Programming Languages and Translators

COMS 4115

Ecosystem Modeling Language

EcoMod Final Report

Vika Kanchakouskaya

August 10, 2007

Introduction

This report describes EcoMod 1.1 language – ECOsystem MODeling language, which was designed in attempt to help scientists of various domains, who don't have special training in Computer Science, simulate a wide range of ecosystems.
The document consists of the following parts:

- **Introduction**

1. White Paper

2. **EcoMod Language Reference Manual**
3. **EcoMod Language Tutorial**
4. **Project Description**
5. **Language Limitations**
6. **Lessons Learned**
7. **Appendix**

The idea of EcoMod originated in response to the popular demand of computer modeling in science and technology. Experimenting and interacting with virtual models enables specialists to get a deeper insight into our surroundings and predict effects of various circumstances on the environments of interest. EcoMod is designed to help simulate such simplified virtual models of ecosystems, which can be studied and experimented with.

In particular, EcoMod is able to provide support for building various models of ecosystems of plants, animals and physical environments. With EcoMod one can create a simplified system inhabited by living and non-living entities, watch the system develop, simulate interference of natural and artificial phenomena into the system, and test robustness of the components of the system. Models created with EcoMod allow for observation of these systems in the time progression, and help understand development of its components, their responses to various conditions, and their reaction to stimuli of leaving and non-leaving elements of the environment.
The first release of EcoMod, version 1.1 is in its beta stage and has a limited support for the features mentioned about. However the language can be further extended and instrumented with various features, which will help build rather complex models or environments.

In this document we give a detailed language description, discuss features of EcoMod, provide the language tutorial, talk about the process of the language and the compiler creation, as well as give illustrations, examples, and tips to help first-time users get comfortable with EcoMod.

1. EcoMod Language White Paper

1.1 Introduction

Computer modeling is becoming increasing important for understanding how our world works. Experimenting and interacting with virtual models enables us to get a deeper insight into our surroundings and predict effects of various circumstances on the environments we live in. EcoMod is a language that helps simulate simplified virtual models of ecosystems, which can be studied, observed and experimented with.

1.2 Value Proposition

EcoMod provides support for creating various ecosystems of plants, animals and physical environments. With EcoMod one can create a simplified system inhabited by living and non-living entities, watch the system develop, simulate interference of natural and artificial phenomena into the system, and test robustness of the components of the system. EcoMod makes it easy for specialists in various domains to simulate such ecosystems as Rainforest, Desert, Marine, Human, and Urban ecosystems, etc. Models created with EcoMod allow for observation of these systems in the time progression, and help understand development of its components, their responses to various conditions, and their reaction to stimuli of leaving and non-leaving elements of the environment. The language is especially valuable for scientists who are, trying to understand evolution, predict effects of climate change and global warming on the life on Earth, or determine impact of human beings on the environment.

1.3 EcoMod Users

EcoMod is simple enough to be adopted by scientists of various domains who don't have special training in Computer Science, but is powerful enough to simulate a wide range of ecosystems. EcoMod users are scientists willing to experiments and understand properties of different communities, their habitats, and interaction of their components. Biologist can simulate ecosystems of Taiga, Tundra, Savanna, and Desert with their unique flora and fauna. Ecologists, with the help of EcoMod models can explore the effects of deforestation on the habitat of Rainforests, and impact of Greenhouse effect and raising temperatures on various ecosystems around the world. Urban Planning specialists can explore the effects of urbanization on human and environmental health. Marine scientists can observe the depths unreachable by human beings.

1.4 Properties of the language

EcoMod is a high-level domain-specific language which makes ecosystem modeling easier than with a general purpose programming languages like C++ or Java. EcoMod is compiled into C++ code, so it is supported on any platform capable of running C++ programs. The constructs of the language such as **system**, **objects**, **qualities**, **states**, and **actions** make it simple to create a model of an ecosystem, and observe its development with progression of time.

System is defined as the simplified ecosystem being modeled.

System is inhabited by **objects.** Various components of an ecosystem, like animals, plants and other non-living elements can be introduces to the model by defining the corresponding objects.

Objects may possess certain qualities, which can be depicted with the help of **quality** language construct. Qualities are a set of significant attributes, such as age, sex, defense strategy, health, etc, which describe and help differentiate between the objects of an ecosystem.

The behavior of the objects is modeled via finite state machine representation. **States** in EcoMod are constructs that store certain past information about the object up to the present time, and changes between the states occur by means of transitions. E.g. for a mammal, the following states of development can be defined: embryonic, metamorphosis, regeneration, aging, and death, and transitions between these states occur when an animal reaches a certain age.

Actions represent a way of interaction between objects and their community as well as between each other. Actions can be originated from the environment, and from other members of the habitat. Actions may trigger transitions between states and affect qualities of objects.

2. EcoMod Tutorial

In order to write a correct EcoMod program, the users need to follow the rules of the language. Please use the section below to familiarize yourself with these rules.

