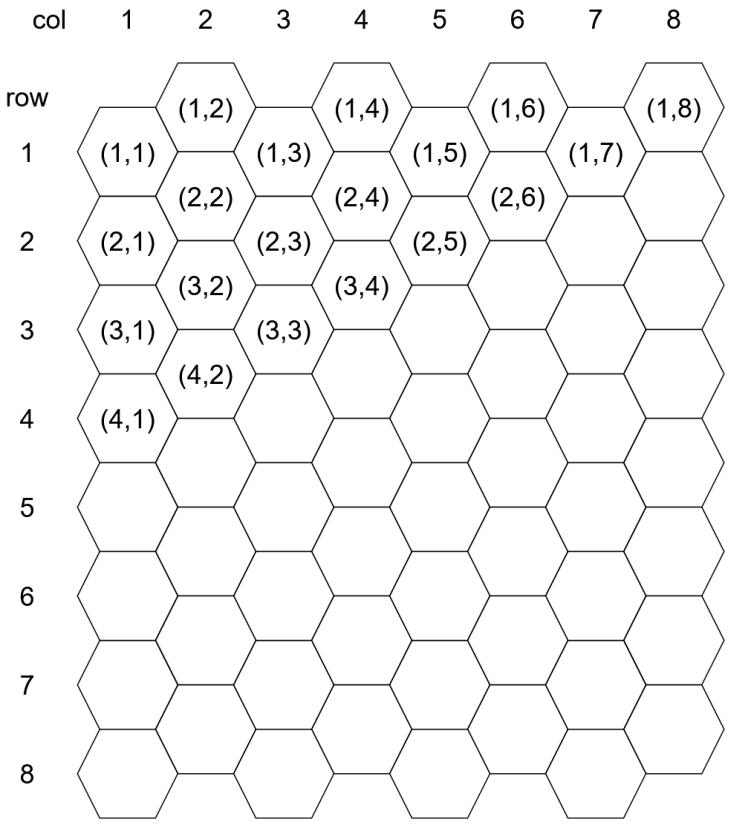
Kickstart Offense with Minion's Best in an Amicable Territory (KOMBAT)

As a leader of your tribe residing in a limited-area territory, you have trained an army of minions capable of performing a variety of martial arts. Scarcity of resources has caused your tribe to split into two factions, each equipped with its own army that you have trained. The other faction has appointed a leader of its own, ready to challenge the status quo with you. To ensure your faction's survival, your job is to strategize your remaining army to show that you deserve to remain a true leader of the tribe.

Overview of KOMBAT

KOMBAT is a turn-based game in which two players attempt to eliminate the other player's *minions* on the playing field. The playing field is an 8×8 field of hexagons (called *hexes* for short). The coordinate of a hex is specified by a pair of row and column numbers of the hex. Observe that hexes on the same row but adjacent columns will skew vertically, as shown in the following figure.



A *minion strategy* is a script that dictates how a minion will act in each turn. Before the game begins, both players agree on the strategy for each kind of minion. These strategies will be used until the end of the game and cannot be modified partway.

The game ends when all the minions belonging to one of the players have been eliminated; the other player is the winner. The game also ends when each player has played a specified number of turns. In the latter case, the winner is determined by the number of minions left in the territory, the HPs for the remaining minions, and the remaining budget.

Setup

First, the game should read the configuration file for necessary parameters.

Each game will work in one of three modes: duel, solitaire, and auto. In the duel mode, two players take turns playing; in the solitaire mode, one of the players becomes a bot, but the game proceeds identically otherwise. In the auto mode, both players become bots.

Before the start of the game, the players (excluding the bot) first agree on the number of kinds of minions that may be placed during the game. The game supports between 1 and 5 kinds of minions. The players should assign a name to each kind of minion. Each kind of minion has a *defense factor*, which will be used to reduce the amount of attack damage, and a strategy that will be executed in each turn. The players should agree on the defense factor and strategy for each kind of minion during setup. Minion strategies and defense factors cannot be altered once the game begins.

In the auto mode, humans still choose minion strategies and defense factors.

