dabor Language Reference Manual

Khaela Harrod(klh2173), Seoyoung Kweon(sk4865), Nicholas Greenspan(nrg2148), Annie Song(afs2185), Siyi Hong(sh4325), and Hemanth Chandra Palle(hp2581)

Table of Contents

1. Introduction

`dabor` is an object-oriented language that facilitates modeling of 2D board games. With a variety of built-in types and features, `dabor` factors out the most commonly used chunks of game logic so that board game programmers can focus on what matters the most: making good games.

2. Lexical Conventions

Lexical conventions can be broken down into six classes: tokens, identifiers, keywords, literals, operators, and separators.

i. Tokens

In `dabor`, blanks, tabs, or newline symbols are considered as whitespace characters. They are mostly used to separate tokens and are oftentimes ignored.

```
let whitespace = [‘ ’ ‘\t’ ‘\r’]
let newline = ‘\n’
```

Comments can be single lined.

```
// This is the start of a single line comment.
```

ii. Identifiers

Identifiers in the language are defined as sequences of characters and digits, and they must start with characters. Identifiers are case sensitive as the difference between upper and lower case letters are distinct.

```
let alpha = [‘a’-’z’ ‘A’-’Z’]
let digit = [‘0’-’9’]
```

iii. Keywords

```
// new keywords for our language
vector
diagonalLeft
diagonalRight
horizontal
vertical
matrix
matrix_create
move

//primitive types
bool
```

```
int
string
true
false

// statements
if
else
while
for
break
continue
and
or
not

// non-primitive data type
struct
duple
```

iv. Literals

Literals are sequences of characters whose values cannot be changed or reassigned.

Integer literals are a sequence of digits represented by whole numbers

```
let digit = [‘0’-‘9’]
let int = digit+
```

Boolean literals are keywords only: true or false

```
true
false
```

String literals are a sequence of characters enclosed between a set of double quotation marks

```
string_intro = “Welcome to dabor”;
```

v. Operators

Operators are used to perform common transformations or operations on one or more entities. `dabor` includes operators for primitive built-in types as well as non-primitive built-in types. Here we list the operators that we support.

Arithmetic: `+, -, *, /, %`

Assignment: `=`

Equivalence: `==, !=, <, <=, >, >=`

Logical: `and, or, not`

Move Function: `move`

vi. Separators

Separators are used to denote the separation between tokens.

```
( ) { } [ ] ; , :
```

3. Types

i. Primitive Data Types

int

The integer data type stores 32-bit signed two's complement integers with minimum value of -2^{31} and maximum value of $2^{31}-1$.

Syntax:

```
int x;
x = 32;
```

string

The string type stores a sequence of ASCII characters.

Syntax:

```
string x;
x = "hello";
```

bool

Implemented as either 'true' or 'false'.

Syntax:

```
bool x;
x = true;
```

ii. Non-Primitive Data Types

Matrix

The matrix type represents the state of the game board, as well as possibly sub portions of the board. It is a 2d array that can hold a mixture of data types such as strings or ints (or pointers to structs). It is declared using the “matrix” keyword which acts as a constructor function which takes a 2d array as input. The elements of the underlying 2d array can be modified, but the array itself can’t change in size.

Syntax:

```
matrix board;
matrix board2;
matrix board3;
board = matrix_create([[0, 1, 0],[1, 0, 1],[0, 1, 0]]);
board2 = matrix_create([[“water”, “water”, “land”],[“water”, “land”,
“land”],[“land”, “land”, “land”]])
board3 = matrix_create([“water”, “water”, “water”],[1, 0, 1],[0, 1, 0]])
board[10, 10] = 5
```

Grammar:

```
MATRIX_C LPAREN matrix_rule RPAREN { MatrixCreate($3) }// matrix creation
ID LBRACK INT COMMA INT RBRACK { MatrixAccess($1, $3, $5) } // matrix access
ID LBRACK id_rule RBRACK { MatrixAccess($1, $2) } // matrix access
```

Struct

A struct can be used to represent objects for use during game play, such as a board piece or a player. The fields of a struct are defined by the programmer and can have different types.

