SPARK-PL: Introduction

Alexey Solovyev

Abstract

All basic elements of SPARK-PL are introduced.

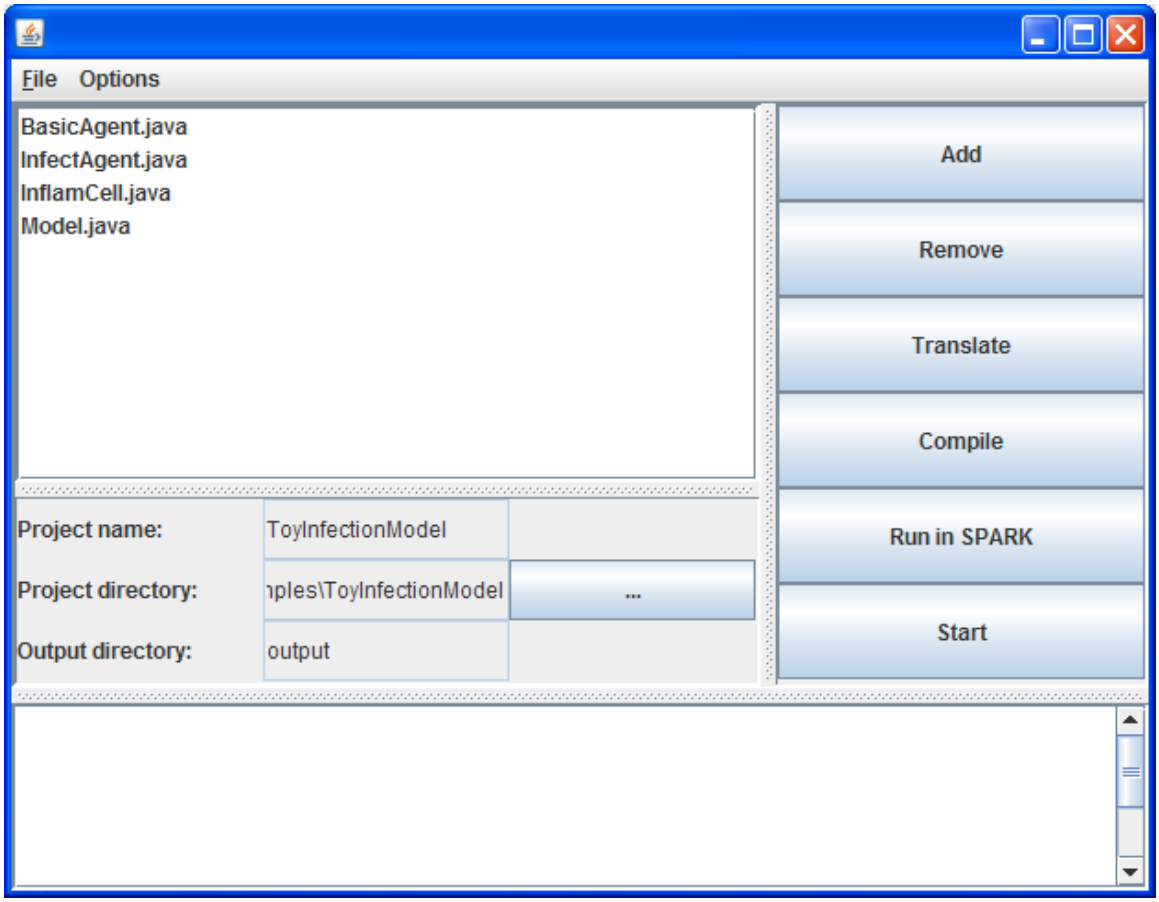
Table of Contents

1. Introduction to SPARK-PL

SPARK-PL (SPARK Programming Language) is a special language for implementing SPARK models. Its syntax is derived primary from Logo programming language (influenced by NetLogo) and Java programming language. All models written in SPARK-PL are translated into Java source code first, and then a Java compiler is used to produce the machine code which can be executed by the SPARK simulation engine. SPARK-PL has static type system which helps to eliminate many programmers' errors concerning an incorrect use of variables of incompatible types. Type inference mechanism of SPARK-PL makes it easy to implement models in the way similar to languages with a dynamic type system. With type inference, it is not required to provide an explicit type for every new variable. The translator is capable to find the correct type in many cases automatically by looking at the expressions where variables are used.

Every SPARK model written in SPARK-PL consists of several SPARK-PL source code files. These files constitute a SPARK-PL project. SPARK-PL projects are managed by a special program called the SPARK-PL Project Manager.