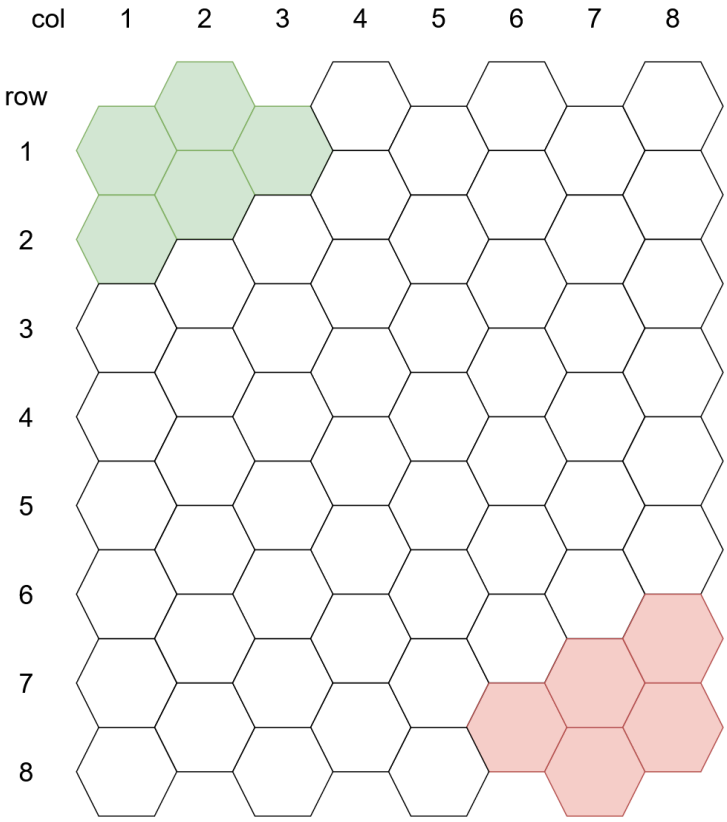
During setup, after agreeing on minion kinds, each player must spawn one minion of choice, for free. The Gameplay section describes valid spawn locations for each player.

Each player (including the bot) will be given an equal amount of initial budget as specified in the configuration file. Then, it is the first player's turn to start the game.

Gameplay

At the beginning of each turn, the current player (including the bot) is given additional budget as specified in the configuration file. Next, the overall budget accrues interest. If the resulting budget exceeds the maximum allowed budget as specified in the configuration file, the excess budget is forfeited. Then, the player will have an opportunity to spawn a new minion in a designated area. Initially, the designated area of the first player is the five hexes at the top-left corner of the territory; the designated area of the second player (or the bot) is the five hexes at the bottom-right corner of the territory, as illustrated in the following figure. The player can only spawn a new minion in an empty hex within the designated area. If all such hexes are occupied, the player loses this opportunity.

Each player can spawn new minions up to the number specified in the configuration file. Additional spawns are prohibited after this number is reached.



Additionally, before spawning a minion, the player has an opportunity to purchase a new hex so as to spawn new minions therein. This new hex must be adjacent to the existing hexes that can spawn new minions; attempts to purchase a hex elsewhere are not permitted.

The newly spawned minion will have an initial HP as specified in the configuration file. Purchasing a new spawnable hex and spawning a new minion incur costs as specified in the configuration file.

Once the player has finished spawning a new minion (if possible), all the minions belonging to the player will execute its own strategy, one minion at a time. The oldest minion executes its strategy first; the most recently spawned minion does so last.

After that, the other player's turn begins.

Summary of the order of actions in each turn

1. The player's budget increases by a set amount and accrues interest according to the calculated interest rates.
2. The player may choose to buy a new spawnable hex adjacent to the existing spawnable hexes.
3. The player may choose to spawn a new minion of a chosen kind on a spawnable hex.
4. Each minion, from oldest to newest, executes its own strategy.

