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
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

// print functions
print_int
print_string
print_matrix
```

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

```
let string = ‘“‘ ((digit | alpha | ‘_’ | ‘ ‘)* as s) ‘“‘
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 an integer data type. It is declared using the “matrix [row][col]” keyword which acts as a constructor function which takes a 2d array as input. “row” and “col” specifies the size of matrix to be declared. The elements of the underlying 2d array can be modified, but the array itself can’t change in size.

Syntax:

```
matrix[3][3] board;

board = matrix_create([[0, 1, 0],[1, 0, 1],[0, 1, 0]]);
board[2, 2] = 5;
```

Grammar:

```
MATRIX_C LPAREN matrix_rule RPAREN { MatrixCreate($3) }// matrix creation
ID LBRACK INT_LITERAL COMMA INT_LITERAL RBRACK { IndexAccess($1, $3, $5) }
// matrix access
ID LBRACK id_rule RBRACK { IndexAccessVar($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. The struct member cannot be involved in expressions or accessing matrix. Assign with literal or id is only possible.

Syntax:

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

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

Grammar:

```
STRUCT ID LBRACE struct_def_list RBRACE SEMI { 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 with pair of integer. It is denoted `vector [vector_name]` in declaration `[vector_name] = <[int], [int]>` in definition. They can be added, subtracted from each other, and multiplied and divided with an integer.

Syntax:

```
vector vector_1;
vector_1 = <0, -1>;
```

Grammar:

```
// vector create
LT expr_rule COMMA expr_rule GT       { VectorCreate($2, $4) }
```

Duple

The duple type is a pair of integers that represents the direction of a move. The first integer represents change in the row of a matrix and the second change in column of a matrix.

Syntax:

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

Grammar:

```
LPAREN INT COMMA INT RPAREN { DupleCreate($1, $3) } // duple create
ID LBRACK INT_LITERAL RBRACK { DupleAccess($1, $3) } // duple access
```

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 MOD expr_rule       { Binop($1, Mod, $3) }
      | MINUS expr_rule               { Unop(Neg, $2) }
```

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) }
      | Not 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. `<, <=, >, >=` can be applied to integers, and `==, !=` can be applied to integer and string.

Syntax:

```
bool = (“hello” == “hello”);
```

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, EqLess, $3) }
    | expr_rule GT expr_rule  { Binop($1, Greater, $3) }
    | expr_rule GEQ expr_rule { Binop($1, EqGreater, $3) }
```

v. Move Function

The “move” function needs two inputs: a duple and a vector. The duple provides the coordinate pair of current location and the vector declares the magnitude and direction that will move it. The move function will apply the vector on duple and return a new duple. The sequence of application should always be `<duple> move <vector>`. The following example shows: First initialize of board of 3*3, then move the “king” piece located on (1,0) to 1 step in each row and column, so the new position of “king” is (0,1).

Syntax:

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

matrix[3][3] 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 = <1, 1>;
loc = (1, 1);
pi = Piece {name: “king”, location: loc};
x = board[loc];
board[loc] = 0;
loc = loc move vector_1;
pi.location = loc;
board[loc] = x;
```

Grammar:

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

vi. Print Functions

The print functions are supported for most of data types for convenience of users. The functions are pre-defined for each data types. Matrix print function can only be used by calling the id of a assigned matrix.

Syntax:

```
int x;
string s;
matrix[2][2] m;
duple d;
vector v;

x = 10;
s = “hello”;
m = matrix_create([[1, 2], [3, 4]]);
d = (1, 2);
v = <1, 1>;

print_int x;
print_string s;
print_matrix m;
print_duple d;
print_vector v;
```

Grammar:

```
PRINTI expr_rule                    { PrintInt($2) }
PRINTS expr_rule                    { PrintStr($2) }
PRINTM id_rule                      { PrintMat($2) }
PRINTD expr_rule                    { PrintDup($2) }
PRINTV expr_rule                    { PrintVec($2) }
```

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[3][3] 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[3][3] board;
board = matrix_create([[0, 0, 0],[0, 1, 0],[0, 0, 0]]);
```

iii. 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)};
```

iv. Vector Expressions

Vectors have possible directions on row and column and store distance values. These expressions are used to facilitate the Piece movement through move function.

```
vector vector_1;
vector1 = <-2, 1>;
```

v. Duple Expressions

Duples have two integers that represent row and column locations of matrix. These expressions are used to indicate location in the matrix and facilitate the Piece movement through move function.