EcoMod program consists of 2 main parts: declaration and main.

2.1 Declaration

Declaration part is used to "describe" the system and its inhabitants. It should have declaration of the System and the Object that inhabits it.

2.1.1 System

The System declaration is very simple; it consists of System keyword and the System name:

```
System MySystem
{
}
```

2.1.2 Object

Object declaration should be nested into the System, and has the following form

```
Object MyObject
{
}
```

It is also required that the body of Object declaration has all the necessary components: Qualities, States, Transitions, and optional Actions. Here is an example:

```
Object MyObject
{
Qualities:
string color = “green”;
int hunger = 3;
// The qualities and similar to regular variables, and their default values
// should be assigned at declaration
States:
welfare = “hungry”, “happy”;
// There may be more than 1 state, and each state corresponds to and FSM.
// State values represent the nodes of the FSM and their relationships are
// defined in Transitions block below
Transitions:
welfare{
hungry:
   if hunger < 2 goto happy;
happy:
   if hunger >=2 goto hungry;
}
//onEntry(), onExit() – not supported by version 1.1
Action Feed()
{
   hunger = 0;
}
```

```
//Actions are similar to regular functions. They can reference other Object
members as well as have their local variables.

}  //end of Object declaration
```

2.2 Main

Main part of the program is where all the "action" takes place. There users can manipulate their objects with the help of the defined Actions, as well as by referencing States and Qualities directly.

```
void Main()
{
  int days = 0;
  int numberMeals = 0;
  Object person = 0;
 //All variable declarations have to include initial value assignment
//Object keyword declares an instance of type Object MyObject

while (days<100)
{
  days = days+1;
  hunger = hunger+1;
  person.updateFSM();
// a call to the build-in System function which will in turn call all the
// functions-transitions and will update the values of states associated
// with them
if(welfare == "hungry")
```

```
//the FSMs can be treated as variables, and their states and values
{
   pereson.feed(); //action call
   numberMeals=numberMeals+1;
}
}

print(”Number of meals: ”);
print(numberMeals);
print(“\n”);
//print will output the values of its parameters to
standard output
}
```

This short and somewhat trivial EcoMod program illustrates the basic building blocks of an EcoMod program. For more detailed information on the language please refer to EcoMod Language Reference Manual.

2.3 Another Example

Below we provide another sample EcoMod program we commentaries on the properties of the language

```
// This very simple program defines an experimental humanity
// which is originally inhabited by one person.
// 100 years of humanity development is simulated
// in order to observe how its population grows.
// We make a simplifying assumption that a person
// is able to regenerate independently.
System Humanity
{
```

```
  // nested Object declaration
  Object Human
  {
  // Qualities are member variables of the  Object
    Qualities:
       int age = 0;
       int avgNumChildren = 2;
       bool hasChildren = false;

    States:
       // FSM name followed by states comprising it
       development = “child”, “adult”, “aging”,
“dead”;

   // Transitions must be defined for every pair of states that can be
    // transformed from one to another with help of Transitions keyword
   Transitions:
     development {
       child:
         if age >= 16 goto adult;
       adult:
         if age >= 45 goto aging;
       aging:
         if age >= 80 goto dead;
       death:
 // onEntry() is defined for State death
 //the Object removes itself from Habitat
           onEntry { remove();}
  }

  Action Regenerate()
  {
  // checks what the current state of FSM
  // and if the Human regenerated already
    if (development==adult && hasChildren==false)
    {
       for (int i=0; i<avgNumChildren; i++)
       {
         Humanity.add(Human); //copies Human and adds
                              //it to Habitiat
```

```
        }
        hasChildren = true;
      }
    }
  }
}

// Main part of the program
void main()
{

  int period = 100;

  for(int year =0; year < period; year = year+1)
  {
     for (int index = 0; index < Habitat.Size();
          index = index+1)
     {
        Habitat.next().age++;
        Habitat.next().updateFSM();
        Habitat.next().Regenerate();
     }
  }
  print("Humanity has ");
  print(Habitat.Size());
  print(" people\n");
}
```

3. EcoMod Language Reference Manual

3.1 Notation

The following conventions are used in this manual to describe the rules of the language.

- `*` is used to denote one or more of the preceding token
- `< >` are used to enclose a group of tokens
- `[ ]` are used to enclose an optional token or group of tokens

3.2 Lexical Conventions

There are 5 types of tokens in EcoMod: keywords, identifiers, constants, operators and punctuation. White spaces denoted by space, tab, newline, and carriage return characters serve as separators between tokens and are otherwise ignored. At least one white space character is required to separate keywords, identifiers, constants and operators.

3.2.1 Comments

Comments follow C++ or Java style single line comment convention: all characters between // and new line or carriage return are considered comments and are ignored by the compiler. There is no support for multi-line comments, and // has to be placed at the beginning of every line intended as a comment.