Syntax:

```
struct player {points: int, num_pieces: int};
struct player p;

p = player{points: 0, num_pieces: 4};
p.points = p.points + 1;
```

Grammar:

```
STRUCT ID LBRACE struct_def_list RBRACE { StructDef($2, $4) } //struct
definition
LBRACE struct_list RBRACE {StructCreate($2)} //struct creation
ID DOT ID {StructAccess($1, $2)} //struct access
```

Vector

The vector type represents the direction and magnitude of the move in the matrix. It is denoted `vector <vector_name>` in declaration `<vector_name> = <movement_key>(<int>)` in definition. They can be added, subtracted from each other, and multiplied and divided with an integer.

Syntax:

```
vector vector_1;
vector_1 = horizontal(-1);
```

Grammar:

```
// vector create
DIAGRIGHT LPAREN INT RPAREN   { VectorCreate(DiagR, $3) }
DIAGLEFT LPAREN INT RPAREN    { VectorCreate(DiagL, $3) }
HORIZONTAL LPAREN INT RPAREN  { VectorCreate(Hori, $3) }
VERTICAL LPAREN INT RPAREN    { VectorCreate(Vert, $3) }
```

Duple

The duple type is a pair of integers that represents the direction of a move. The first integer represents change in the x position and the second change in y position.

Syntax:

```
duple dir;
dir = (10, 8);
```

Grammar:

```
// vector create
LPAREN INT COMMA INT RPAREN { DupleCreate($1, $3) }
```

4. Operators

i. Arithmetic

The binary arithmetic operators are `+, -, *, /, %`. These operators associate left to right, where operators `+` and `-` are evaluated first preceded by `*, /, %`. The operators `+` and `-` can be used between two vectors, and `*, /, %` can be used between a vector and an integer.

Syntax:

```
x = (1+2)*4;
```

Grammar:

```
expr_rule:
      expr_rule PLUS expr_rule        { Binop($1, Add, $3) }
      | expr_rule MINUS expr_rule     { Binop($1, Sub, $3) }
      | expr_rule MULTIPLY expr_rule  { Binop($1, Mult, $3) }
      | expr_rule DIVIDE expr_rule    { Binop($1, Div, $3) }
      | expr_rule MOD expr_rule       { Binop($1, Mod, $3) }
```

ii. Assignment

The assignment operators are `=`. This operator assigns an expression to a variable.

Syntax:

```
x = 1;
```

Grammar:

```
id_rule ASSIGN expr_rule { Assign($1, $3) }
```

iii. Logical

The logical operators are `and, or, not`. These operators are used for combining evaluations of any original expression.

Syntax:

```
if (not valid)
if (valid or not_valid)
if (a_valid and b_valid)
```

Grammar:

```
expr_rule:
      expr_rule AND expr_rule    { Binop($1, And, $3) }
      | expr_rule OR expr_rule   { Binop($1, Or, $3) }
      | expr_rule expr           { Unop(Not, $2) }
```

iv. Equivalence

The relational operators are `<, <=, >, >=`. They all have the same precedence. Below them are `==, !=`. Relational operators have lower precedence than arithmetic operators.

Syntax:

```
bool = ((1, 2) == (1, 2));
```

Grammar:

```
expr_rule:
    | expr_rule EQ expr_rule  { Binop($1, Equal, $3) }
    | expr_rule NEQ expr_rule { Binop($1, Neq, $3) }
    | expr_rule LT expr_rule  { Binop($1, Less, $3) }
    | expr_rule LEQ expr_rule { Binop($1, Leq, $3) }
    | expr_rule GT expr_rule  { Binop($1, Greater, $3) }
    | expr_rule GEQ expr_rule { Binop($1, Geq, $3) }
```

v. Move Function

The “move” function needs two inputs: a vector and a duple. The vector declares the direction of movement (diagonalLeft, diagonalRight, horizontal, vertical) and how many steps to move the coordinate pair represented by the duple. The following example shows: First initialize of board of 3*3, then move diagonally up and right 1 step the “king” piece located on (1,0), so the new position of “king” is (0,1).

