LINGI2132 Langages et traducteurs
Définition de la syntaxe

gr10 : Mulders Corentin, Forêt Nicolas, Pelsser François
17/02/2012

1 Introduction

Our language is an oriented oriented language that uses message passing. Everything element is an object. Basis classes are : Integer, Boolean, Tuple, Stdio and Messages. To communicate with an object, we send it messages the object can then either manage the message or sent it to an other object. It is dynamically typed.

2 Syntax description

⟨*identifier*⟩ : := 'a..z' { 'a..z,A..Z,_,0..9,'}

⟨*class_identifier*⟩ : := 'A..Z' { 'a..z,A..Z,_,0..9,'}

⟨*tuple*⟩ : := '['']'
| '[' ⟨*element*⟩ {',' ⟨*element*⟩}']'

⟨*message*⟩ : := '{'⟨*identifier*⟩ {',' ⟨*element*⟩}'}'

⟨*message_sending*⟩ : := ⟨*element*⟩ '<-' ⟨*message*⟩

⟨*boolean_value*⟩ : := true | false

⟨*element*⟩ : := ⟨*identifier*⟩
| ⟨*message_sending*⟩
| ⟨*instantiation*⟩
| {⟨*digits*⟩}
| ⟨*boolean_value*⟩
| ⟨*operation*⟩
| ⟨*message*⟩
| ⟨*tuple*⟩
| # | self | super
| ⟨*element*⟩.⟨*identifier*⟩

⟨*operation*⟩ : := ⟨*element*⟩ ⟨*binary_operator*⟩ ⟨*element*⟩
| ⟨*unary_operator*⟩ ⟨*element*⟩
| (⟨*operation*⟩)

⟨*binary_operator*⟩ : := '+' | '-' | '*' | '/' | '%'
| '<=' | '=>' | '<' | '>' | '==' | '!='
| 'and' | 'or'

⟨*unary_operator*⟩ : := ! | -

⟨*instantiation*⟩ : := ⟨*class_identifier*⟩ '<-' '{new' {',' ⟨*element*⟩}'}'

⟨*assignment_expression*⟩ : := ⟨*identifier*⟩ '=' ⟨*element*⟩

⟨*statement*⟩ : := ⟨*message_sending*⟩ ';'
| ⟨*assignment_expression*⟩ ';'
| return ⟨*element*⟩ ';'
| ⟨*while_statement*⟩
| ⟨*if_statement*⟩

⟨*comment*⟩ : := // .. suite de caracteres ASCII .. \n

⟨*block code*⟩ : := ⟨*statement*⟩ ⟨*comment*⟩

⟨*while_statement*⟩ : := 'while' '(' ⟨*expression*⟩ ')' '{'⟨*block code*⟩'}'

⟨*if_statement*⟩ : := '`if`' '`(`' ⟨*expression*⟩ '`)`' '`{`'⟨*block code*⟩'`}`'

⟨*class*⟩ : := '`class`' ⟨*class_identifier*⟩ (extends ⟨*identifier*⟩)? '`{`' ⟨*class body*⟩ '`}`'

⟨*class_body*⟩ : := {⟨*attribute_declaration*⟩} {⟨*message_declaration*⟩}

⟨*message_declaration*⟩ : := '`def`' '`{`'⟨*identifier*⟩ {',' ⟨*identifier*⟩}'`}`' '{'⟨*code_block*⟩'}'

⟨*attribute_declaration*⟩ : := ⟨*identifier*⟩ '`;`'

3 Data types

We would define some basic classes to store data.
– **Integer** represents integer numbers.
– **Boolean** represents a <boolean_value>.
– **Tuple** represents a collection (a <tuple> corresponds to a Tuple object).
– **Message** is a subclass of Tuple and represents a <message>.
– **Stdio** represents the standard input and output.

4 Syntaxic sugars

{digits} is translated in Integer←(new, {digits})
4 + 2 is translated in Integer←(new, 4)←(add, Integer←(new, 2))
The operator 'not' is a unary operator. Example : Integer←(new, 2)←leq(4)←(not). Syntactic sugar :!(2 <= 4)

Binary operators :

+	*sum*
−	*difference*
*	*multiplication*
/	*division*
%	*mod*
and	*and*
or	*or*
<=	*leq*
>=	*geq*
==	*eq*
!=	*neq*
<	*lt*
>	*gt*

Unary operators :

!	*not*
−	*minus*

5 Semantics

As there is no primitive types, we define a notation <Class value> which represent an instance of the class "Class" wich has the value "value"

5.1 Basic classes

5.1.1 Integer

The class integer represents all integer.
Sequences of digits are considered as an instance of the Integer class :

eval(n) = <Integer n> Where n ∈ {Z}
Integer class respond to these messages :
sum, difference, multiplication, division, mod, minus, leq, geq, eq, neq, lt, gt
The first part (sum, difference, multiplication, division, mod, minus), return a new Integer that has as value the result of the operation
Ex : eval(<Integer 4> <- {sum, <Integer 2>}) = <Integer 6>
The second part (leq, geq, eq, neq, lt, gt), return a new Boolean that also has as value the result of the operation
Ex : eval(<Integer 4> <- {lt, <Integer 2>}) = <Boolean true>

5.1.2 Boolean

This class represents the boolean values true and false.
The literals true and false are considered as instance of the Boolean class : eval(true) = ¡Boolean true¿ eval(false) = ¡Boolean false¿
Boolean class respond to these messages by creating a new boolean instance with as value thhe result of the operation :
not, and, or Ex : eval(<Boolean true> <- {and <Boolean false>}) = <Boolean false>

5.1.3 Tuple

The class tuple is used to keep collection of items. Elements separated by comas and surrounded by square brackets are considered as a tuple. eval([<Integer 4>,<Integer 3>,<Boolean true>]) = <Tuple <Integer 4>,<Integer 3>,<Boolean true>> Tuples responses to messages :

add, item a dd item at the end of the tuple

remove, i remove the item at position i of the tuple

itemAt, i return the item at position i of the tuple

The position of the items are Integer instances incrementing of 1 for every item and beginning at 0.