3.2.2 Keywords

The following items are reserved as EcoMod keywords and may not be used otherwise:

if	*int*	*float*	*System*	*States*
else	*string*	*return*	*Object*	*Habitat*
while	*bool*		*Qualities*	
goto	*true*		*Transitions*	
	false		*Action*	

EcoMod is a case sensitive language, and upper and lower case letters are considered different. It should be noted that domain-specific keywords like *System* and *Object* start with an upper case letter to differentiate them from general keywords like *for* and *else.*

3.2.3 Identifiers

Identifiers are sequences of alphanumeric characters or underscores starting with a letter. EcoMod keywords cannot be used as identifiers. E.g. *Whale, piglet, human_being* and *zebra2* are valid identifiers, while *_frog, 4rest,* and *Object* are invalid.

3.2.4 Constants

There are 3 types of constants: integers, floats and strings.

Integers are sequences of digits and can only be decimal.

Floats consist of integer part, decimal point, and fractional part. Integer and fractional parts are sequences of digits and both are required to form a valid floating constant.

Strings are sequences of characters enclosed in double quotes. A double quote can be included into a string by escaping it with '\' .

Boollean constants are *true* and *false*

3.2.5 Operators

EcoMod supports a number of arithmetic, relational and logical operators.

Arithmetic operators: + - / * %

Relational operators: < > <= >= == !=

Logical operator: && ||

Initialization operator: =

Dot operator: .

The meaning and the precedence of the operators is the same as in C++ and Java. Dot operator is used to select a member of an Object or a System, similar to selecting a member of a class in C++/Java.

3.2.6 Punctuation

Punctuation is used to delimit other language tokens. The following punctuation symbols are supported in EcoMod

;	statement delimiter, denotes end of statement
{}	delimiter for a block of statements
()	delimiter for arguments to a function
“”	string delimiter

3.3 Types

EcoMod supports the following storage types

int	32 bit integer
float	floating point number with fractional part
string	a string of characters
bool	boolean type with values true or false
Object	type for objects inhabiting the ecosystem

3.4 Statements

In EcoMod statements are executed sequentially, unless stated otherwise. This section describes the types of statements supported in EcoMod

3.4.1 Expression statements

This is a most common type of statement. Expressions are usually assignments or function calls and should be separated by semicolons. It is a good practice to put one expression per line in an EcoMod program.

3.4.2 Conditional statements

The following types of conditional statements are supported:

*if (expression) { statement; * }*

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

In each conditional statement expression is evaluated, and if non-zero or true, the block of statements is executed, otherwise, if *else* clause is present, the block of statements following *else* is executed. The *else* ambiguity is resolved according to C language convention: else is bound to the last elseless *if*.

3.4.3 Loops

EcoMod supports the traditional while loops of the following format:

*while (expression) { statement; * }*

In the *while* loop, the expression is repeatedly evaluated in a loop and the block of statements is executed as long as the expression is non-zero or true.

3.4.4 Return statements

return expression;

Return statements are used to return the value of the expression to the caller of the function.

3.5 Functions

Due to the fact that EcoMod is a domain specific language that uses FSM to describe the states that the objects are in, there are several types of functions supported by the language.

3.5.1 Transitions

Transitions are functions that check on a specific condition associated with them and change the state in which the object is in, if the condition holds true. Transitional functions are implicitly defined as part of Object definition in the following form:

*FSM_name { < state : if expression goto state; > * }*

FSM_name denotes the name of the group of states and transitions that are part of one Finite State Machine. This is used to provide support for multiple FSM's per Object. Transitions are never called directly, and their evaluation can be triggered by calling updateFSM() built-in member function as follows:

Object_name.updateFSM(FSM_name);

3.5.2 Actions

Actions are special function that are accessible by the users and provide the user interface for interacting with the members of the *Habitat* and for

manipulating their behavior. Actions don't have return values and are defined with *Action* keyword:

*Action identifier ([parameter list]) { <statement> *}*

3.5.3 Standard functions ** *not supported in version 1.1*** **

EcoMod also supports standard C++ or Java -like functions, which should be defined in a traditional way including return type, function name, list of arguments, and function body enclosed in { }. Such function are private to Objects and can only be accessed from the body of the Object declaration.

3.5.4 Built-in functions

add(Object) - member of *System*, adds a copy of the *Object* to the global *Habitat* array

remove() - member of *Object*, removes the *Object* from *Habitat*

size() - member of *Habitat*, returns the array size

next() - member of *Habitat,* returns next element

updateFSM() - member of Object, call all transitional functions and updates the states

onExit() - member of a State, if defined – is implicitly called when Object leaves the state

onEntrance() - member of a State, if defined – is implicitly called as Object enters the state

print() - prints constants to stdout

3.6 Additional EcoMod Language Specifications and Summary

An EcoMod program generally consists of two parts: *definition* and *main*.