Interest rates and calculation

An interest rate is specified as a percentage, which is always a positive number. Although budget queries in a minion strategy can only return integers, KOMBAT should keep track of the budget as a double, as decimals may affect interest calculations. If *m* is the current budget of a player, and *r* is the interest rate percentage, then the interest for the current turn is *m***r*/100 (floating-point arithmetic), which is then added to the budget. If the current budget of a player is less than 1, no interest is accrued.

If *b* is the base interest rate percentage as specified in the configuration file, *m* is the current budget of a player, and *t* is the current turn count of a player, the interest rate percentage *r* to be used is $b * \log_{10} m * \ln t$ (floating-point arithmetic).

Grammar for minion strategies

Minion strategies are governed by the following grammar. Terminal symbols are written in `monospace font`; nonterminal symbols are written in *italics*.

Strategy → *Statement*+
Statement → *Command* | *BlockStatement* | *IfStatement* | *WhileStatement*
Command → *AssignmentStatement* | *ActionCommand*
AssignmentStatement → `<identifier>` = *Expression*
ActionCommand → `done` | *MoveCommand* | *AttackCommand*
MoveCommand → `move` *Direction*
AttackCommand → `shoot` *Direction Expression*
Direction → `up` | `down` | `upleft` | `upright` | `downleft` | `downright`
BlockStatement → `{` *Statement** `}`
IfStatement → `if (` *Expression* `) then` *Statement* `else` *Statement*
WhileStatement → `while (` *Expression* `)` *Statement*
Expression → *Expression* + *Term* | *Expression* - *Term* | *Term*
Term → *Term* * *Factor* | *Term* / *Factor* | *Term* % *Factor* | *Factor*
Factor → *Power* ^ *Factor* | *Power*
Power → `<number>` | `<identifier>` | `(` *Expression* `)` | *InfoExpression*
InfoExpression → `ally` | `opponent` | `nearby` *Direction*

- + means at least one
- * means zero or more
- `<number>` is any nonnegative integer literal that can be stored as Java's long data type.
- `<identifier>` is any string not a reserved word.
  Identifiers must start with a letter, followed by zero or more alphanumeric characters.

The following strings are reserved words and cannot be used as identifiers: `ally, done, down, downleft, downright, else, if, move, nearby, opponent, shoot, then, up, upleft, upright, while`

Minion strategy evaluation

Variables

Each variable has the initial value of 0. After each turn, a variable retains its value for the beginning of the next turn. For example, if a variable *x* has been assigned a value of 47 at turn *t*, then its value will remain at 47 at the beginning of turn *t*+1, and could be assigned to another value in the next round of evaluation.

Variables whose names start with a lowercase letter are local, i.e., they are never shared between minions. If two minions have strategies containing local variables of the same name, their values can be different. Assignments to a local variable *x* in one minion's strategy do not affect *x*'s value in another minion's strategy.

Variables whose names start with an uppercase letter are global for that player, i.e., they are shared among all minions belonging to the same player.

Special variables

KOMBAT designates many identifiers in minion strategies as special, read-only variables. An attempt to assign any of these variables results in a "no-op": it's like this assignment never exists. The special variables are as follows (note case sensitivity).

`row`

Whenever this variable is evaluated, the current row number of the minion is returned.

`col`

Whenever this variable is evaluated, the current column number of the minion is returned.

`Budget`

Whenever this variable is evaluated, the player's remaining budget is returned.

`Int`

Whenever this variable is evaluated, the interest percentage rate for the player's current budget, as calculated according to the specification, is returned.

`MaxBudget`

Whenever this variable is evaluated, the maximum allowable budget, as specified in the configuration file, is returned.

`SpawnsLeft`

Whenever this variable is evaluated, the remaining number of minions that can still be spawned by the player is returned.

