do fifty-two:
An Introductory Programming Language

Sinclair Target
Yichi Liu
Josephine Tirtanata
Jayson K Ng
Yunfei Wang

September 23, 2014

Project Group

Sinclair Target / wst2109 / Language Guru
Yichi Liu / yl3057 / System Integrator
Josephine Tirtanata / jt2706 / System Architect
Jayson K Ng / jkn2121 / Verification & Validation
Yunfei Wang / yw2577 / Manager

Motivation

Card game languages have been a popular choice among past PLT students. In 2008, one group created a language called PCGSL, for Playing Card Game Simulation Language. In 2009, another group created a language called AIC, for All In the Cards. And in 2012, a third group created a language called Cardigan, which may or may not have been well implemented relative to the others but certainly has the best name of the three.

Each of these teams tried to simplify the process of programming a card game by creating a domain-specific language that abstracts away the details common to all card games. But the problem, as this team sees it, is that these languages have in most cases hewed too closely to traditional C-style syntax to be useful. Only people who have already done some programming in a language like C or Java could learn the language in a reasonable amount of time, yet those same people are also capable of programming a card game quite easily by themselves, because programming a card game isnt that difficult. If youre using an object-oriented language, all you really need is a Card class with an integer for rank

and maybe a string for suit. Put some instances of that class in an array and youre ready to go. The effort that a language like PCGSL or Cardigan might save is small, and that saved effort is likely going to be offset by the effort required to learn the language in the first place.

Proposal

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 weve 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.

Specification

do fifty-two is an imperative, procedural, statically typed language. In that sense, it is like C and its many progeny in the fundamentals of how it works; a *do* program consists of a series of statements organized into procedures that act upon variables. But the syntax of the language attempts to be less cryptic and more intuitive to the average non-programmer, while simultaneously incorporating several elements that make programming card games a breeze.

operators

do includes most of the standard operators found in any programming language, but changes a few so that their meaning is more obvious, and adds two specific

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

to card games. In particular, 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.

Built-in Types

Type	Meaning
Number	Integer
String	String
Boolean	Boolean
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

There are really only three primitive types in *do*, along with a few composite data types meant 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 gets 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 must not appear inside a function. It means that all instances of the first data type have a field 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.

Control Flow

Control-flow Statement	**Meaning**
if, else if, else	Standard conditional statements.
while, for	Standard loop statements.
{ } * N	Simple loop, executes the statement(s) within the curly braces *N* times.
{ } until B	Conditional loop. Like a negative do-while.
break, continue	Standard control 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 though 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 intimidating syntax of a *for* loop, or for the bizarreness that is zero–based numbering.

Blocks in *do* are organized according to the indentation rule, like in Python. Multiline statements can be written by using a backslash. Curly braces are used only for the shorthand loop syntax discussed above.

Procedure Syntax

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, and they might have been called "subroutines", but the term "subroutine" is intimidating and stuffy.

In *do*, a function is declared like this:

```
do procedureName with Type parameterName
```

Additional parameters can be added by using the *and* keyword. A function is called like this:

```
do procedureName with argumentName
```

Again, arguments can be added by using the *and* keyword. If a procedure takes no arguments, the procedure declaration and call consists only of the *do* keyword and the procedure name. Procedures do not have to be declared before

they are used.

A quick of *do* is that procedures cannot return a value. This is why they are not "functions"; they don't map input to output. All arguments are passed by reference, and the procedure must change those arguments if it is to return anything. Procedures therefore cannot be used as expressions. Conceptually, a procedure can only *do* something—it can only act on certain data to produce a change in state. We believe that this is easier to understand than a function that can return a value *and* change its arguments. The alternative would have been to disallow pass by reference, but then no procedure could return more than a single value.

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 thingit better conforms to the laypersons understanding of programming as "telling the computer to do things".

Remaining Keywords and the *do* Environment

Variable are declared in *do* according to the following syntax:

```
new typeName variableName:  value
```

Declared variables are not automatically initialized to anything—you must assign a value to a variable when you declare it. Variables are either local to a function or global.

The compiler automatically initializes a small set of environmental variables that define properties of the game being played like the number of players or whether aces are high. There is also a variable that represents the deck and one for every participating player. These variables can be reassigned using the `configure` keyword, like so:

```
configure variableName:  value
```

A configure assignment is also global. You cannot have a configure assignment within a function.

Finally, the functions input and output can be called anywhere in a program. Input reads until a newline character and interprets what it reads according to the type of its argument, where it then stores the input data. Output takes a string and prints it to the console.

An Example Program: War!

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
```