3.6.1 Definition Part

Definitions consist of defining *System* and *Objects* that inhabit it. There can only be one *System* per EcoMod program. System has a built-in *Habitat* array that contains all the objects inhabiting the *System. Objects* that are defined within the body of the *System* definition are added to *Habitat* automatically. Additional *Objects* can be inserted via a*dd(Object)* function call.

Object definition should be nested into *System* definition. The first release of EcoMod only supports one type of *Object* per *System*, consequently there can be only one nested *Object* definition; however multiple instances of that *Object* can be added to the system with the help of *add(Object)* function. *Objects* may possess certain *Qualities*, which can be depicted with the help of *Qualities* keyword. *Qualities* are a set of significant attributes, such as age, sex, defense strategy, health, etc. The behavior of the objects is modeled via FSM representation, with *States* and *Transitions*. Every state has built-in functions *onExit()* and *onEntrance()*, which may or may not be defined. *Actions* are *Object* members that provide a public interface for *Object* manipulation.

3.6.2 Main

Main part of the program in enclosed in *void main()* function. This is the part of the program where simulation takes place. *main()* function with void return type and no arguments should always be part of an EcoMod program.

4. Project Plan

4.1. Specifications

Specifications for EcoMod were developed in the initial stage of the project. In the White Paper we outlined the value proposition for the language, the target users, and the basic properties of EcoMod. The main goal was to create a language that would make it easy for scientists in various domains create models of the environment that would help them in their research. We envisioned EcoMod users as specialists who would like to better understand evolution, predict effects of climate change and global warming on the life on Earth, or determine impact of human beings on the environment. We proposed EcoMod to be a high-level domain-specific language which would make ecosystem modeling easier than with a general purpose programming languages like C++ or Java, because of a simpler syntax and the presence of the language constructs such as **system**, **objects**, **qualities**, **states**, and **actions.**

In the next stage we developed EcoMod Language Reference Manual, which described in detail the lexical convention, the grammar and the constructs of the language. The language reference manual also contained a sample EcoMod program, which illustrated how EcoMod can be used to write a program that helps observe population grows in 100-year period of time.

4.2 Planning

Once the specifications for the language have been finalized, we had to come up with a plan for building the EcoMod compiler.

A decision has been made to develop the compiler modules in a rather sequential manner, starting with the back end followed by the front end. We came up with the following plan:

1. Develop EcoMod Lexical Analyzer and perform a unit test
2. Develop EcoMod Parser and perform a unit test
3. Develop Tree Walker and unit test
4. Develop Java Runtime Libraries that perform AST node's translation to Java output code and unit test
5. Integrate the components and perform an integration test

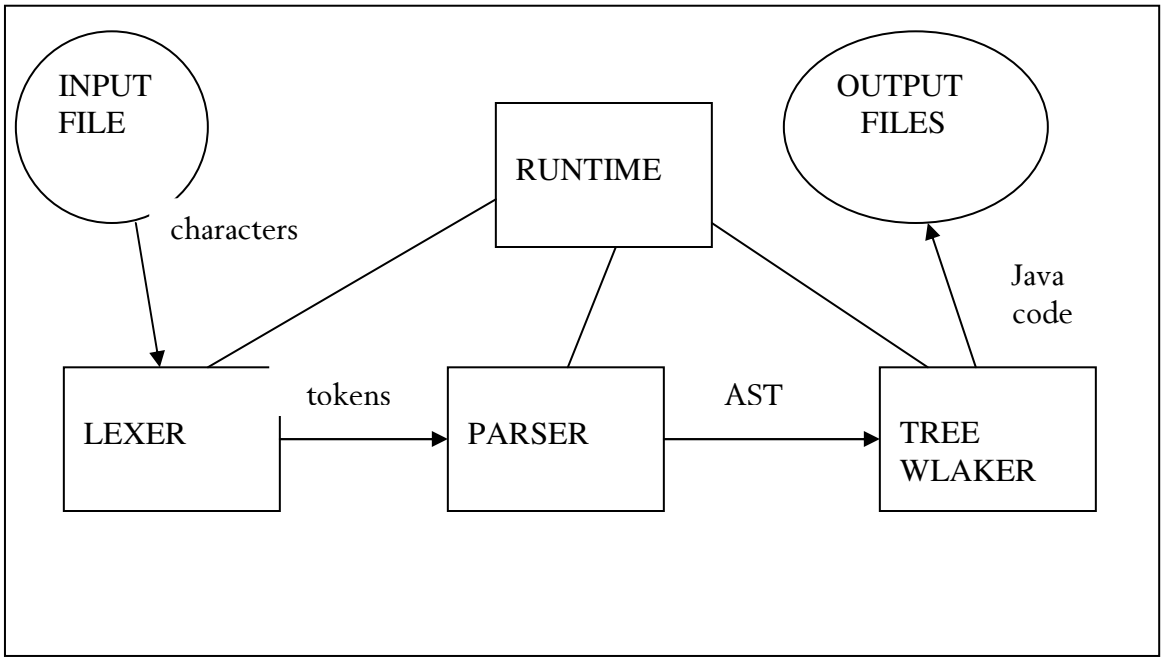
4.3 Development Environment and Tools

We used Java NetBeans version 4.1 to develop the libraries and to test the correctness of the output of the compiler

ANTLR v2 along with ANTLRWorks was used to develop the Lexer, the Parser, and the AST Walker. Unfortunately, ANTLRWorks tool only supports later versions of ANTLR, so we only used it for the text editing purposes, and the compilation and debugging was done with the help of command-line utilities.