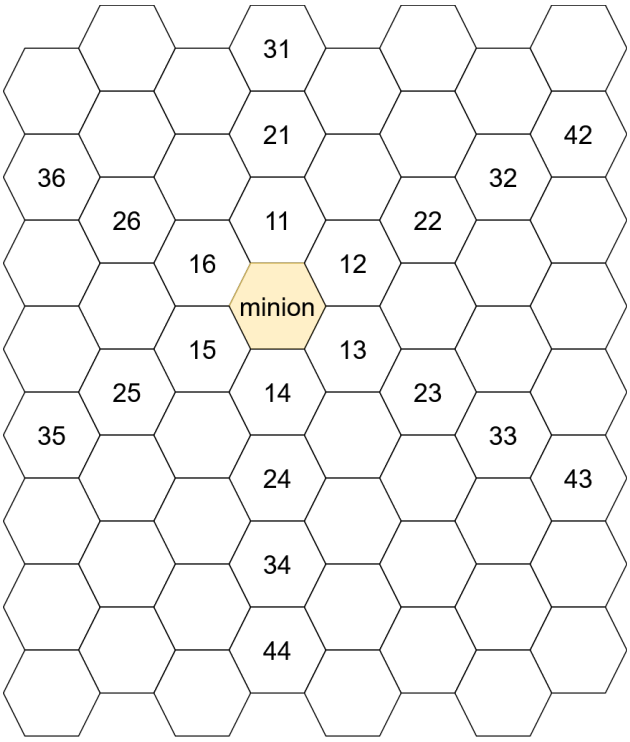
`random`

Whenever this variable is evaluated, a random value between 0 and 999 (inclusive) should be returned.

Information expressions

`opponent` expression

The `opponent` expression returns the location of the closest hex in one of the six directions from the minion that contains a minion belonging to the opponent. The following diagram shows a pattern of the expression's results for the locations to be returned. Hexes without a number are never reported even if occupied.



The value of each location is relative to the minion. The unit digit is between 1 and 6, and indicates the direction of the location. The remaining digits indicate the distance from the minion to that location.

If there are multiple opponent's hexes with the same minimal distance from the minion, return the location of the one with the lowest direction number. In other words, the returned value is the smallest possible one.

If there are no visible opponent's hexes whatsoever from the minion's current location, the `opponent` expression should return 0.

`ally` expression

The `ally` expression works in the same way as the `opponent` expression, except that it returns the information about the closest hex containing a minion belonging to the same player.

`nearby` function

The `nearby` function looks for a minion in a given direction that is closest to the current location of a minion. If the discovered minion belongs to the opponent, this function should return $100x+10y+z$, where x is the number of digits in the HP of the discovered minion, y is the number of digits in the defense factor for that minion, and z is the distance from the minion to that opponent as shown in the diagram above. This can be helpful when planning attacks.

If the discovered minion belongs to the same player, this function should return the negative of the value calculated as aforementioned.

If there is no minion in the given direction, the `nearby` function should return 0.

Action commands

During each turn, a number of action commands can be executed, although certain kinds of commands may be executed at most once per turn. After executing such commands, the turn ends automatically.

`done` command

Once executed, the evaluation of the minion strategy in that turn ends. This is similar to the `return` statement in a procedure.

`move` command

The `move` command moves the minion one unit in the specified direction. If the target hex is already occupied by another minion or falls outside the boundary of the territory, the command acts like a no-op. Whenever this command is executed (regardless of validity), the player's budget is decreased by one unit. If the player does not have enough budget to execute this command, the evaluation of the minion strategy in that turn ends.

`shoot` command

The `shoot` command tells the current minion to attempt to attack a hex located one unit away from it in the specified direction.

Each attack is given an expenditure, i.e., how much the minion would like to spend in that attack. This will be deducted from the player's budget. Each attack also has a fixed budget cost of one unit. That is, the total cost of an attack is $x+1$, where x is the given expenditure. If the player does not have enough budget to execute this command, the command acts like a no-op.

If the target hex is unoccupied, the player still pays the total cost of the attack, but the attack itself will have no effects otherwise.

Otherwise, the target hex contains a minion. The HP of the target minion will be decreased by no more than the expenditure. Specifically, if the expenditure is *x*, the target minion's current HP is *h*, and its defense factor is *d*, then after the attack, the target minion's HP will be max(0, *h*-max(1, *x*-*d*)). If the HP becomes less than one, the minion dies and disappears from the territory immediately.

