BGGL (Board Game Generator Language)
Language Reference Manual

COMS W4115: Programming Languages and Translators
Professor Stephen Edwards
sedwards@cs.columbia.edu

{
Matt Chu (mwc2110)
Steve Moncada (sm2277)
Hrishikesh Tapaswi (hat2107)
Vitaliy Shchupak (vs2042)
}

BGGL Language Reference Manual

1. Lexical Conventions

1.1 Introduction

There are five classes of tokens used in BGGL. They are identifiers, keywords, numbers, string literals, and other tokens. White space is used to separate different tokens in BGGL. In the context of this language, white space refers to spaces, tabs, and new lines.

1.2 Comments

Single-line comment-style introduced by two forward slashes "//" causes the compiler to ignore the rest of the line.

```
// this is a comment
```

1.3 Identifiers

Identifiers in BGGL are any combination of letters, digits, and underscores ("_") whose first character is a letter.

```
king    queen_3   P_
```

1.4 Keywords

The following words are reserved and therefore may not be used as identifiers:

General Keywords

```
boolean
break
continue
else
false
if
int
print
return
string
true
while
```

Domain-Specific Keywords

```
board
col
diagonal
game
jump
move
movement
orthogonal
piece
row
rule
thisplayer
winner
```

1.5 Numbers

Numbers in BGGL will consist of digits. All numbers will be treated as standard 32-bit integer types.

1.6 String Literals

String literals are any characters enclosed by a set of double quotes (“<any combination of characters>”). To include an opening or closing double quote inside a string literal, type backslash + ”. (\”)

1.7 Other Tokens

There are a set of single characters that must be used correctly according to BGGL’s syntax. They include:

```
{    }    (    )    [    ]
+    -    *    /    %    <    >
!    :    ;    ,    #    _
```

There are also a set of character pairs which must be used correctly according to BGGL’s syntax. They include:

```
&&    ||    ==    <=    >=    !=
```

2. Types

`boolean`	:	standard Booleans, true or false
`int`	:	standard 32-bit integers
`string`	:	a string of characters
`move`	:	BGGL type for action-level game specifications
`piece`	:	BGGL type for individual player-level specifications

3. Expressions

3.1 Introduction

The following section explains the behavior of expressions in BGGL. The order of the subsections indicates expression operator precedence and the specific linear associativity will be explicitly stated within each subsection. For the most part, expressions in BGGL follow standard conventions almost identically.

3.2 Primary Expressions

Primary expressions include identifiers, parenthesized expressions, function calls, and BGGL-specific constructs. A function call contains a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to a function. *The associativity of primary expressions is left-to-right.*

3.3 Unary Operator

The unary operator includes the ! token. It produces the negation of the adjacent expression. *The associativity of the unary operator is right-to-left.*

3.4 Multiplicative Operators

Multiplicative operators include the tokens *, /, and %. The binary * triggers

multiplication. The binary / triggers division. The binary % triggers the modular division operator, which gives the remained from the division of the first expression by the second expression. *The associativity of multiplicative operators is left-to-right.*

3.5 Additive Operators

The additive operators include the tokens + and -. The binary + yields the sum of the expressions and the binary – yields the difference of the expressions. *The associativity of additive operators is left-to-right.*

3.6 Relational and Equality Operators

The relational and equality operators include the tokens <, >, <=, >=, ==, and !=. Both sets of operators yield 0 if the binary relation is false and 1 if it is true. *The associativity of relational and equality operators is left-to-right or right-to-left.*

3.7 The Boolean AND Operator

The Boolean AND operator includes the token &&. It returns 1 if both expressions surrounding the token are non-zero. It returns 0 is both expressions are 0. *The associativity of the Boolean AND is left-to-right.*

3.8 The Boolean OR Operator

The Boolean OR operator includes the token ||. It returns 1 if either expression surrounding the token are non-zero. It returns 0 is both expressions are 0. *The associativity of the Boolean OR is left-to-right. If the first expression is non-zero, the second expression is not evaluated.*

3.9 Assignment Operator

The assignment operator includes the token =. It requires that an l_value be its left operand and that the type of the assignment expression is the same as the left operand. After assignment, the resulting value is stored in the left operand. *The associativity of the assignment operator is right-to-left.*

4. Statements

4.1 Introduction

Statements in BGGL are executed sequentially from top to bottom. For the most part, statements in BGGL follow standard conventions almost identically.