5. Architectural design

To help us visualize the compiler and the project, we outlined the basic building blocks of the compiler and the data flow in the following diagram:



The diagram depicts data flow through the compiler, as well as interaction of the compiler components. The EcoMod input file gets passed to the Lexer, which takes a stream of characters as an input and outputs EcoMod tokens. The Parser uses LL(k) (with k=2) recursive-decent approach to processes these tokens and builds an AST. The Tree Walker interprets the tree and produces the output Java compliable program. Runtime is used to initialize the compiler components and forwards the data from one component to another. At the level of Tree Walker, Runtime also help construct the correct compiler output and generate output files.

6. Testing

Due to the time restrictions, brief Unit testing was done as the modules were developed. We unit-tested these modules by passing simple code junks to the components and verifying the correctness of the output.

Integration testing was also brief and consisted of passing complete but simple EcoMod programs to the compiler.

6.1 Example of a testing program

```
System Humanity
{
Object Human
{
Qualities:
  int age=0;
  int avgNumChildren = 2;
  bool hasChildren = true;
  int numChildren = 0;
  int numReprodChildren = 0;

States:
  development="child","adult","old","dead";

Transitions:
development{
  child:
```

```
  if age>=16 goto "adult";
  adult:
  if age>=60 goto "old";
  old:
  if age>=85 goto "dead";
  dead:
  if age>=85 goto "dead";
}


Action Regenerate()
{
  int i = 0;

  if(development=="adult" && hasChildren==false)
  {
     while(i<avgNumChildren)
     {
        Habitat.add(Human);
     }
  }
}
}
```

```
void main()
{

int i = 0;
int j = 0;
Object nxt = 0;

while(i > 0)
{

  while(j<Habitat.size())
  {
    nxt = Habitat.next();
    nxt.Regenerate();
    nxt.age = next.age + 1;
    nxt.updateFSM(development);

  }
 }
}
```

6.2 Generated Java code

The following 2 Java file have been generated for the source by the compiler:

Humanity.java

```
import java.util.*;
import java.io.*;

class Humanity{

public Humanity(){
}
 public void updateFSM(){
// not implemented
}

public static Vector Habitat = new Vector() ;
public static Iterator hIter = Habitat.iterator() ;

public static Human next(){
if(!hIter.hasNext())
  hIter=Habitat.iterator();
  return (Human)hIter.next(); }

public static class Human{
public Human(){
}
```

```
public void updateFSM(){
// not implemented
}

 public int age = 0 ;
 public int avgNumChildren = 2 ;
 public boolean hasChildren = true ;
 public int numChildren = 0 ;
 public int numReprodChildren = 0 ;
 public String development = "child" ;

public void Regenerate(){
int i = 0 ;
if (development == "adult" && hasChildren ==
false){
while (i < avgNumChildren){
Habitat.add(new Human());
}
}

}

private void child(){
if (age >= 16){
development = "adult"; }
}
```

```
private void adult(){
if (age >= 60){
development = "old"; }
}

private void old(){
if (age >= 85){
development = "dead"; }
}

private void dead(){
if (age >= 85){
development = "dead"; }
}

};
};
```

HumanityMain.java

```
import java.util.*;
import java.io.*;

class HumanityMain{

public HumanityMain(){
}
public static void main(String[] args){
Humanity System = new Humanity();
int i = 0 ;
int j = 0 ;
Humanity.Human nxt = new Humanity.Human() ;
while (i > 0){
while (j < Habitat.size()){
nxt = Habitat.next();
nxt.Regenerate();
nxt.age = next.age + 1;
nxt.updateFSM();
}
}
}
};
```

6.3 Language Limitations

There are a few limitations we had to adopt in order to comply with the project deadline. Currently the following built-in onEntry() and onExit() EcoMod functions are not supported, as well as updateFSM() and print().

EcoMod only supports Objects of one type in the System, and only one System per EcoMod program.

In addition to that, the Tree Walker is a little broken. It generates the *declarations* part of the program OK, but the *main()* part may be slightly broken. The error detection on the Tree Walker level also needs some work. However the Lexer, the Parser, and the Java runtime are working well.