If the target hex contains a minion belonging to the current player (i.e., a player is attacking its own minion), the command has the same effect as the normal scenario, i.e., it acts as self destruction. Be careful who you shoot at!

Arithmetic evaluation

Evaluation of expressions in minion strategies uses integer arithmetic. Division using the `/` operator is integer division, truncating decimals. Overflow and underflow are allowed to occur in order to fit results in Java's long data type. A division by zero results in the end of the evaluation (like the `done` command).

Boolean evaluation

Conditions in the `if` and `while` statements evaluate to integers. To interpret integer values as booleans, positive values are considered true, and negative values and zero are considered false.

Avoiding infinite loops

It is possible for a `while` loop to execute for too many iterations. To avoid infinite loops during an evaluation of a minion strategy, a `while` loop that has run for 10000 iterations is terminated, i.e., its guard automatically becomes false, and the subsequent statement is then evaluated.

In effect, a `while` statement in a minion strategy

```
while (e) s
```

works like the following code in most programming languages:

```
for (int counter = 0; counter < 10000 && e > 0; counter++) s
```

Endgame

The game ends when a player no longer has a minion in the territory. The game also ends after each player has played a number of turns as specified in the configuration file. In the latter case, the player with more minions left in the territory wins. If both players have the same number of minions left, the player with the greater sum of HPs of the remaining minions wins. If both players also have the same sum of HPs, the player with more remaining budget wins. Otherwise, the game results in a tie.

Sample minion strategy

# indicates the beginning of a comment, and is not part of the minion strategy. You can choose to support end-of-line comments in your tokenizer if desired.

```
t = t + 1  # keeping track of the turn number
m = 0  # number of random moves this turn
while (3 - m) {  # made less than 3 random moves
  if (Budget - 100) then {} else done  # too poor to do anything else
  opponentLoc = opponent
  if (opponentLoc / 10 - 1)
  then  # opponent afar
    if (opponentLoc % 10 - 5) then move downleft
    else if (opponentLoc % 10 - 4) then move down
    else if (opponentLoc % 10 - 3) then move downright
    else if (opponentLoc % 10 - 2) then move right
    else if (opponentLoc % 10 - 1) then move upright
    else move up
  else if (opponentLoc)
  then  # opponent adjacent to this minion
    if (opponentLoc % 10 - 5) then {
      cost = 10 ^ (nearby upleft % 100 + 1)
      if (Budget - cost) then shoot upleft cost else {}
    }
    else if (opponentLoc % 10 - 4) then {
      cost = 10 ^ (nearby downleft % 100 + 1)
      if (Budget - cost) then shoot downleft cost else {}
    }
    else if (opponentLoc % 10 - 3) then {
      cost = 10 ^ (nearby down % 100 + 1)
      if (Budget - cost) then shoot down cost else {}
    }
    else if (opponentLoc % 10 - 2) then {
      cost = 10 ^ (nearby downright % 100 + 1)
      if (Budget - cost) then shoot downright cost else {}
    }
    else if (opponentLoc % 10 - 1) then {
      cost = 10 ^ (nearby upright % 100 + 1)
      if (Budget - cost) then shoot upright cost else {}
    }
    else {
      cost = 10 ^ (nearby up % 100 + 1)
      if (Budget - cost) then shoot up cost else {}
    }
  else {  # no visible opponent; move in a random direction
    try = 0  # keep track of number of attempts
    while (3 - try) {  # no more than 3 attempts
      success = 1
```

```
      dir = random % 6
      # (nearby <dir> % 10 + 1) ^ 2 is positive if adjacent cell in <dir> has no ally
      if ((dir - 4) * (nearby upleft % 10 + 1) ^ 2) then move upleft
      else if ((dir - 3) * (nearby downleft % 10 + 1) ^ 2) then move downleft
      else if ((dir - 2) * (nearby down % 10 + 1) ^ 2) then move down
      else if ((dir - 1) * (nearby downright % 10 + 1) ^ 2) then move downright
      else if (dir * (nearby upright % 10 + 1) ^ 2) then move upright
      else if ((nearby up % 10 + 1) ^ 2) move up
      else success = 0
      if (success) then try = 3 else try = try + 1
    }
    m = m + 1
  }
}  # end while
```

Display and controls

We recommend that you display the territory on a webpage. As coders, you can use the Spring Framework to link your Java backend to your web frontend. As players, you should be able to send commands from the frontend to the Java backend.