4.2 Expression statement

A majority of statements in BGGL are expression statements, which are of the following form:

```
expression ;
```

4.3 Conditional statement

There are two forms of BGGL's conditional statement:

```
if ( expression ) { [ statement ] * ; }
if ( expression ) { [ statement ] * ; }
  else { [ statement ] * ; }
```

In both cases the expression is evaluated and if it is non-zero, the first sub-statement is executed. In the second case the second sub-statement is executed if the expression is 0. As per convention, ambiguity regarding the else is resolved by connecting an else with the last-encountered "elseless if."

4.4 Iterative Statements

BGGL supports a standard while loop structure. This statement will execute the commands and statements inside the open and closed brackets while the expression contained in parenthesis evaluates to a non-zero number using BGGL's Boolean support. This expression is checked when the statement is first encountered in the program's sequence, and at each time execution reaches the final close-bracket. The loop will continue to repeat until the expression evaluates to 0, at which time the execution will bypass the block and proceed sequentially through the rest of the program.

```
while ( expression ) { [ statement ] * ; }
```

4.5 Return Statement

A function returns to the invocation token via a return statement, which may have either of the following two forms:

```
return ;
return ( expression ) ;
```

In the first case nothing is returned. In the second case, the value of the expression is returned to the caller of the function.

5. Functions

5.1 Introduction

A function gathers a sequence of BGGL statements into a named piece of code. This section describes how functions are defined and how they are invoked.

5.2 Function Declarations

The following structure illustrates the acceptable method of defining a function:

```
return_type function_name ( parameter_list ) {
  [ statement ] * ;
  [ return_statement ] ;
}
```

Function_name is the name of the function that user has given and the parameter list is a list of identifier of arguments, separated by commas. Parameter list can be empty.

5.3 Function Invocations

Function calls are of the following form:

```
function_name ( parameter ) ;
```

A function must be defined before it is invoked, and the parameter list must contain types identical to the ones expected according to the function definition.

6. BGGL-Specific Conventions

6.1 Introduction

Two conventions which are used extensively in BGGL are ones which pertain to board game layout's coordinate system and rule-specification syntax. The deviations from the aforementioned rule structure demonstrate the way BGGL is tailored to the specific domain of board game generation.

6.2 Board Coordinate Conventions

A board may be declared in two ways:

```
Board(8,8);  //all pieces are _ by default
Board  =    [
            [#,B,#,B,#,B,#,B]
            [B,#,B,#,B,#,B,#]
            [#,B,#,B,#,B,#,B]
            [_,#,_,#,_,#,_,#]
            [#,_,#,_,#,_,#,_]
            [W,#,W,#,W,#,W,#]
            [#,W,#,W,#,W,#,W]
            [W,#,W,#,W,#,W,#]
            ];
```

In both cases a board with eight regions across and eight regions down is created. The tokens used in the more-explicit second example are #, _, and identifiers. The # indicates that the region is invalid. The _ indicates a Piece variable for "no piece." The identifiers also indicate Piece variables.

BGGL only permits one board declaration. This way, a specific coordinate on the board can always be accessed using the following syntax:

```
[x][y]
```

This means that checking to see if a specific region of the board is empty would be accomplished using the following statement:

```
if ( [3][3] == _ )
```

Array out of bounds checking is implicit within the program and will print an error that will not halt execution of the program.

The ability to quickly reference specific rows or columns is also supported using the following syntax.

```
row[1]
```

```
col[1]
```

In both of these cases, an array is returned. The structure of this array will be identical to the initialization syntax. For example, in tac tac toe, to check if the game is over, you could use

```
if (row[1] == [X,X,X])
```

This row and column referencing ability can also be used during initialization.

```
row[1] = [X,X,X]
```

6.3 Rule Syntax Conventions

Rules are blocks of code which automatically check `move`'s that involve movement of pieces for validity. For example:

```
Rule rulename:piece1, piece2 {
       length == 1;
       movement == orthogonal;
       jump == false;
       emptysquare == true;
       x2 > x1;
}
```

This sets the four basic parameters that define the movement of a piece to the values specified.

- `length` is how many valid board squares (not including any squares that might have been marked invalid when the board is declared) the piece is allowed to move.
- `movement` defines the direction in which the piece is allowed to move. '`orthogonal`' is along the x and y axes and '`diagonal`' is diagonal.
- `jump` defines if the piece is allowed to jump over any other region

when moving.