Lessons Learned

The main lesson learned was that no matter how many “lessons learned” of the former students mention that it’s a good idea to start early, starting early seems to always be the hardest part of the project. Every single day (and night) I devoted to the project, I was wishing for more time and regretted to have pushed all the work to the very end. Never the less, it was a fun and enjoyable experience, although the outcome would have been significantly better if I had followed the advice.

Due to the time limitations, the compiler and the language didn’t quite come out as planned, and I had to introduce a number of limitations and quite a few simplifying assumptions.

To summarize, the good way to approach this project is as follows:

1. Start early
2. Think through and design all the components before starting to code. This can make the implementation much cleaner and can save a lot of time.
3. Unit-test thoroughly. The unit bugs during the integration process may result in a lot of frustration.
4. Know your tools. It’s a good idea to familiarize yourself with ANTLR and Java (or any other tools you use) ahead of time, and not while you are wring the compiler. I had limited or no experience with both, and found that is complicated the whole process.
5. Make it simple. Avoid unnecessary rules and use recursion when possible. Large and twisted grammars are hard to keep in your head and not easy to debug.

Appendix

2. Lexer, Parser, and Tree Walker (ANTLR)

```
header{
import java.util.*;
}

//LEXER
class EcoModLexer extends Lexer;
options { k = 2; }

WS        : (' ' | '\t' | '\n' { newline(); } | '\r' )+
          { $setType(Token.SKIP); } ;

COMMENT   :"//" (~('\n'|'\r'))*
          { $setType(Token.SKIP); };

protected LETTER : ('a'..'z' | 'A'..'Z') ;
protected DIGIT  : '0'..'9' ;

ID               :  LETTER ( '_' | LETTER | DIGIT | DOT )*;

NUMBER           : (DIGIT)+ (DOT (DIGIT)+)? ;
STRING           : '"'! ('"' '"'! | ~('"'))* '"'! ;

//arithmetic/other  //comparison         //braces
PLUS : '+';         ASSIGN : '=' ;       LBRACE  : '{';
MINUS: '-';         LE     : "<=";       RBRACE  : '}';
MULT : '*';         LT     : '<' ;       LPAREN  : '(';
DIV  : '/';         GE     : ">=";       RPAREN  : ')';
MOD  : '%';         GT     : '>' ;       LBRAKET : '[';
```

```
SEMI : ';';          EQ     : "==";      RBRAKET : ']';

DOT  : '.';          NE   : "!=";        OR  : "||";
COMA : ',';          AND  : "&&";
COL  : ':';          NOT  : '!';
```

//PARSER

```
class EcoModParser extends Parser;
options { buildAST = true; k = 2; }

program :      "System"^   ID   LBRACE!   objdec RBRACE! main
EOF!;
```

//DECLARATIONS

```
objdec     :   "Object"^   ID   LBRACE!   quals states trans
(acts)* RBRACE!;
quals    :     "Qualities"^   COL! (locdec)* ;
states   :     "States"^      COL! (statedec SEMI!)*;
trans    :     "Transitions"^ COL! (transdec)* ;

statedec   :    ID ASSIGN^ STRING (COMA! STRING)*;
transdec   :    ID^ LBRACE! (transn)*  RBRACE!; //pfunct
transn     :    ID^  COL!  ("if"  bool  "goto"  STRING  SEMI!)?
("onEntry"fbody)? ("onExit" fbody)?;
acts  :   "Action"^ ID LPAREN! pars RPAREN!  fbody ;
```