Figure 1. The SPARK-PL Project Manager



The list on the left shows all files in a project. New files can be added by clicking 'Add' button which brings in an open file dialog. Button '…' allows to choose another model directory. Projects can be saved via 'File' menu 'Save project...' command. The button 'Translate' is used for translation a SPARK-PL source code into a Java source code which then can be compiled by clicking the 'Compile' button. 'Run in SPARK button' runs the compiled model in SPARK. 'Start' button is a shortcut for all three main actions: translate, compile, and then run in SPARK.

Step by step instruction on creating new SPARK-PL projects. First, create SPARK-PL source code files in any plain text editor. Save them in a folder which will be a main project folder. Click '...' button in the project manager and select the project folder. Add source code files into the manager using 'Add' button. Save the project description file by selecting 'Save project...' item from the 'File' menu. The project description file is a simple xml file which contains the information about all files inside the project. Note that this file should be in the same project folder as source code files. Click 'Translate' button. If any error occurs, correct it and click 'Translate' again. After elimination of all translation errors and after successful translation, click 'Compile' button. Again, there could be some compilation errors which also can be corrected. After a successful compilation process, click 'Run in SPARK' button to run a model in SPARK. It is always possible to use only 'Start' button instead of three other buttons. But for the first time, it is better to use 'Translate', 'Compile', 'Run in SPARK' sequence in order to simplify the error correction process.

To open an existing project description file, select 'Open project...' item from the 'File' menu. Also, 10 most recently open projects are available directly in the 'File' menu. If you later add some new files to an existing project, do not forget to save the description file again using 'Save project...' command.

2. Alphabet of SPARK-PL

Valid symbols in SPARK-PL source code files are all Latin letters, digits, period '.', colon ':', comma ',', square brackets '[' and ']', parentheses '(' and ')', and symbols '_', '^', '+', '-', '*', '/', '=', '%', '<', '>', '!'. Semicolon ';' denotes a comment: everything after a semicolon is not parsed by a SPARK-PL translator and treated as a comment.

Type names, variable names, command names are identifiers. Identifiers start from a letter and can include any letters, digits, and dashes '-'. All identifiers are case sensitive.

```
; A comment

; A variable named 'some-variable' is declared and
; initialized
var some-variable = 20.2

; This is a different variable
var Some-variable = 10.123e-10

; This is the third variable
var SomeVariable = "string"
```

SPARK-PL has many similarities with languages from the Logo family. That's why identifiers can include the dash symbol. Because of that, an attention should be paid to distinguish dash and minus. The rule is simple: minus should be always separated by spaces.

3. Types and variables

All SPARK-PL source files should begin with one of the following declarations

```
class class-name : parent-type
agent agent-name : parent-type
model model-name : parent-type
```

In fact, everything in SPARK-PL is encapsulated inside classes (or types). 'agent' and 'model' are special types which will be described later. All types has a name and a parent type from which they inherit functions and properties. Usual classes and models are not required to have a parent type. For agents a parent type is required (see the agent tutorial for details). After a new type declaration on the first line of a source file, a type definition follows. Everything written after a type declaration is considered to be included inside that type. But it is also possible to include several type declarations in one source file. To do so, just start a new type declaration in any place of a source file. Then everything before that new declaration belongs to the previously declared type, and everything after will belong to a new type.

```
class Class1

; definition of Class1

; Declare a new type which has the previous
; type as its parent type
class Class2 : Class1
```

```
; definition of Class2

; One more type
class Class3

; definition of Class3
```

Note that each SPARK model must contain one and only one model class.

Several types (classes) can be defined in one file, but it is also possible to split a declaration of one class into several files. It can be achieved by adding the 'partial' keyword before declaring the corresponding type:

```
; Inside file1
partial class Class1

; definition of Class1

partial class Class2 : Class1

; definition of Class2

; Inside file2
partial class Class1

; The parent type should be the same for all partial declarations
partial class Class2 : Class1
```

Models, agents, and classes can be declared with the 'partial' keyword. It is always recommended to completely declare one type in one file unless there is a definite benefit of using partial declarations. One possible usage of partial declarations is to split the main model into two files: one file contains all model functions, another file contains all global variables and parameters. In this way it is simpler to add new parameters and to manage global variables.