```
duple x;
x = (1, 1);
```

vi. 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) }
```

6. Grammar

i. Scanner

```
let digit = ['0'-'9']
let alpha = ['a'-'z' 'A'-'Z']
let int = digit+
let id = alpha (digit | alpha | '_')*
let string = '"' ((digit | alpha | '_' | ' ')* as s) '"'

let newline = '\n'

rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf }
| "//" { comment lexbuf}
| "vector" { VECTOR }
| "matrix" { MATRIX }
| "matrix_create" { MATRIX_C }
| "move" { MOVE }

| "bool" { BOOL }
| "int" { INT }
| "true" { BLIT(true) }
| "false" { BLIT(false) }

| "string" { STRING }
| "struct" { STRUCT }
| "tuple" { TUPLE }
| "duple" {DUPLE}

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

| "and" { AND }
| "or" { OR }
| "not" { NOT }

| "print_int" { PRINTI }
| "print_string" { PRINTS }
```

```
| "print_matrix" { PRINTM }
| "print_duple" { PRINTD }
| "print_vector" { PRINTV }

| '.' { DOT }

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

| '+' { PLUS }
| '-' { MINUS }
| '*' { MULTIPLY }
| '=' { ASSIGN }
| "==" { EQ }
| "!=" { NEQ }
| '<' { LT }
| "<=" { LEQ }
| '>' { GT }
| ">=" { GEQ }
| '%'  { MOD }

| digit+ as lem { INT_LITERAL(int_of_string lem) }
| string { STRING_LITERAL(s) }
| id as lem { ID(lem) }
| '"' { raise (Failure("unmatched quotation")) }
| eof { EOF }
|  _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

ii. Parser

```
%{ open Ast %}
%token DOT SEMI COLON COMMA LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token PLUS MINUS MULTIPLY MOD ASSIGN PRINTI PRINTS PRINTM PRINTD PRINTV
%token EQ NEQ LT LEQ GT GEQ AND OR NOT
%token IF ELSE WHILE CONTINUE BREAK BOOL INT STRING DUPLE STRUCT
%token VECTOR MATRIX MATRIX_C MOVE TUPLE
%token <int> INT_LITERAL
%token <bool> BLIT
%token <string> ID
%token <string> STRING_LITERAL

%token EOF

%right ASSIGN
%left NOT
%left OR
%left AND
%left EQ NEQ LT LEQ GT GEQ
%left MULTIPLY 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                           { Bind($1, $2) }
 | STRUCT ID LBRACE struct_def_list RBRACE SEMI   { StructDef($2, $4) }
```

```
typ_rule:
   INT         { Int  }
 | BOOL        { Bool }
 | STRING      { String }
 | MATRIX LBRACK INT_LITERAL RBRACK LBRACK INT_LITERAL RBRACK   { Matrix($3,
$6) }
 | 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,
Block([]))}
 | WHILE LPAREN expr_rule RPAREN stmt_rule               { While ($3,$5) }

rest_of_list_rule:
   INT_LITERAL COMMA rest_of_list_rule                { $1::$3 }
 | INT_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 RBRACK          { DupleAccess($1, $3) }
| ID LBRACK INT_LITERAL COMMA INT_LITERAL RBRACK { IndexAccess($1, $3, $5) }
| ID LBRACK id_rule RBRACK              { IndexAccessVar($1, $3) }

expr_rule:
   BLIT                                  { BoolLit $1 }
 | INT_LITERAL                           { IntLit $1 }
 | STRING_LITERAL                        { StringLit $1 }
 | id_rule                               { IdRule $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, Multi, $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, EqLess, $3) }
 | expr_rule GT expr_rule                { Binop ($1, Greater, $3) }
 | expr_rule GEQ expr_rule               { Binop ($1, EqGreater, $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) }
 | MINUS expr_rule                       { Unop (Neg, $2) }
 | NOT expr_rule                         { Unop (Not, $2) }
 | LT expr_rule COMMA expr_rule GT       { VectorCreate($2, $4) }
 | LPAREN expr_rule RPAREN               { $2 }
 | ID LBRACE struct_list RBRACE          { StructCreate($1, $3) }
 | MATRIX_C LPAREN matrix_rule RPAREN    { MatrixCreate($3) }
 | LPAREN expr_rule COMMA expr_rule RPAREN    { DupleCreate($2, $4) }
 | PRINTI expr_rule                    { PrintInt($2) }
 | PRINTS expr_rule                    { PrintStr($2) }
 | PRINTM id_rule                      { PrintMat($2) }
 | PRINTD expr_rule                    { PrintDup($2) }
 | PRINTV expr_rule                    { PrintVec($2) }
```

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