//MAIN

```
main :    "void"! "main"^ LPAREN! RPAREN!  fbody ;
```

```
//COMMON
fbody     :    LBRACE! (stmt)* RBRACE!;
fcall     :    ID^  LPAREN!  ((factor  (COMA!  factor)*)*)
RPAREN!;
pars :    (type ID (COMA! type ID)* )  | /*nothing*/;
locdec    :    type          ID          (          ASSIGN
(NUMBER|STRING|"true"|"false"))? SEMI!;
locdec2 : type ID SEMI!;

type :    ("int" | "string" | "bool" | "float" | "Object");

stmt :    locdec  //declaration
     | loc ASSIGN^ bool SEMI!         //assignment
     | ifstmt
     | whilestmt
     | "return" bool SEMI!     //return
     | fcall SEMI!                      //fcall
     | SEMI!
     ;

ifstmt      :  "if"^  LPAREN!  bool  RPAREN!  LBRACE!  (stmt)+
RBRACE!  (options  {greedy=true;}:  "else"  LBRACE!  (stmt)+
RBRACE!)? ;//cond

whilestmt   :  "while"^   LPAREN!   bool   RPAREN!   LBRACE!
(stmt)+ RBRACE!; //loop

loc   : ID^ LBRAKET! (loc|NUMBER) RBRAKET! | ID;
bool  :   join     (OR^ join)* ;
join  :   equality (AND^ equality)* ;
equality :     rel      ((EQ^ | NE^) rel)* ;
```

```
rel   :   expr     ((LT^ | LE^ | GT^ | GE^) expr)* ;
expr       :   term     ((PLUS^ | MINUS^ ) term)* ;
term       :   unary   ((MULT^ | DIV^ | MOD^) unary)*;
unary      :  (NOT^ unary | factor) ;
factor     :  fcall  |  loc  |  NUMBER  |  STRING  |  "true"  |
"false") ;
```

```
//TREE WALKER
class EcoModWalker extends TreeParser;
{
SimbolTable local = new SimbolTable(null);
SimbolTable global = new SimbolTable(null);
String systemName;
String objectName;
}

program returns [String p = null;]
     {EcoObject o = null;
      EcoObject m = null;
      EcoSystem s = null;}
     :#("System"

     (ID {systemName=#ID.getText();
          global.put(systemName, Type.Object);
          global.put("add", Type.Void);
          global.put("size", Type.Void);
          global.put("updateFSM", Type.Void);
          global.put("next", Type.Void);
          global.put("Habitat", Type.Object);
          s = new EcoSystem(systemName);}
      o=defs m=main)
```

```
     {p = s.toString(o) + m.toString();
     EcoFile   dec   =   new   EcoFile(systemName+".java",
s.toString(o));
     EcoFile maine = new EcoFile(systemName + "Main.java",
m.toString());}
     );

defs returns [EcoObject o = null]
     {Type t = null;
      EcoMember m = null;
      Vector v = null;
      }
      :#("Object"

     ID { objectName = #ID.getText();
          global.put(#ID.getText(), Type.Object);
          o = new EcoObject (#ID.getText()); }
#("Qualities" (m = decs    {o.addMember(m, "public");} )* )
#("States"    (m = states  {o.addMember(m, "public");} )* )
#("Transitions"(v=trans{o.addMemberSet(v, "private");})* )
               (m = act {o.addMember(m, "public");})*);


decs returns [EcoMember m = null]
     { Type t = Type.None;
       String value;}
:(t=type ID ASSIGN (NUMBER {value = #NUMBER.getText();}
          |STRING{value = "\"" + #STRING.getText() + "\"";}
          |"false"{value = "false";}
          |"true" {value = "true";})
               {String typeName = t.getType();
```

```
                if(typeName.equals("Object"))
                {
                  typeName   =   systemName   +   "."   +
objectName;
                  value  =  "new  "  +  systemName  +  "."  +
objectName + "()";
                }
                local.put(#ID.getText(), t);
              m  =  new  EcoMember(typeName,  #ID.getText(),
value);
              global.put(#ID.getText(), t);
              }
        );

decs2   returns [EcoMember m = null]
     { Type t = Type.None;
       String typeName;
       String value="";}
     :(t=type ID {typeName = t.getType();
                if(typeName.equals("Object"))
                {
                typeName = systemName + "." + objectName;
                value  =  "new  "  +  systemName  +  "."  +
objectName + "()";
                }
                local.put(#ID.getText(), t);
              m  =  new  EcoMember(typeName,  #ID.getText(),
value);}
         );

//returns type based on string encountered
type returns [Type t]
```

```
{ t = null; }
     :("bool"   {t = Type.Bool;}
     | "string" {t = Type.String;}
     | "int"      {t = Type.Integer;}
     | "float"  {t = Type.Floating;}
     | "Object" {t = Type.Object;})
     ;

//puts state name as String to sym tbl
states  returns [EcoMember m = null]
     {String value="";}
     :#(ASSIGN
          (ID { global.put(#ID.getText(), Type.String);
               m  =  new  EcoMember  (Type.String.getType(),
#ID.getText(), "");}
               (STRING       {global.put(#STRING.getText(),
Type.String);
                        if(value.equals(""))
value=#STRING.getText();})
        )
        {m.setValue("\"" + value + "\"");})
     ;

trans   returns [Vector ms = new Vector()]
     {EcoFunction m = null;
      String state;}
     :#(ID  {state  =  #ID.getText();}  (m  =  tran[state]
{ms.add(m);})*)
     ;


tran [String state] returns [EcoFunction m = null]
```

```
     {String body = "";
      String op = null;}