Syntax:

```
struct Piece {name: string, location: duple};

matrix board;
vector vector_1;
duple loc;
struct Piece pi;
int x;

board = matrix_create([[0, 0, 0],[0, 1, 0],[0, 0, 0]]);
vector_1 = diagonalRight(1);
loc = (1, 1);
pi = Piece {name: “king”, location: loc};
x = Board[pi.location];
Board[pi.location] = 0;
pi.location = pi.location move vector_1;
Board[pi.location] = x;
```

Grammar:

```
expr_rule MOVE expr_rule   { Binop ($1, Move, $3)  }
```

5. Statements and Expressions

The high level difference between statements and expressions is that expressions have a value, but statements do not.

Variables are declared and assigned separately. Variables are declared with a type annotation and string identifier on the left hand, and they are assigned with the string identifier, equals sign and the value of the variable on the right side of the equals sign.

An expression followed by a semicolon is a statement. All variables must be declared in the program before there are any other statements or expressions.

Syntax:

```
matrix board;
board = matrix_create([[0, 1, 0],[1, 0, 1],[0, 1, 0]]);
```

i. Literal Expressions

Literal expressions must end with a semicolon.

```
“Matrix”; // string literal expression
5;        // int literal expression
true;     // bool literal expression
```

ii. Matrix Expressions

A matrix is a two dimensional matrix that simulates the board and stores types that the user chooses to represent their pieces with. A matrix is created by calling the matrix keyword and specifying the content and size of the matrix.

```
matrix board;
board = matrix_create([[0, 0, 0],[0, 1, 0],[0, 0, 0]]);
```

iii. Vector Expressions

Vectors have possible directions such as diagonalLeft, diagonalRight, horizontal, vertical and store distance values. These expressions are used to facilitate the Piece movement through move function.

```
vector vector_1;
vector1 = diagonalLeft(-2);
```

iv. Struct Expressions

Struct expressions are enclosed in braces separated by a comma. Struct definition has a variable name and type defined separated by colon. These expressions provide a new data type to structure variables. Struct must be defined before the statement

```
struct Piece {name: string, location: duple};
struct Piece p;

p = Piece{“king”, (0,0)};
```

v. Control Flow

If/Else:

The if/else statement executes a block of code for a given true condition. Otherwise another block of code is executed.

Syntax:

```
if (condition) {
//block of code
} else {
//block of code
}

if (condition) {
//block of code
}
```

Grammar**:**

```
if_statement:
   IF LPAREN expr_rule RPAREN stmt_rule %prec NOELSE    {If ($3, $5,[])}
 | IF LPAREN expr_rule RPAREN stmt_rule ELSE stmt_rule  {If($3, $5, $7)}
```

While:

The while expression executes a block of code while the condition is true. If this condition is true, the block of code is then executed and control returns the expression. If the condition is false, then the while expression completes.

Syntax:

```
while (condition) {
//block of code
}

int i;
i = 0;

while ( i < 15 ) {
    i = i + 1;
}
// i would be equal to 15.
```

Grammar**:**

```
WHILE LPAREN expr_rule RPAREN stmt_rule { While($3, $5) }
```

break/continue

The language supports break and continue expression in control flow.

A `break` expression immediately terminates the innermost while loop enclosing it.

A `continue` expression immediately terminates the current iteration of the innermost while loop enclosing it, returning control to the conditional expression of the while loop.

Both of them are only permitted in the body of a loop.

