do fifty-two:

Language Reference Manual

Sinclair Target

Jayson Ng

Josephine Tirtanata

Yichi Liu

Yunfei Wang

1. Introduction

We propose a card game language targeted not at proficient programmers but at people who have never programmed before. For these people, programming a card game in something like C or Java would be prohibitively difficult, so it makes sense to give them an alternative language tailored to their inexperience and to the problem of creating a card game. The time they invest in learning this language is not time they might otherwise have spent coding, but rather an opportunity for them to grow more comfortable with the idiosyncrasies of computer programming. Therefore our language will be first and foremost a pedagogical tool. It will not try to compete as an “easy” option for a programmer who could code a card game in dozens of different ways. It will instead be a way to introduce students to programming as gently as possible in a familiar domain.

Our goals for the language then are as follows: *do fifty-two*, as we’ve chosen to call it, must be intuitive and high-level, so that students can grasp it without feelings of frustration or despair. But it must also be sufficiently related to more traditional programming languages to serve as a good stepping-stone to them.

2. Types

Primitive Types	**Meaning**
Number	Numbers are defined as a 64bit signed integer with a range of –2,147,483,648 to 2,147,483,647. Which is similar to the integer class found in other languages such as Java.
String	String is a sequence of characters surrounded by double quotes. A string is

	made of an array of characters.
Boolean	Booleans are defined by true and false keywords. Booleans are considered their own type, thus when an expression is defined with both a boolean operator and a non-boolean variable it will create an error.

Composite Types	**Meaning**
Card	A data type representing a card. **Fields:** Number rank Number suit String desc
Set	A data type representing an ordered collection of cards, whether that collection is supposed to be a deck, hand, or something else. **Fields:** Number size String desc Card top Card bottom
Player	A data type representing a player, or possibly a dealer. **Fields:** Set hand String Desc

Data in *do-fifty-two* is expressed in a finite and well defined set of data types. There are three primitive types in *do*, with several composite data types to facilitate the creation of card games. There are no floating point numbers because the rules of card games don't ever -- as far as we know -- call for fractional parts. Each of the composite data types has a field called *desc*, which contains a string description of the data it contains. When an instance of a composite data type is passed as an argument to the output function (see below), the *desc* field is what is printed.

New data types cannot be created, but the existing data types can be extended. New fields can be added to an existing type like so:

typename has *typeName* called *variableName*

A type extension is global and should not appear inside a function. Meaning that, all instances of the first data type have a filled of the second data type with the given name. For example,

```
Player has Number called score
```

would make the field *Player_score* available throughout a program.

3. Lexical Conventions

3.1 Identifier

An Identifier is a sequence of letters, digits or underscores. The first character must be alphabetic. An underscore ("_") is not considered to be alphabetic. Upper and lower case letters are considered to be different.

3.2 Keywords

The following identifiers are reserved for use as keywords and may not be used.

- if
- else if

- else
- while
- for
- break
- continue
- until
- true
- false
- do
- quit
- with
- output
- new
- configure
- setup
- int
- boolean
- String

3.3 Literals

Literals are values written in a conventional form with a fixed value.

An **Number Literal** consists of an optional minus sign, followed by one or more digits within the range of an Integer.

A **Boolean Literal** represents boolean values for true or false.

3.4 New Line

New lines are used to signify the end of a declaration, when preceded by the “\” token. In the latter case, the new line is ignored by the compiler. If there is no such token proceeds a new line, the compiler will treat the new line token being used to complete the declaration

3.5 Whitespace

Whitespace consists of any combination of *blanks* or *tab* characters. Whitespace is used to separate tokens and format programs.

3.6 Punctuations

Punctuators, are characters that have their own syntactic and semantic significance; they are not operators or identifiers

Punctuator	**Use**	**Example**
{}	Statement list delimiter	if (boolean) { statements }
()	Conditional parameter delimiter. (Expression precedence)	if (boolean)

3.7 Comments

The characters "//" introduce a single line comment. For example:

```
// Hello World !
```

3.8 Operators

do includes most standard operators found in any programming languages, but changes a few so that their meaning is more obvious, and adds two specific to card games. An operator is a token that creates an operation on at least one operand, which in turn yields a result. the assignment operator is not an equals sign, because assignment and tests for equality are distinct concepts in programming, yet that fact is often lost on beginner programmers who—and quite rightly—get confused by the equals sign. An added benefit here is that the actual equality operator can be a single equals sign as opposed to two.