Think about how to display the territory so that players have enough information to decide what to do and enjoy the game.

Configuration file

The configuration file contains many parameters that can be adjusted to balance the game. Each parameter value must be representable in Java's long data type.

Each parameter specification is of the form `<name>=<value>`. The meanings of parameter names are as follows:

- `spawn_cost`: the cost of spawning a new minion
- `hex_purchase_cost`: the cost of purchasing a hex to make it spawnable
- `init_budget`: the initial budget
- `init_hp`: the initial HP for a newly spawned minion
- `turn_budget`: the amount of additional budget for each turn
- `max_budget`: maximum allowable budget
- `interest_pct`: interest rate percentage
- `max_turns`: maximum turns allowed per game
- `max_spawns`: maximum number of spawns allowed per player per game

The order of names in the configuration file need not follow the list above.

Sample configuration file

```
spawn_cost=100
hex_purchase_cost=1000
init_budget=10000
init_hp=100
turn_budget=90
max_budget=23456
interest_pct=5
max_turns=69
max_spawns=47
```

Project timeline (subject to change)

- Mon 5 Jan:
    - Project team contract due
        - This is a formal document outlining the expectations, roles, and responsibilities of your team, covering all aspects of the team's work. It should specify the team's goals, how team members will communicate with each other, and what happens if someone violates the contract.
        - This includes guidelines or principles that team members agree upon, in order to facilitate collaboration, communication, and productivity within your team.
- Mon 5 Jan – Mon 12 Jan:
    - Team contract review with TAs (30 minutes)
- Mon 12 Jan:
    - Revised project team contract due
        - After meeting with TAs, your team might decide to revise your project team contract. This is your opportunity to resolve any lingering issues in the contract.
- Fri 30 Jan: Phase-1 deadline
    - Design overview document for user flow, game state, and minion-strategy evaluator due
        - This includes the outline of class hierarchy for your game state, parser, and evaluator.
        - This includes the outline of how you will represent your game state.
        - This includes the user flow of your game, written in FigJam.
        - Your design will be given feedback, so you can address flaws before you submit your code.
- Mon 2 Feb – Fri 6 Feb:
    - Phase-1 design review with TAs (45 minutes)
- Wed 11 Feb: Phase-2 deadline
    - Implementation of game state and minion-strategy evaluator due
        - At this point, your project should be able to parse and execute minion strategies, given a mock game state required for the execution.
    - Design overview document for user interface due
        - This includes the outline of the display (view and controller), designed in Figma.
        - Your design will be given feedback, so you can address flaws before you submit your code.
- Thu 12 Feb – Wed 18 Feb
    - Phase-2 design and implementation review with TAs (45 minutes)
- Mon 16 Feb / Thu 19 Feb:
    - Project update in class
        - Your team will present the design of your user interface to classmates. This is your opportunity to get feedback from classmates.

- Wed 25 Feb: Phase-3 deadline
    - Implementation of user interface due
        - At this point, your evaluator should be integrated with the game state.
        - At this point, your user interface should be displayable and can mock some strategy commands
    - Design overview document for server-client implementation due
        - This includes the outline of the interaction between game state and user interface.
        - Your design will be given feedback, so you can address flaws before you submit your code.
- Thu 26 Feb – Wed 4 Mar
    - Phase-3 design and implementation review with TAs (45 minutes)
- Wed 11 Mar: Phase-4 deadline
    - Server-client implementation due
        - At this point, your game state should be able to interact with your user interface, and vice versa.
    - Project report due
        - Architectural overview of your system
        - Report on what was done according to plan and outside the plan
        - Retrospect on what was done well, and what could have been better
    - Possible extension: after the final exam period (no later than Mon 30 Mar)

Failures to meet any of these requirements will result in the score of zero for all the subsequent requirements.