COMS W4115

Language Reference Manual

Vence Stanev (UIN: vs2226)

Turn based simulation language TBSL

Abstract

The turn based simulation language (TBSL) is a functional language that enables programmers to describe a current state of a system comprised of objects. The goal of TBSL is to run that simulation for a number of turns in order to examine the effects of particular phenomena on the system.

Applications

Among other things, TBSL can be used to describe a group of business entities with different strategies and observe the effect over time.

1. Lexical Conventions

1.1 Identifiers - An identifier is a sequence of letters, digits and the underscore character. Each identifier starts with a letter. Identifiers are case sensitive - upper and lower case letters are considered different.

1.2 Comments – Comments are introduced with the opening character sequence /* and closed with the sequence */. Comments cannot be nested - the characters /* introduce a comment, which terminates with the first occurrence of the characters */.

1.3 Keywords - Keywords are identifiers that are reserved words in TBSL. They have specific function and cannot be used as regular identifiers.

Init – initialize an object

Relation - define a relation

Func – define a function

List – define a list of “Objects”

Turns – makes the simulation go to the next turn

1.4 Operators

1.5 Punctuation

Punctuation	Use	Example
/* */	Comments	/* This is a comment */
" "	String constant	"This is a string"
;	Indicates the end of a statement	Compare (a,b);
,	Argument list separator	Compare (a,b);
()	Argument list delimiter	Compare (a,b);

{}	Function body or block of statements	Func Compare (a,b)
		{
		Body of function here
		}

->	Reference a variable attribute	a->cost

1.6 Constants – constants are used to initialize variable attributes.

1.6.1 Integer constants – integer constants are represented with whole numbers in decimal format. An integer constant constitutes only of digits; decimal point and exponent are not allowed. A unary – operator is allowed. An example of an integer constant is 4 or 6000 or 12. The system stores all numbers as floating point numbers so each integer constant is implicitly converted to a float.

1.6.2 Floating point constants – floating point constants are represented with a whole part, a decimal point and a fractional part. The whole part and the fractional part are made up only of digits. A unary – operator is allowed. An example of a floating point constant is 5.3 or 0.12345.

1.6.3 String constants – string constants are made up of a sequence of characters that are enclosed in quotes. For example "this is a string" or "5" or "Some characters @#$%^&(".

2. Basic types - TBSL has only one basic type, which is called "Object". No notion of type conversion is defined. TBSL also supports lists of "Objects".

2.1 "Object" type - When declaring a variable, type is not specified but the variable needs to be initialized. A variable is initialized by providing a custom list of attributes, which is a list of tuples, each tuple being a name\value pair. The name is always a string and the value can be an int, float or string. Defining a second variable with the same name in the same scope is not allowed. A variable has no predefined attributes. Attributes are all custom and could be added at initialization time as well as later in the program.

Syntax example:

```
Init a (("status","active"), ("cost", 5.7), ("ValueAddPerTurn",10));
```

This syntax initializes the variable a.

Syntax example:

```
Attribute(a, (“cost”, 5.7));
```

This syntax will add the “cost” attribute to the “a” variable if the attribute doesn’t already exist and it will update it if it does.

2.2 Reference a variable attribute – A variable attribute could be referenced by providing the following syntax:

Syntax example:

```
a->cost
```

2.3 List of “Objects” – TBSL supports grouping of variables in a list.

Syntax example:

```
List ObjList;  ObjList.Append(a); ObjList.Prepend(a); ObjList.Remove(a);
```

3. Operators - Operators in TBSL are tokens that allow for particular operations on data. The standard Math operators are available (i.e. +, -, *,/) as well as the logical operators AND and OR (i.e. &,|). In addition the brackets operator (i.e. ( )) is also available. These operators are defined for variable attributes and are ranked by precedence.

Syntax example:

```
Init a ((“status”,”active”), (“cost”, 5.7), (“ValueAddPerTurn”,10));

Init b ((“status”,”inactive”), (“cost”, 4.0), (“ValueAddPerTurn”,12));

/* Addition*/

Attribute (a, (“cost”, a->cost+3));

/* concatenation */

Attribute (a, (“cost”, “foo” +”bar”));
```

4. Syntactic constructs – TBSL supports the following control constructs

4.1. <u>If than else</u> – conditional control logic

Syntax example:

```
If ( a->cost >3) then
    Attribute (a, (“cost”, 1003));
Else
    Attribute (a, (“cost”, a->cost+1));
```

4.2. <u>Loops</u>

Syntax example:

```
Attribute (a, (“cost”, 0));

While(a->cost <10)
 {

        Attribute (a, (“cost”, a->cost+1));

 }
```

5. Functions -TBSL supports functions in order to promote modularity. A function is a collection of statements that are given a name. Functions in TBSL do not have a return type; all parameters are “passed by reference” and the outcome of the function is reflected directly on the input.

Syntax example:

```
Func MyFunciton (ListOfObjects)

{

        Init a ((“status”,”active”), (“cost”, 5.7), (“ValueAddPerTurn”,10));

        Init b ((“status”,”inactive”), (“cost”, 4.0), (“ValueAddPerTurn”,12));

        ListOfObjects.Append(a);

        ListOfObjects.Append(b);

}
```

6. Scope – TBSL supports the notion of scope by defining blocks of code much like C and Java do. A block of code is defined by wrapping it in {}.

7. Example of an Algorithm

```
Init simulation ((“turns”,10)(“turnDecrement”,1));

Init store_a ((“status”,”active”), (“balance”, 7.2), (“ValueAdd”,10));

Init store_b ((“status”,”inactive”), (“balance”, 4.0), (“ValueAdd”,12));

While (simulation->turns >0)

{

  Attribute (store_a, (“balance”, store_a->balance+store_a->ValueAdd));

  Attribute (store_b, (“balance”, store_b->balance+store_b->ValueAdd));

  Attribute (simulation, (“turns”, simulation->turns – simulation->turnsDecrement));

}

/* Prints all attributes of the object */

Print(store_a);

Print(store_b);
```