Operator	**Meaning**
+.-. *,/	Standard arithmetic operators. Integer arithmetic only. Standard precedence.
= != <>>= <=	Standard relational operators, except that the equivalence operator is = not ==. Standard precedence.
— & !	Logical operators and the unary NOT

	operator. No bitwise operators.
:	Assignment operator.
-	Equivalent to the dot operator in many languages. Accesses a field within an object.
>><<	Prepend and append operators. Take a Set (see below) and a Card and either adds the card to the front or the back of the Set. If card is null, does nothing.
+	String Concatenation Operator

4. Control Flow Statements

do incorporates most of the common control-flow statements with the exception of *switch*. In addition, it incorporates an intuitive shorthand for expressing simple loops, which can be thought of as "multiplying" a series of statements by a number. The shorthand is redundant, because it could be replaced with a *for* loop. But in our experience many algorithms can be expressed using only the shorthand, so it might be introduced as a simpler form of looping to students not ready for the menacing syntax of a *for* loop, or for the bizarreness that is zero-based numbering.

4.1 The *if* Statement:

You can use the ***if* statement** to conditionally execute part of your program, based on the truth value of a given expression. Here is the general form of the ***if* statement**:

```
if (test1)
     then-statement1
else if (test2)
     then-statement2
else
     else-statement
```

If *test1* evaluates to true, then *then-statement1* is executed. if *test1* evaluates to false, but *test2* evaluates to true, then *then-statement2* is executed and neither *then-statement1* nor *else-statement* is. If *test1 and test2* evaluate to false, then *else-statement* is executed and *then-statement1 or then-statement2* is not. The *else if* clause and *else* clause are optional.
Example:

```
if (count = 5)
     count: count + 1
else if (count = 6)
     count: 12
else
     count: count – 1
```

If *count = 5* evaluates to true, then the statement *count: count + 1* is executed. If *count = 5* evaluates to false and count = 6 evaluates to true, then the statement *count: count + 1* is not executed, but the statement *count: count: 12* is executed. If neither *count = 5* nor *count = 6* is true, then the statement *count: count - 1* is executed.

4.2 The *while* Statement:

The ***while* statement** is a loop statement with an exit test at the beginning of the loop. Here is the general form of the ***while* statement**:

```
while (test)
     statement
```

The ***while* statement** first evaluates *test*. If *test* evaluates to true, *statement* is executed, and then *test* is evaluated again. *statement* continues to execute repeatedly as long as *test* is true after each execution of *statement*.

This example increments the integer from zero through nine:

```
new Number counter: 0
while (counter < 10)
     counter: counter + 1
```

4.3 The for Statement:

The ***for*** **statement** is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops. Here is the general form of the **for statement**:

```
for (initialize; test; step)
     statement
```

The ***for*** **statement** first evaluates the expression *initialize*. Then it evaluates the expression *test*. If *test* is false, then the loop ends and program control resumes after *statement*. Otherwise, if *test* is true, then *statement* is executed. Finally, *step* is evaluated, and the next iteration of the loop begins with evaluating *test* again.

Most often, *initialize* assigns values to one or more variables, which are generally used as counters, *test* compares those variables to a predefined expression, and *step* modifies those variables' values. Here is another example that prints the integers from zero through nine:

```
for (new Number x: 0; x < 10; x: x + 1)
   do output with "x = " + x
```

First, it evaluates *initialize*, which assigns *x* the value 0. Then, as long as *x* is less than 10, the value of *x* is printed (in the body of the loop). Then *x* is incremented in the *step* clause and the *test* re-evaluated.

All three of the expressions in a *for* statement are optional, and any combination of the three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression. You could also write the above example as:

```
new Number x: 1
for (; x <= 10; x: x + 1)
     do output with "x = " + x
```

In this example, *x* receives its value prior to the beginning of the ***for* statement.**
If you leave out the *test* expression, then the *for* statement is an infinite loop (unless you put a ***break* statement** somewhere in statement). This is like using 1 as *test*; it is never false.
This *for* statement starts printing numbers at 1 and then continues infinitely, always printing *x* incremented by 1:

```
for (new Number x: 1; ; x++)
     do output with "x = " + x
```

If you leave out the *step* expression, then no progress is made toward completing the loop—at least not as is normally expected with a *for* statement.
This example prints the number 1 over and over, infinitely:

```
for (new Number x: 1; x <= 10;)
     do output with "x = " + x
```

4.4 The *{ }* * *N* Statement:

The ***{ }* * *N* Statement** execute the statement(s) in the *{ }* for *N* times. The statements in *{ }* must be executable, and *N* must be a positive integer that is greater or equal to 1.
Example:

```
{
   new Number x: 1
   do output with "x = " + x
   } * 10
```

This statement executes the statements *new Number x: 1* and *do output with "x = " + x* for 10 times.