SPARK-PL consists of commands, control structures and declarations. There are many globally defined commands, for example 'ask', 'create', 'random', 'random-vector', etc. Commands defined inside classes are called methods. Variables defined inside classes are called fields. Declaration of a variable looks like the following

```
var x : number
```

Here 'var' is a keyword, 'number' is a type. A type name can be omitted. In that case a variable is treated as a variable of unknown type. It is possible that the type will be determined automatically later. For example, if we have a code

```
var x
x = 2
```

Then after the assignment the type of x becomes 'number'. On the other hand, there are many situations when the type cannot be determined automatically (especially in the current implementation of the type inferring algorithm). Because of this, it is recommended to explicitly specify types of all fields and method arguments. It is possible to initialize a variable during its declaration

```
var x = 3
```

There is another keyword 'global' for declaring global variables. These variables are available not only in the type where they are declared but also in other types. And you only need to know their name to access them in any part of a code. Global variables can be declared inside any class or agent, but it is a good practice to declare global variables only inside model types.

```
model Model

; x will be available in any part of a code
global x = 30
```

Methods are declared as follows

```
to method-name [arg1:type arg2:type] : method-type
; Method's body
end
```

If there are no arguments in a method then you may omit square brackets for arguments. All explicit type declarations are unnecessary. A return type of a method, 'method-type', can also be omitted. In that case a method will have no return type. If a method has a return type then its last command should be 'return some-value' where 'some-value' has the same type as method's type.

```
to add-numbers [a b] : number
[
 var result = a + b
 return result
]
```

There is one very important rule associated with SPARK-PL commands and methods. If a command returns some value then it is required to use parentheses for all its arguments which have complex structure (not a single identifier or a constant). For commands without return value it is not required. It seems to be confusing and not useful but it helps to reduce the number of parentheses in a program. In any case, it is always better to enter additional parentheses in order to avoid unexpected compile errors.

```
agent SomeAgent : SpaceAgent

to some-method
 var v : vector
 ; The next command is translated as
 ; var x = (random 100) + 20
 ; because 'random' returns a value so all its arguments
 ; must be inside parentheses of have a simple structure
 var x = random 100 + 20

 v.x = x
 x = random (100 + x)
 v.y = y

 ; The next command is translated as
 ; move-to (v + (x * v))
 ; because 'move-to' does not return any value.
 move-to v + x * v
end
```

Every SPARK model should have the main model class. This class is declared using 'model' keyword. The minimal working SPARK model written in SPARK-PL may look as follows

```
model Model

space StandardSpace -10 10 -10 10 true true
```

Right now you may ignore the second line where the model space is declared. You will learn about spaces later. Now you only need to know that it is a requirement to declare a space.

This minimal model is not useful. To make it more interesting, several methods should be added. The first method to add is the 'setup' method. This method is called every time when a model is initialized. Note that if you have an explicit initialization for a global variable then this initialization will be processed each time the 'setup' method is called. There are other two special methods: 'begin-step' and 'end-step' inside the main model class. The method 'begin-step' is called before each simulation step, the method 'end-step' is called after each simulation step. The precise declarations of these methods are the following

```
model Model

space StandardSpace -10 10 -10 10 true true

to setup
 ; Initialize a model
end

to begin-step [tick] : boolean
end

to end-step [tick] : boolean
end
```

The return type 'boolean' is not required for 'end-step' and 'begin-step' methods. It will be added automatically. Moreover, you don't need to write a return command. By default, these methods return false value. If any of these methods returns true value, then a simulation will be stopped. So these methods can control a simulation process and stop it if necessary.

```
to begin-step [tick]
 ; Automatically stops a simulation after 1000 steps
 if tick >= 1000
  [ return true ]
end
```

Also, it is possible to omit 'tick' argument for these two methods. This argument counts the number of steps passed from the beginning of a simulation process. If you don't supply this argument, then it will be automatically created.

4. SPARK-PL basic commands

The most important command which controls a program flow is 'if'. It has two arguments: a boolean value and a special argument representing a block of a code. This block of code is executed only if the argument is true. An example clarifies everything.

```
var a = 3

if a > 2
[
 do-something
]
```

There is a modification of this command 'ifelse' which has an additional code block argument which is executed when the condition (the first argument) is false.

```
ifelse a > 2
[
 do-something
]
[
 do-something-else
]
```

The command 'repeat' repeats a block of commands several times.

```
var n = 100

; Do something 100 times
repeat n
[
 do-something
]
```

The command 'while' repeatedly executes a given block of commands meanwhile a condition is true.

```
var n = 100

while n > 0
[
 do-something
 n = n - 1
]
```

There is a very special command (a keyword) 'for' which is used for iterating some value. It has the following syntax

```
for var-name = first-value : step : last-value
[
 do-something
 ; var-name is available here as a numerical variable
]
```

Here 'var-name' is any identifier. The only condition is that there is no local variable with the same name. 'first-value', 'step', and 'last-value' are numbers (or numerical expressions). 'first-value' specifies the initial value of the variable 'var-name'. 'step' specifies an increment of that variable. 'last-value' specify when to stop the iterations. Iterations will stop when the value of 'var-name' becomes greater than 'last-value'. 'step' is an optional argument and can be omitted.

```
; Prints out numbers from 1 to 100
for i = 1 : 100
[
 print i
]
```

If a user method in SPARK-PL has return type then the command 'return value' can be used to exit from this method at any point. For methods without return type this command is not available because it always requires an argument. In that case, the command 'exit' can be used.