     :#(ID {global.put(#ID.getText(), Type.Void);
            m = new EcoFunction("void", #ID.getText(), "",
"");  }
        ("if" {body += "if (";})
        (#(EQ  {op = " == ";} (ID  {body += #ID.getText();
body  +=  op;})  (NUMBER  {body  +=  #NUMBER.getText();}|STRING
{body += #STRING.getText();}) )
        |#(LE  {op = " <= ";} (ID  {body += #ID.getText();
body  +=  op;})  (NUMBER  {body  +=  #NUMBER.getText();}|STRING
{body += #STRING.getText();}) )
        |#(LT  {op = " < ";}  (ID  {body += #ID.getText();
body  +=  op;})  (NUMBER  {body  +=  #NUMBER.getText();}|STRING
{body += #STRING.getText();}) )
        |#(GE  {op = " >= ";} (ID  {body += #ID.getText();
body  +=  op;})  (NUMBER  {body  +=  #NUMBER.getText();}|STRING
{body += #STRING.getText();}) )
        |#(GT  {op = " > ";}  (ID  {body += #ID.getText();
body  +=  op;})  (NUMBER  {body  +=  #NUMBER.getText();}|STRING
{body += #STRING.getText();}) )
        )
        ("goto"{body += "){\n";})
        (STRING   {body   +=   state   +   "   =   \""   +
#STRING.getText()+"\"; }\n";})
        {m.setValue(body);} )
     ;

act  returns [EcoFunction m = null]
     {String body;
      String name;}
     :#("Action"
```

```
          (ID {name = #ID.getText();
               global.put(name, Type.Void);
                m = new EcoFunction("void", name, "", "");}
     (body = fbody[name])
     {m.setValue(body);}
        ))
     ;


main    returns [EcoObject o = null]
     {String body;
      EcoFunction m;}
     :#("main"
     body=fbody["main"]
     {o = new EcoObject(systemName + "Main");
      m  =  new  EcoFunction("static  void",  "main",  body,
"String[] args");
      o.addMember(m, "public"); }
     )
     ;

fbody   [String fname] returns [String body = "";]
        {EcoMember m = null;
         String s, a, b;
         SimbolTable saved_environment = local;
         local = new SimbolTable(global);
         if  (fname.equals("main"))  body  +=  systemName  +  "
System = new " + systemName + "();\n";}
     :((m = decs {body += m.toString() + "\n";})*
      (s = stmt {body += s + "\n";} ))
      { local = saved_environment; }
     ;
```

```
stmt returns [String s]
     { String e1, e2; s = ""; String s1, s2; EcoMember
m=null;}
     :#("if" e1=expr s1=stmt {s += "if (" + e1 + "){\n" +
s1 + "}\n";}
     ( "else" s2=stmt
     {s += "else {\n" + s2 + "}\n";})?)
     |#("while"

     e1=expr
     s1=stmt
     {s += "while (" + e1 + "){\n" + s1 + "}\n";}

     )
     | (e1=expr {s += e1 + ";\n";})*
     | #("return" e1=expr  {s += "return " + e1 + "\n";})
     | SEMI { s += ";\n"; }

     ;


expr returns [String e]
     {String a, b, p="", id; e = "";}
     : #(OR a=expr b=expr    {e = a + " || " + b;} )
     | #(AND a=expr b=expr   {e = a + " && " + b;} )
     | #(EQ a=expr b=expr    {e = a + " == " + b;} )
     | #(NE a=expr b=expr    {e = a + " != " + b;} )
     | #(LT a=expr b=expr    {e = a + " < "  + b;} )
     | #(LE a=expr b=expr    {e = a + " <= " + b;} )
```

```
      | #(GT a=expr b=expr    {e = a + " > "  + b;} )
      | #(GE a=expr b=expr    {e = a + " >= " + b;} )
      | #(PLUS a=expr b=expr  {e = a + " + "  + b;} )
      | #(MINUS a=expr b=expr {e = a + " - "  + b;} )
      | #(MUL a=expr b=expr   {e = a + " * "  + b;} )
      | #(DIV a=expr b=expr   {e = a + " / "  + b;} )
      | #(NOT a=expr          {e = "!" + a;} )
      | #(ASSIGN a=expr b=expr {e = a + " = " + b;})
      | NUMBER                 {e = #NUMBER.getText();}
      | STRING                  {e = "\"" + #STRING.getText()
+ "\"";}
      | "true"                 {e = "true"; }
      | "false"                {e = "false";}
      | #(ID
           {id=#ID.getText();
           Type i = global.get(id);
           if (i == null)
              {i = local.get(id);}
           if (i == null)
              {System.out.println("Error:   in   scope   of
element  "  +  local.getName()  +  "  variable  "  +  id  +  "
undeclared");
               System.exit(1);}
           else
              {System.out.println(id  +  "  declaired  "  +
i.toString());
               e = id;

              }
              System.out.println(id);
           }
```

```
          (ID
            {i = global.get(#ID.getText());
             if (i == null)
               {i = local.get(#ID.getText());}
             if (i == null)
               {System.out.println("Error:   in   scope   of
element " + local.getName() + " variable " + #ID.getText()
+ " undeclared");
                System.exit(1);}
             else
               {System.out.println(#ID.getText()     +     "
declaired " + i.toString());
                if(id.equals("Habitat.add"))
                  {p= "new " + objectName + "()";}
                else if(!p.equals(""))
                  { p+=","; p+=#ID.getText();}
               }
               }
             )*

                {if(!p.equals("")){e +="(" + p + ")";}
                 else if(i.getType().equals("void"))
              e+="()";
               }
             );
```