4.5 The *{ } until B* Statement:

The ***{ } until B* Statement** is similar to the C language statement *do...while…* . The *{ } until B* Statement executes the statement(s) in *{ }* repeatedly until the *B* statement evaluates true. Example:

```
new Number x: 1
 {
 x: x + 1
 do output with "x = " + x
 } until (x > 10)
```

This example declares a new *Number* type variable which is initialized to be 1, and execute the statements *x: x + 1* and *do output with "x = " + x* repeatedly until statement *x > 10* is true. Note that if *B* is an empty statement, or if *B* is not able to terminate the statements in *{ }* by its restriction(s), the statements in *{ }* will be executed infinitely.

4.6 The *break* Statement:

User can use the **break statement** to terminate a *while, for, { } * N* or *{ } until B* statement. Here is an example:

```
for (new Number x: 1; x <= 10; x: x + 1)
    {
      if (x = 8)
        break;
      else
        do output with x
    }
```

That example prints numbers from 1 to 7. When *x* is incremented to 8, x = 8 is true, so the ***break* statement** is executed, terminating the for loop prematurely.

If you put a ***break* statement** inside of a loop statement which itself is inside of a loop statement, the *break* only terminates the innermost loop statement.

4.7 The *continue* Statement:

User can use the ***continue* statement** in loops to terminate an iteration of the loop and begin the next iteration. Here is an example:

```
for (new Number x: 0; x < 100; x: x + 1)
     {
       if (x / 5 > 10)
         continue
       else
         do output with " x = " + x
     }
```

If you put a ***continue* statement** inside a loop which itself is inside a loop, then it affects only the innermost loop.

5. Procedures

In *do,* functions are called “procedures.” They are not called “functions” for reasons that will shortly become clear. They might have been called “methods,” except they are not associated with objects. They might also have been called “subroutines,” but the term “subroutine”, as I’m sure you’ll agree, is by this point rather passé.

A procedure must be defined in order for a call to the procedure to make sense, but the definition can come anywhere in a source file outside of a block. This means that a procedure does not have to be defined before it is called. Procedures are not first-class objects and they cannot be nested. They do not have to be—and in fact cannot be—declared.

A procedure consists of a header and a body. The header specifies the name of the procedure along with the types and names of its parameters. A header has the following syntax:

```
procedureName with Type parameterName and Type parameterName:
```

procedureName is the identifier associated with the procedure. `with` is a keyword that separates the identifier from the list of parameters, which consists of an arbitrary number of variable declarations separated by the keyword `and`. The colon terminates the header and begins the block that contains the body of the procedure. If a procedure takes no arguments, `with` can be omitted.

A procedure can be called anywhere within a source file. A procedure call looks like this:

```
do procedureName with argument1 and argument2
```

`do` is a keyword that signals a procedure call. The identifier *procedureName* tells the compiler which procedure to call. Everything after the keyword `with` is part of the list of arguments. The arguments are separated by the keyword `and` just as the parameters are in a procedure header. If the arguments passed to a procedure do not match the parameter types in the procedure header, the compiler will throw an error. Arguments are evaluated left to right. Again, if a procedure takes no arguments, `with` can be omitted.

You may have noticed that the syntax for a procedure header does not include its return type. That is because procedures cannot return anything. This is a quirk of *do*; there are no functions that map input to output. Procedures can only change the values of their parameters (which are passed by reference) and the values of global variables. They are not expressions. Conceptually, a procedure can only *do* something—it can only act on data to produce a change in state.

do is thus a hyper-imperative language. A *do* program consists mostly of statements with only a few expressions here and there. For large programs, this would become a nightmare. But while the programs remain simple, we believe that using procedures instead of functions is a good thing—it better conforms to the layperson's understanding of programming as "telling the computer to do things."

6. Expressions

6.1 Arithmetic Expressions

An arithmetic expression consists operands and operators. The operands can be integer constants or variable of the Number type. Only integer arithmetic are allowed. Example of expressions:

```
2 + 2
3 - 1
4 * 5
10/2
```

Expressions can be surrounded with parenthesis to indicate order of evaluation. For example:

```
2 * (4 + 5)
```

The expression `4 + 5` will be evaluated first, resulting in `9`. Then, `2 * 9` will be evaluated, resulting in `18`.

There are 4 types of arithmetic operators.

- `+`
  Adds the two operands together.
- `-`
  Subtract the right operand from the left operand.
- `*`
  Multiply the two operands together.

- `/`

  Divide the operand on the left by the operand on the right.

6.2 Assignment Expressions

The assignment operator `:` stores the value of its right operand in the variable specified by the left operand. Example:

```
Number n : 5 * 2
```

7. Operators

7.1 Relational operators

You can use relational operators to determine how two operands relate to one another. Do Fifty-Two supports 6 types of comparisons.