5.1.4 Message

The class Message represents messages sends to objects.
Elements separated by comas and surrounded by curly brackets are considered as a message.
The first element of the message is an identifier. It has to be formatted as an identifier (see BNF).
eval({messageName, param1, param2}) = <Message messageName,param1,param2>

5.1.5 Stdio

This class is used to print elements on an output. There is a global constant "stdio" wich represent the standard output.
This class manages the {print element} message that print a representation of the element on the output.

5.2 Operations and assignments

Let E represent the environment.
By default eval(expression) = F where F is a new Boolean in E and F as false as value. Else the value of the evaluations of expressions is defined by the following rules :
- eval(n, E) = i where i is a new Integer in E and i has n as value if n ∈ ℤ
- eval(b, E) = B where B is a new Boolean in E and B has b as value if b ∈ {true, false}
- eval(id, E) = O where O is the object in E referenced by the identifier id if id is a valid identifier.
- eval(id1.id2, E) = O where O is the Object referenced by id2 from the instance variable of the object referenced by id1 in E if id1 is a valid identifier and id2 is a valid identifier for a variable from the object of id1.

– eval({t1, t2, ... , tn}, E) = T where T is a new tuple in E that contains eval(t1), eval(t2), ... , eval(tn) if eval(ti) is an Object $\forall i \in [1, n]$.
– eval({id, t1, ... , tn}, E) = M where M is a new message in E with id as identifier and that contains eval(t1), eval(t2), ... , eval(tn) if eval(ti) is an Object $\forall i \in [1, n]$.
– eval(n1 + n2, E) = eval(n1) <-{sum, eval(n2)}) if eval(n1) is an Object and eval(n2) is an Object.

⋮

The same is valid for syntaxic sugars for the other binary operators with the messages from section 4.

⋮

– eval(!n1, E) = eval(n1 <-{not}) if eval(n1) is an Object.
– eval(-n1, E) = eval(n1 <-{minus}) if eval(n1) is an Object.

5.3 Class definition

A class contains 4 informations :
– a name
– a super class
– attributes
– messages definitions
There are 2 predefined messages, "new" and "default". The first one is called to create an instance of a class. The number of arguments depends of the definition of the new message(s) in the class. The second, "default", is called when the object received a message wich is not in its managed messages or in managed messages of one of its parents, this method can get the message to send it to another object with the character "#".
Every class attribute of the class have to be declared at the beginning of this one. They cannot be instanced during the declaration.
To define the managed message we should define every of them. This take place after the attributes declaration. For every message we have to set the name as an identifier, then add an identifier for every parameter. The parameter's identifier can then be used to retriveve parameter's values. The body can contain return statement that will define what will be returned. Ex :

```
def {messageName, param1, param2}
{
    return param1 + param2;
}
```

In this exemple we have a message named "messageName", it takes 2 parameters that can be used in the body of the message definition with identifier "param1" and "param2". The message call return the sum of the 2 parameters.

5.4 Sending Messages

– instr(o<-message, E) = instr(block code, E∪ {self→ eval(o, E), #→ eval(message, E)) where block code is found as defined below if o is the identifier of an Object in E.

5.4.1 Block code executed when a message is sent

– If the identifier of the message corresponds to a message defined for the receiving object then the block code defined for this object is executed.
– If none of the above, if the identifier of the message corresponds to a message defined for the receiving object's super class then the block code defined for this object's super class is executed.
– If none of the above, if the identifier of the message corresponds to a message defined for the receiving object's default receiving class then the block code defined for this object's default class is executed.

– If none of the above, if the identifier of the message corresponds to a message defined for the default receiving class of the the receiving object's super class then the block code defined for the default class of this object's super class is executed.
– If none of the above, no code is executed.

To summarize, the message is sent first to the destination object, if it can't do anything with it it tries it's super class messages, if no super class in the hierarchy knows this message then we try to send it to the destination object's default receiver (for example a method that prints an error message), if this receiver isn't defined we try to use one from the super class.

5.5 Control Sequences

5.5.1 Conditional execution

– instr(if(cond) {block code, E}) = instr(block code, E) if eval(cond, E }) = <Boolean true>
– instr(if(cond) {block code 1, E} else {block code 2, E}) = instr(block code 1, E) if eval(cond, E) = <Boolean true>
– instr(if(cond) {block code 1, E} else {block code 2, E}) = instr(block code 2, E) if eval(cond, E) = <Boolean false>

5.5.2 While loop

– instr(while(cond) {block code, E}) = instr(block code, E) while eval(cond, E) = <Boolean true>

6 Code examples

6.1 Definition of a new class

```
class Point{
                // two instance variables x and y declared
    x;
    y;

                // what to do when an instance is created
    def {new, px, py}
    {
        self.x = px;
        self.y = py;
    }

                //definition of an add message
    def {add, px, py}
    {
        self.x = self.x + px;
        self.y = self.y + py;
    }

                // getters
    def {getX}
    {
        return self.x;
    }

    def {getY}
    {
        return self.y;
    }
```

```
    def {gt, p2}
    {
        px = p2 <-{getX};
        py = p2 <-{getY};
        return (self.x*self.x + self.y*self.y) > (px*px + py*py)
     }
}

p = Point <- {new, 12, 14}
p2 = Point <- {new,13 ,28} + p

stdout <- {print, p > p2}
```