Name: Shant Stepanian
Uni: sps2141
Semester: Summer 2013 CVN
Course: COMS W4115
Assignment: Language Project Proposal

BoredGame Language Project Proposal

Purpose

BoredGame is a language designed to help people create their own games. As the tagline goes, “if you are bored from your existing board games, create a new one!”

BoredGame will specialize in specifying games based on a two-dimensional board. Among the features of the language that will help in this goal:

- Defining symbols for players and pieces
- Reading input and printing output
- Flexible input parsing to support various input formats
- Quick setup and access for the 2d board data

Note to professor on features:

There are a few languages features that ideally I would add, but as this is the first shot at a language, and I want to get something working above all, I wanted to start simple for now

- Supporting maps easily (e.g. with easy declaration and usage, similar to Python) would be nice, including support for maps of maps
    - But as I thought about it, it would prove difficult with the way Python does it with dynamic typing; I’d like to start with static typing
    - Though we can do this w/ static typing via Java-like generics (as I’ve tried w/ the “board” type), my board type is restricted to enums as its type, which I feel would be easier to support to start with than purely generic generics

I’m also still debating the following:

- In my switch statement for strings/regexps, I am considering if I should have regexps not require quotes, as to not make it equivalent to strings, like Java has it
    - Making it a string like Java is what I’m familiar with (and other Java folks could be familiar with), but having this as a regexp may be clearer for both the compiler

and the user (and less need and worries on expressing regular expressions in strings)
    - I am leaning towards having regexps be separate from strings, but I'm going to work through this as I experiment with the code

Language Grammar Overview:

Types:

- Primitives
    - *int* := [0-9]+
    - *string* := any character, no special characters or escapes
    - *boolean* := true false
- Ability to define new enum types using the *enum* keyword. This is here to facilitate the definition of pieces and players for the game
- *board<Type>* type that represents the 2d array of the board
    - Methods/operators available on board:
        - myboard.rowlength := returns the # of rows
        - myboard.collength := returns the # of columns
        - myboard[row,col] := read/write accessor. These are 1-indexed as most board coordinate systems are not 0-indexed
- *func* keyword to define functions
- Identifiers := [a-zA-Z]+ (i.e. only alphanumerics)

Keywords and Basic Language Constructs:

- Reserved keywords: int, string, boolean, enum, board, if, else, switch, case, func, printString, readString
- semi-colon to split statements
- Declarations are separate from assignments
- Blocks (i.e. for if/while/func) will be demarcated by braces {}
- Entry point is the function "main"
- No function overloading

Control constructs:

- if <stmt> else <stmt>
- for <id> in <array> { (<stmt>;)*}
- switch <id> (grp1, grp2, …) { case "string" {} … }
    - We will allow for switches on strings, with regular expressions supported and optionally being able to extract matched groups into variables. This is to facilitate reading groups

Built-in functions:

- printString(String), printMove(Move), …
    - i.e. a print method for each type
- readString(String), readMove(Move), …
    - i.e. a read method for each type

Standard Operators:

- Int operators (returns Int): + - * / (no floating points - all will be rounded)
- Int comparison operators (returns Boolean): < > <= >= == !=
- Boolean operators (returns Boolean): == != && ||

Coding Examples:

Coding Example #1 - Checkers:

```
enum PieceType { c C };
enum Player { p1 p2 };
enum Piece { x X o O _ };

func int main() {
    -- board runtime check - must be x by y exactly
    board<Piece> myboard;
    myboard = [{
        x _ x _ x _ x _ \
        _ x _ x _ x _ x \
        x _ x _ x _ x _ \
        _ _ _ _ _ _ _ _ \
        _ _ _ _ _ _ _ _ \
        _ o _ o _ o _ o \
        o _ o _ o _ o _ \
        _ o _ o _ o _ o \
        }];

    Player turn;
```

```
	turn = p1;
	string move;

	while (true) {
		Player winner;
		winner = gameover(board, turn);
		if (winner == null) {
			move = readString();

		} else {
			// game over
		}
	}
}

/*
We have this pattern-matching switch statement to allow for various input moves, e.g. for chess
0-0-0 or 0-0 for castling notation, which is different from the other move notations that are of the
form ([a-h])([0-9]])-([a-h])([0-9]])
*/
func boolean eval(string input, board<Piece> myboard, Player player) {
	int srcrow;
	int srccol;
	int tgtrow;
	int tgtcol;
	switch input {
		case “([a-h])([0-9]])-([a-h])([0-9]])” (string s_srcrow, string s_srccol, string
	s_tgtrow, string s_tgtcol) {
			srcrow = stringToInt(s_srcrow);
			srccol = stringToInt(s_srccol);
			tgtrow = stringToInt(s_tgtrow);
			tgtcol = stringToInt(s_tgtcol);

			Piece curpiece;
			curpiece = myboard[srcrow,srccol];
			if (player != getPiecePlayer(curpiece)) {
				printString(“Invalid move - player must own the piece”);
				return false;
			}

			Piece targetpiece;
			targetpiece = myboard[tgtrow,tgtcol];
			if (getPiecePlayer(targetpiece) != _ ) {
```

```
                        printString("Invalid move - target must be occupied");
                        return false;
                    }

                    // for now, will gloss over the capture steps to show move execution
                    myboard[srcrow, srccol] = _;
                    myboard[tgtrow, tgtcol] = curpiece;
                    return true;
                }
                case default {
                    printString("Invalid move input format");
                    return false;
                }
            }

}


// Would have preferred a more succinct way to represent this in the language,
// e.g. some kind of mapping syntax (x => X), but de-scoping this for now
func Player getPiecePlayer(Piece piece) {
        if (piece == x || piece == X) {
                return p1;
        } else if (piece == x || piece == X) {
                return p2;
        } else {
                return null;
        }
}

// Would have preferred a more succinct way to represent this in the language,
// e.g. some kind of mapping syntax (x => X), but de-scoping this for now
func PieceType getPieceType(Piece piece) {
        if (piece == x || piece == o) {
                return c;
        } else if (piece == X || piece == O) {
                return C;
        } else {
                return null;
        }
}
```

```
func int ind_to_number(input)
	switch (input) {
		case “a” {
			return 0;
		}
		case “b” {
			return 1;
		}
		// so on and so forth
		case default {
			return -1
		}
	}

func Player gameover(board, player) {
	map counts;
	int i;
	int j;

	int p1score;
	int p2score;
	p1score = 0;
	p2score = 0;

	for (i = 1; i <= board.rowlength; i++) {
		for (j = 1; i <= board.collength; i++) {
			piece = board[i,j];
			if (getPiecePlayer(piece) == p1) {
				p1score = p1score + 1;
			} else if (getPiecePlayer(piece) == p2) {
				p2score = p2score + 2;
			}
		}
	}

	if (p1score == 0) {
		return p1;
	} else if (p2score == 0) {
		return p2;;
	} else {
		return null;
	}
```

```
}
```