COMS W4115 Programming Languages and Translators

Sim2D Language Reference Manual

David Suess
dcs2136@columbia.edu

June 20, 2008

Introduction

Sim2D is a language designed for simulation of the movement of objects on a 2 dimensional map. A programmer may define objects and their behavior. The interpreter will present the simulation on a 2 dimensional map for observation of the interaction of the objects. A program consists of one or more object definition blocks followed by one or more rule definition blocks to define custom behavior. Possible applications of this language would be to simulate the traffic of air, land and sea vehicles, or the interactions of animals in an environment.

1. Lexical Conventions

1.1 Comments

Single line comments begin with the // characters and end at the end of line. Multi line comments begin with the characters /* and end with the */ characters.

1.2 Identifiers

Identifiers represent the names of user defined variables and functions. Identifiers consist of one or more characters and must begin with one of the characters a to z upper or lower case. After the first character, the identifier may consist of characters upper or lower case a to z, digits 0 to 9 and the underscore character. Recognition of an identifier is case sensitive.

1.3 Keywords

Keywords are reserved words that may not be used as identifiers. They are:

rule	**destroy**	**speed**	**heading**	**visible**	**distance_to**
bearing_to	**x**	**y**	**if**	**else**	**true**
false	**string**	**float**	**object**	**integer**	

1.4 Constants

Integer literals can take the form of a sequence of digits preceded by an optional – character to indicate a negative number. All integers are considered base 10.

Floating point literals take the form -123.456e-38 where the -123 characters and e-38 characters are optional. The – in the exponent is also an optional part of the exponent notation.

Strings begin with a “ character and end with a “ character. A “” sequence will denote a “ character inside the string.

Booleans can be assigned and compared with the *true* and *false* keywords.

1.5 Operators

Operators are used for expressions, assignments and object dereferencing.

.	Dereference fields of an object
=, !=	Object, string or boolean comparison
=, <, >, <=, >=, !=	Integer or float comparison
<-	assignment
and, or	boolean logic

1.6 Separators

Separators are used to distinguish arguments, expressions, statements, object blocks, rule blocks and function blocks.

{ } ( ) ;

1.7 Whitespace

No whitespace characters will have any significance to the language other than to separate the tokens of the language and to improve the readability of the source code.

2. Types

Supported types will be integer, float, boolean, string and object.

integer is a 32 bit integer
float is a floating point number

string is sequences of characters

2.1 Object

The object type is a data structure that may take on as many user defined fields of **integer**, **float**, **boolean** and **string** types desired. In addition, an object will always contain the integer fields **x**, **y**, **speed**, **heading**, and the **boolean** field **visible**.

x, y	The 2 dimensional location of the object in the simulation.
speed, heading	The velocity of the object.
visible	Indicates whether or not the object is visible on the graphical representation of the simulation.

Objects contain the built-in functions **distance_to** and **bearing_to**:

distance_to*(object-identifier)*
returns an integer value indicating the distance to the passed in object

bearing_to*(object-identifier)*
returns an integer value indicating the bearing to the passed in object

3. Expressions

Expressions can be identifiers, constants or combinations of them joined by operators. Parentheses may be used to specify precedence in the ordering of evaluation of the operations.

3.1 Binary Expressions

[identifier or constant] operator [identifier or constant]

3.2 Object Dereference

An expression can refer to a field within an object by using the . operator in the following form:

identifier.identifier

The first identifier is the name of the object, the second identifier is the name of the field within the object.

4. Statements

Statements may appear within rule blocks. The following types of statements are possible.

4.1 Assignment Statement

An assignment statement has the form:

identifier <- *expression;*

Where the identifier may be a field of an object and expression must result in a matching data type.

4.2 Conditional Statement

A conditional is of the form:

if *(expression)*
{
    statement;
    ..
    statement;
} **else** {
    statement;
    ..
    statement;
}

5. Rules

A rule is a way to give behavior to an object. A rule is of the form:

rule *identifier* {
    statement;
    ..
    statement;
}

Example:

```
rule united432 {
    heading <- bearing_to(KPHL);
}
```

The fields of the object for which the rule is defined are implicit and need not be dereferenced in the body of the rule.
Rule blocks are executed once for every iteration of the simulation loop.

6. Object Definition, Creation and Destruction

Objects may be created at initialization if the object statement appears outside of a rule statement. Object statements inside rules will be executed every time they are encountered within the executing of the rule.

An object is defined and created with the object keyword:

object *identifier {*
type-specifier assignment-statement;
..
type-specifier assignment-statement;
}

Example:

```
object  KPHL {
        x <- 250;
        y <- 250;
        visible <- true;
        string landing_aircraft <- “united432”;

}
```

The type-specifier may be **integer, float** or **string**.
All objects contain the fields **x, y, speed, heading,** and **visible**. These fields are initialized to 0 if they are not explicitly defined in the object. Their types are implied and need not be stated in the object definition.

An object is destroyed with the destroy keyword:

destroy *object-identifier;*

When an object is destroyed, it’s rule is no longer evaluated, and any references to the destroyed object will result in a run-time error.

Example Code For Dog and Cat Chase Simulation

```
// This program will simulate a dog chasing a cat.  The dog will make a
// bee line for the cat.  The cat will run in the opposite direction when the
// dog enters a certain distance.

object dog {
        x <- 50;
        y <- 50;
        speed <- 4;
        heading <- 45;
        boolean hungry <- true;
}
object cat {
        x <- 150;
        y <- 150;
        speed <- 0;
        heading <- 45;
}

rule cat {
        if (distance_to(dog) < 20) {
                // cat will run from dog in opposite direction
                heading <- bearing_to(dog) + 180;
                speed <- 3;
        }
}

rule dog {
        if (hungry) {
                heading <- bearing_to(cat);
                if (distance_to(cat) = 0) {
                        destroy cat;
                        hungry <- false;
                }
        }
}
```