There are two commands which control the iterative processes 'while', 'repeat', and 'for'. The command 'continue' inside an iteration code block tells to stop the current iteration immediately and start the next iteration. The command 'break' stops all iterations immediately and the commands after an iteration process are executed.

5. SPARK-PL basic types

5.1. number

The most basic type is 'number' (or 'double'). This type represents a numerical value. It corresponds to a double type in many other programming languages. So it can be used for representing both integers and floating point numbers.

By default, all numerical variables are initialized with 0. All usual arithmetical operations are available for numbers: +, - (binary and unary), *, /. The operator '%' finds a residual of a division. The operator '^' raises its left hand side to the power from the right hand size. There are several operators which combines the assignment with an arithmetic operation: +=, -=, *=, /=. These operators are explained in the following example.

```
var x = 20
var y = 30

; Equivalent to
; x = x + 1
x += 1

; Equivalent to
; x = x * y
x *= y
```

There are all usual mathematical functions in SPARK-PL like 'sin', 'cos', 'exp'. Commands 'floor', 'ceil', and 'round' are used for rounding a number (see SPARK-PL dictionary for a full description). The command 'random a' returns a random number uniformly distributed in the interval [0,a) (the returned value is always less than a). The command 'random-in-interval a b' returns a uniformly distributed random number in the interval [a, b).

5.2. boolean

The next basic type is 'boolean' (or 'bool'). This is a logical type. Variables of this type have two values: true or false. The default value is false. Operators for boolean values are: 'and', 'or', 'not'. The meaning of these operators should be clear. There are several comparison operators for numbers which yield a

boolean result: <, >, <=, >=, ==, !=. The last two operators mean equality and inequality respectively. Logic operators are often used in 'if' command for combining different conditions.

```
if (x > 30 and x < 100) or y == 1
[
 do-something x
]
```

5.3. string

Strings in SPARK-PL are represented by the 'string' type. Constants of this type should be always written inside double quotes. There is a concatenation operation for strings represented by the '+' operator. Note: operations and functions in SPARK-PL can be overloaded. It means that the same operator or the same function name can correspond to several actual functions. The right function is choosen based on the types of arguments. An example of a command taking a string argument is 'print'. This command prints out its argument in the standard output stream. Also this command can be used with numerical arguments to print out numbers (or with other types to print out information about that types).

```
to do-something [x]
 var str1 = "hello,"
 var str2 = " world"
 print str1 + str2
 print x
end
```

5.4. vector

The next type is 'vector'. It is more advanced type representing a 3-d (or 2-d) vector. The constants of this type has the form '[a, b, c]' where a, b, c are some numerical constants (in the current SPARK-PL implementation it is not allowed to have variables inside vector constants). There is a command 'create-vector' with three arguments which creates a new vector with the given entries. For this command any arguments can be used. Also, vectors has all usual arithmetic operations: +, - for vectors, * for a vector and a number, / for a vector on the left and a number on the right. Also shortcuts +=, -=, *=, /= work for vectors (last two operators expect numbers on the right hand side). It is also possible to compare two vectors using '==' and '!=' operators.

```
; All vectors initialized by a zero vector by default
var v1 : vector
var v2 = [1, 2, 3.3]

; v1 = [3, 6, 9.9]
v1 = v2 * 3
v2 -= v1 + v2 * 4

var v3 = create-vector v1.x (v2.y + 3) v1.y
```

The vector type has special methods. The method 'length' returns the length of a vector. The method 'normalize' normalizes a vector (makes it of the unit length) and returns the result of the normalization (it modifies the vector which called 'normalize' and returns a reference to the same vector after modification). There are three fields in each vector: 'x', 'y', 'z' which represent the corresponding vector entries.

```
; After the normalization we get v1 == v2
v2 = v1.normalize

; Due to numerical errors it cannot be guaranteed that
; v1.length == 1 exactly
v2.x = v1.length
```