- emptysquare flag is set when the piece can only end up in a region that is empty.
- If any of these parameters are not set, any arbitrary movement is possible.
- It is also possible to include other Boolean statements involving coordinates of the old region and new square that allow greater flexibility in defining the movement.

6.4 Additional Conventions

- A rule is satisfied only if all statements in the rule block are true.
- `function numPieces(piece)` returns the number of pieces on the board.
- `function makemove(m)` first calls the `isValidMove function()`, then atomically applies move(s) m to the board. If any move fails any rule, the entire move is not done.
- syntax of move variable is `x:y:piece:direction`, where direction is either + or -. For example to remove a black piece from (3,3), you could use `m = 3:3:B:-; makemove(m);`
- There are special variables: `thisplayer`, `winner`, `board`
- The main game must be contained in the Game section.
- A turn block loops thru the code inside the block until an exit statement is called
- Typically, in the function definition for `isValidMove()`, a statement like `move1 : rule1 || rule2;` would check all the submoves of move1 and look to see if any of rule1 and rule2 act upon any of the moving pieces involved in move1 and if the movements are valid according to the Rules.
- Most of the basic elements of movement checking for pieces will be covered by the Rules. The programmer is expected to have additional code in functions like `isValidMove()` to check for special cases while determining the legality of a move.

7. Sample Program - Checkers

```
Player p1;

Player p2;

Piece W, B, Wk, Bk; //white, white king, black, black king

//player 1 = White, player2 = black

Board = [[#,B,#,B,#,B,#,B]

         [B,#,B,#,B,#,B,#]

         [#,B,#,B,#,B,#,B]

         [_,#,_,#,_,#,_,#]

         [#,_,#,_,#,_,#,_]

         [W,#,W,#,W,#,W,#]

         [#,W,#,W,#,W,#,W]

         [W,#,W,#,W,#,W,#]];


boolean isGameOver() {

      if (numPieces(B)  + numPieces(Bk) == 0) {

            winner = p1;

            return true;

      else if (numPieces(W) + numPieces(W k)== 0) {

            winner = p2;

            return true;

      }

      return false;

}
```

```
Player nextPlayer() {
      if (thisplayer == p1) {
            return p2;
      }
      else {
            return p1;
      }
}
//specifies rules for valid coordinates
Rule validCoord: W,Wk,B,Bk {
      movement = diagonal;
      emptysquare = true;
      jump = true;
}
Rule whiteForward: W {
      m[1].x > m[0].x;  // whites can move up
}
Rule blackForward: B {
      m[0].x > m[1].x;   // black pieces can only move
down
}
Rule correctPlayer1: W { //player 1 moves white pieces
     thispayer == p1;
}
```

```
Rule correctPlayer2: B {  //player 2 moves black pieces
     thisplayer == p2;
}
Rule capturePiece: B,W {
     m[0].piece != m[2].piece;  //capturing opponents piece
}

//main code for all pieces manipulation is here
Move[] getMove() {
      Move m[3]; // up to 3 moves at a time
      // (x1, y1) = piece to be moved
      // (x2, y2) = where to place
      x1 = input ("Enter x1:");
      x2 = input ("Enter x2:");
      y1 = input ("Enter y1:");
      y2 = input ("Enter y2:");
      m[0] = x1:y1:[x1][x2]:-; //remove piece from square
      m[1] = x2:y2:[x2][y2]:+; //place piece on destination
square
    //if jumping over a piece, remove the piece jumped over
      if (length(m) == 2) {
            m[2] =
(x1+x2)/2:(y1+y2)/2:[(x1+x2)/2]:[(y1+y2)/2]:-;
      }
      return m;
```

```
}

isValidMove(m) {

      //check that move m follows all the rules (for the
piece for each rule)

      m: validCoord && nonKing && whiteForward &&
blackForward && correctPlayer1 && correctPlayer2 &&
capturePiece;

}


Game

{

      thisplayer=p1;

      turn mainturn

      {

            Move m[] = getMove(); //two moves, one removes
piece, one adds piece

            makemove m;

            }

            else {

                  gototurn mainturn;

            }

            if (isGameOver()) {

                  print "Game Over! Winner is " + winner);

            }

            thisplayer = nextPlayer();

      }}
```