6. Grammar

i. Scanner

```
let digit = [‘0’-‘9’]
let alpha = [‘a’-‘z’ ‘A’-‘Z’]
let int = digit+
let id = alpha (digit | alpha | ‘_’)*
let string = ‘“’ ((ascii)* as s) ‘”’
let newline = ‘\n’

rule token = parse
 [‘ ’ ‘\t’ ‘\r’] { token lexbuf }
| newline { incr depth; token lexbuf }
| “//” { comment lex}
| “vector” { VECTOR }
| “diagonalLeft” { DIAGLEFT }
| “diagonalRight” { DIAGRIGHT }
| “horizontal” { HORIZONTAL }
| “vertical” { VERTICAL }
| “matrix” { MATRIX }
| “matrix_create” { MATRIX_C }
| “move” { MOVE }


//types
| “bool” { BOOL }
| “int” { INT }
| “true” { BLIT(true) }
| “false” { BLIT(false) }

| “string” { STRING }
| “struct” { STRUCT }
| “tuple” { TUPLE }

| “if”  { IF }
| “else” { ELSE}
| “while” { WHILE }
| “continue” { CONTINUE }
| “break” { BREAK }

| “and” { AND }
```

```
| “or” { OR }
| “not” { NOT }

| ‘.’ { DOT }

| ‘(‘ { LPAREN }
| ‘)’ { RPAREN }
| ‘{’ { LBRACE }
| ‘}’ { RBRACE }
| ‘[’ { LBRACK }
| ‘]’ { RBRACK }
| ‘;’ { SEMI }
| ‘:’ { COLON }
| ‘,’ { COMMA }

| ‘+’ { PLUS }
| ‘-’ { MINUS }
| ‘*’ { MULTIPLY }
| ‘/’ { DIVIDE }
| ‘=’ { ASSIGN }
| “==” { EQ }
| “!=” { NEQ }
| ‘<’ { LT }
| “<=” { LEQ }
| ‘>’ { GT }
| “>=” { GEQ }
| ‘%’  { MOD }

| int as lem { INT_LITERAL(int_of_string lem) }
| string { STRING_LITERAL(s) }
| id as lem { ID(lem) }
| eof { EOF }
| ‘“’ { raise (Exceptions.UnmatchedQuotation(!lineno)) }
| _ as illegal { raise (Exceptions.IllegalCharacter(!filename, illegal,
!lineno)) }

and comment = parse
newline { token lexbuf }
 | _ { comment lexbuf }
```