5.5. Arrays and lists

All basic types discussed so far were primitive SPARK-PL types. Besides primitive types there are composite types. The most important composite types are arrays and lists. In SPARK-PL it is often not important whether a given object has an array or a list type because there are commands which work with both types in the same way. In general, lists are more flexible, meanwhile arrays are more efficient. These types are used for storing several objects of another type in the same variable. To declare a variable which has a list type, use the following syntax

```
class TestClass

; A list of vectors
var x : List<vector>

; An array of strings
var strings : Array<string>

; A list of TestClass objects
var tests : List<TestClass>
```

Due to limitations of the current version of SPARK-PL, it is not possible to use 'number' and 'bool' as a subtype of composite types. Also, composite types cannot be used as arguments of other composite types. If you need a list of numbers, then there are two simple solutions. One is to create a list of vectors and then use only one component of each vector. Another solution is to create a custom class 'NumberClass' with one numeric field and use this class as the argument of composite types. The same trick works for creating composite types which include other composite types.

Another limitation of the current version of SPARK is that it is not possible to use composite types as return types of methods.

The command 'count' returns the number of objects inside a list or an array. The command 'create' creates an array of objects. This command has two arguments: a type name of objects to be created and a number of objects in the new array. This operation cannot be used for numbers and boolean values. All objects in a created array are initialized by their default values. To get a particular element of an array (or a list) use the 'get' command. It has two arguments: an array or a list and the index of an element there (indices start from 0). Lists are created with the command 'create-list' which has one argument: a name of the subtype. Elements can be added to a list with 'add' command, removed from a list with 'remove' command. See the dictionary for all available list and array commands.

```
; Creates an array with 100 vectors
var array = create vector 100

; Get the 4-th element in the array
var v = get array 3
```

```
; Create a list of strings
var list = create-list string

; Add several strings to the list
list.add "first"
list.add "second"

; Remove an element from the list
list.remove "first"

; Prints "second"
print (get list 0)
```

The most useful command which can be used with arrays and lists is the 'ask' command. This command does several things. To understand it completely, first look at another application of the 'ask' command. It can be used with any object (not with a number or a boolean value) as its argument. When a method is executed there is always one implicit variable named 'self'. This variable refers to an object for which the method is executed. Very often it is used implicitly.

```
class SomeClass

to method1
end

to method2
 ; The next command will be translated as
 ; self.method1
 ; that is, the method 'method1' is called
 ; for the current active object.
 method1
end
```

This 'self' variable can be always used explicitly. The command 'ask' with an object argument temporary changes this 'self' reference to its argument.

```
to method
 var v : vector

 v.x = 1
 v.y = 2
 v.z = 3

 ; The same things in another way
 ask v
 [
  ; Inside this block 'self' reference is set
  ; to the vector 'v', so its fields x, y, z
  ; can be accessed directly without
  ; an explicit reference to the object 'v'.
  x = 1
  y = 2
  z = 3
 ]
```

```
end
```

Now it is easy to explain the effect of the 'ask' command applied to an array (or a list). First of all, it iterates through all objects inside an array, and during each iteration it assigns the 'self' reference to the current object in the array and performs all commands inside the code block for that object.

```
to method
 var array = create vector 100

 var i = 0

 ask array
 [
  x = i
  y = 2 * i
  z = 3 * i

  i += 1
 ]
end
```

Sometimes it is required to refer to an object which started the 'ask' command inside the 'ask' command block (remember that inside that block the 'self' reference is different). In order to do so, use 'myself' reference. This reference always refers to an object which started the 'ask' command. Furthermore, 'ask' commands can be nested in each other. There is also the reference 'this' which returns a reference to the object for which the current method is executed. Another reference is 'parent' which returns a reference to the parent object of 'this'. The main purpose of the 'parent' reference is to call a parent class method that has the same name as a method in a child class from the child class.

```
class Parent

var x : number

to function
end

class Child : Parent

var x : vector

to function
 parent.function

 ask create vector 100
 [
  ; This refers to vector's x component
  x = 2

  ; This refers to a variable x defined in Child
  myself.x = [1, 2, 3]

  ; This refers to a variable x defined in Parent
  parent.x = 2
```

```
  ; myself.x is a vector
  ask myself.x
  [
   ; Now myself refers to a vector
   ; from the previous 'ask' code block
   myself.x = 1

   ; Again we refer to Child.x
   this.x = [3, 4, 5]
  ]
 ]
end
```

The last note is about visibility of variables inside 'ask' blocks. All local variables are visible inside 'ask' blocks. If there is a field in the object referred by 'self' with the same name as a local variable, then the local variable always shadows the field with the same name. Here is an example:

```
to some-method
 var x = 1
 var v : vector
 v.x = 2

 ask v
 [
  ; Prints 1
  print x

  ; Prints 2
  print self.x
 ]
end
```