Relational Operator	Description	Example

`=`	Checks for equality	`if (a=3)`
		`  do evaluate`
		`else`
		`  quit`

`!=`	Checks for inequality	`if (a!=3)`
		`  do evaluate`
		`else`
		`  quit`

`<`	Checks whether left operand is smaller than the right operand	`if (a<3)`
		`  do evaluate`
		`else`
		`  quit`

>	Checks whether left operand is greater than right operand	*if (a>3)* *do evaluate* *else* *quit*
<=	Checks whether left operand is equal or smaller than right operand	*if (a<=3)* *do evaluate* *else* *quit*
>=	Checks whether left operand is equal or greater than right operand	*if (a>=3)* *do evaluate* *else* *quit*

7.1 Logical operators

Logical operators can be used to negate or combine relational expressions.

The logical conjunction operator & tests if two expressions are both true. If the first expression is false, the second expression is not evaluated and the entire expression becomes false.

```
if((a>3)&(a<5))
     do output with "a is 4"
```

The logical conjunction operator | test if at least one of the two expressions are true.

```
if((a=3)|(a=5))
     do output with "a is either 3 or 5"
```

The prepend operator `!` tests if the logical expression equates to false.

```
if(!(a=3))
      do output with "a is not 3"
```

7.2 Field Accessor Operator

The field accessor operator _ is attached after an object name and fetches the field whose name is specified on the right side of the underscore. It returns the value of the field. The type depends on the type that the field is set to initially. Example:

```
Card a : deck_top
```

The field accessor operator fetches the field named `top` in the object `deck`. Since the `deck` consists of `Card` types, this expression returns a `Card` type.

7.3 Prepend and Append Operators

The prepend and append operators >> and << respectively takes a `Card` and places it to a `Set`. In both cases, if a `Set` is taken as the source, the operators will simply take the top most `Card` from the `Set`.

The prepend operator >> takes the `Card` or the top most `Card` from the `Set` in the left operand and places it in the back of the `Set` in the right operand.

The append operator << takes the `Card` or the top most `Card` from the `Set` in the right operand and places it in the front of the `Set` in the left operand.

7.4 String Concatenation

The operator + when surrounded with strings will immediately concatenate the left and right string operands. Example:

```
do output with winner + "has lost"
```

8. Program Structure and Scope

8.1 Program Structure

A do-fifty-two program should be entirely self-contained, i.e. all the program code should be contained within a single file. No library or code import is supported.

A program should start with environmental variable assignments (by using `configure)` and type extensions if any, before any procedures are called or declared.

8.2 Scope

Variable declarations made at the top-level of a program (i.e., not within a procedure) are visible to the entire program, including from within procedures. Variable declarations made within procedures are visible only within those procedures. Variable declarations are not visible to the code that comes before them.

Procedure declarations can only happen at the top-level of a program. However, procedures can be called before their declarations.

Also, environmental variables and type extensions are global.

9. Sample Program

War! is a simple card game often played between children. The rules can be found here. What follows is an implementation of War! in do that showcases most of the features of the language.

```
// war in do
configure playerCount: 2
configure acesHigh: false

new Number warCount: 0

Player has Set called table  // player_table available

setup:
     // deal cards
     { player1_hand << deck_top } * (deck_size / 2) // loop
     { player2_hand << deck_top } * (deck_size / 2) // loop

round:
     do turn with player: player1
     do turn with player: player2
     do output with "Player 1 played: " + player1_table_top
     do output with "Player 2 played: " + player2_table_top
     do evaluate

turn with Player player:
     do output with player + "'s turn."
     if (player_hand_size = 0)
          if (player = player1)
               do output player + " has lost. Player2 wins!"
```

```
                do quit
          else
                do output player + " has lost. Player1 wins!"
                do quit

     do output with "Play card?"
     do input with new String in
     if (in = "y")
          player_hand_top >> player_table
     else
          do output with player + " has decided not to play\
                anymore.” // no backslash needed here
          do quit

evaluate:
     if (player1_table_top > player2_table_top)
          do output with "Player 1's card is higher."
          { player1_hand << player1_table_top } \
                * player1_table_size
          { player1_hand << player2_table_top } \
                * player2_table_size
     else if (player1_table_top < player2_table_top)
          do output with "Player 2's card is higher."
          { player2_hand << player1_table_top } \
                * player1_table_size
          { player2_hand << player2_table_top } \
                * player2_table_size
     else
          do output with "It's a tie. That means WAR!"
```

```
        warCount: warCount + 1

    // if a set runs out of cards >> and << won't do anything
        { player1_hand_top >> player1_table } * 4
        { player2_hand_top >> player2_table } * 4
        do output with "Player 1 and 2 put down 4 cards."
        do evaluate
```