ii. Parser

```
%token DOT SEMI COLON COMMA LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token PLUS MINUS MULTIPLY DIVIDE MOD ASSIGN
%token EQ NEQ LT LEQ GT GEQ AND OR NOT
%token IF ELSE WHILE CONTINUE BREAK
%token VECTOR DIAGLEFT DIAGRIGHT HORIZONTAL VERTICAL MATRIX MATRIX_C MOVE
%token STRING STRUCT DUPLE
%token <int> INT_LITERAL INT
%token <bool> BLIT BOOL
%token <string> ID STRING_LITERAL
%token EOF

%right ASSIGN
%left NOT
%left OR
%left AND
%left EQ NEQ LT LEQ GT GEQ
%left MULTIPLY DIVIDE MOD
%left PLUS MINUS
%left MOVE

%start program_rule
%type <Ast.program> program_rule

%nonassoc NOELSE
%nonassoc ELSE

%%

program_rule:
   vdecl_list_rule stmt_list_rule EOF { {locals = $1; body = $2} }

vdecl_list_rule:
   /*nothing*/                  { [] }
 | vdecl_rule vdecl_list_rule    { $1 :: $2 }

vdecl_rule:
   typ_rule ID SEMI                          { ($1, $2) }
 | STRUCT ID LBRACE struct_def_list RBRACE   { StructDef($2, $4) }

typ_rule:
```

```
   INT         { Int  }
 | BOOL        { Bool }
 | STRING      { String }
 | MATRIX      { Matrix }
 | VECTOR      { Vector }
 | DUPLE       { Duple }
 | STRUCT ID   { StructT($2) }

struct_def_list:
   ID COLON typ_rule                          { [$1, $3] }
 | ID COLON typ_rule COMMA struct_def_list   { ($1, $3) :: $5 }

struct_list:
   ID COLON expr_rule                     { [$1, $3] }
 | ID COLON expr_rule COMMA struct_list   { ($1, $3) :: $5 }

stmt_list_rule:
   /* nothing */               { [] }
 | stmt_rule stmt_list_rule    { $1::$2 }

stmt_rule:
   expr_rule SEMI                                          { Expr $1 }
 | LBRACE stmt_list_rule RBRACE                            { Block $2 }
 | IF LPAREN expr_rule RPAREN stmt_rule ELSE stmt_rule   { If ($3, $5, $7) }
 | IF LPAREN expr_rule RPAREN stmt_rule %prec NOELSE     {If ($3, $5, [])}
 | WHILE LPAREN expr_rule RPAREN stmt_rule               { While ($3,$5) }

rest_of_list_rule:
   INT_LITERAL COMMA rest_of_list_rule                { $1::$3 }
 | STRING_LITERAL COMMA rest_of_list_rule             { $1::$3 }
 | INT_LITERAL RBRACK /* no empty lists allowed */     { $1::[] }
 | STRING_LITERAL RBRACK /* no empty lists allowed */  { $1::[] }

list_rule:
   LBRACK rest_of_list_rule { $2 }

rest_of_matrix_rule:
   list_rule COMMA rest_of_matrix_rule  { $1::$3 }
 | list_rule RBRACK                     { $1::[] }

matrix_rule:
   LBRACK rest_of_matrix_rule  { $2 }
```

```
id_rule:
   ID                                    { Id $1 }
 | ID DOT ID                             { StructAccess{$1, $3} }
 | ID LBRACK INT_LITERAL COMMA INT_LITERAL RBRACK { MatrixAccess($1, $3, $5) }

expr_rule:
   BLIT                                  { BoolLit $1 }
 | INT_LITERAL                           { IntLit $1 }
 | STRING_LITERAL                        { StringLit $1 }
 | ID                                    { Id $1 }
 | id_rule ASSIGN expr_rule              { Assign ($1, $3) }
 | expr_rule PLUS expr_rule              { Binop ($1, Add, $3) }
 | expr_rule MINUS expr_rule             { Binop ($1, Sub, $3) }
 | expr_rule MULTIPLY expr_rule          { Binop ($1, Multiply, $3) }
 | expr_rule DIVIDE expr_rule            { Binop ($1, Divide, $3) }
 | expr_rule EQ expr_rule                { Binop ($1, Equal, $3) }
 | expr_rule NEQ expr_rule               { Binop ($1, Neq, $3) }
 | expr_rule LT expr_rule                { Binop ($1, Less, $3) }
 | expr_rule LEQ expr_rule               { Binop ($1, Leq, $3) }
 | expr_rule GT expr_rule                { Binop ($1, Greater, $3) }
 | expr_rule GEQ expr_rule               { Binop ($1, Geq, $3) }
 | expr_rule AND expr_rule               { Binop ($1, And, $3) }
 | expr_rule OR expr_rule                { Binop ($1, Or, $3) }
 | expr_rule MOD expr_rule               { Binop ($1, Mod, $3) }
 | expr_rule MOVE expr_rule              { Binop ($1, Move, $3) }
 | NOT expr_rule                         { Unop(Not, $2) }
 | DIAGLEFT LPAREN INT_LITERAL RPAREN    { VectorCreate(DiagL, $3) }
 | DIAGRIGHT LPAREN INT_LITERAL RPAREN   { VectorCreate(DiagR, $3) }
 | HORIZONTAL LPAREN INT_LITERAL RPAREN  { VectorCreate(Hori, $3) }
 | VERTICAL LPAREN INT_LITERAL RPAREN    { VectorCreate(Vert, $3) }
 | LPAREN expr_rule RPAREN               { $2 }
 | LBRACE struct_list RBRACE             { StructCreate($2) }
 | ID DOT ID                             { StructAccess($1, $3) }
 | MATRIX_C LPAREN matrix_rule RPAREN    { MatrixCreate($3) }
 | ID LBRACK INT_LITERAL COMMA INT_LITERAL RBRACK { MatrixAccess($1, $3, $5) }
 | ID LBRACK id_rule RBRACK                       { MatrixAccess($1, $3) }
 | LPAREN INT_LITERAL COMMA INT_LITERAL RPAREN    { DupleCreate($2, $4) }
```

7. Reference

- Rusty Language Reference Manual: http://www.cs.columbia.edu/~sedwards/classes/2016/4115-fall/lrms/rusty.pdf
- GNU C Reference Manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html