CμLOG Project Final Report

An Entity Interaction Simulation Language

John Demme (jdd2127)
Nishant Shah (nrs2127)
Devesh Dedhia (ddd2121)
Cheng Cheng (cc2999)

Columbia University

December 17, 2008

Contents

1 Introduction: CμLOG

CμLOG is a logic language designed for entity interaction simulation. It uses a brute force method for solution searching similar to Prolog but uses a syntax similar to C, making it easier on the typical programmer's eyes, and is compatible with some code tools, such as code indenters and Emacs's c-mode.

Simulations in CμLOG involve a set of entities written in CμLOG which interact in the simulator. The "environment" entity defines the board on which the "agents" play, and defines the game which the entities play. It is a turn-based simulation during which each agent can look at the contents of the environment and decide which which direction it should move. During this decision, the agents can modify their own working memory, thus affecting their decision for the next turn.

Additionally, the CμLOG interpreter may be invoked separately from the simulator. The stand-alone interpreter searches for all the solutions for "main", but typically the output of these programs will be from "print" directives specified in the program.

1.1 Application & Features

One uses the language to provide a set of facts and rules, and the program is run by asking a question, which the interpreter attempts to answer using inferences based on the fact and rule set. CμLOG is designed for simulation, so typically a simulator will ask a given agent program what its next action will be. The agent program then uses CμLOGs entity interaction features to gather information about its environment and decide what to do. Each agent program can communicate with other programs to find out more information about other agents or its own status in the environment. The simulator stores all the contextual information pertaining to the environment and all of the agents present.

As this language is going to be used for simulating real life agents, we strongly emphasize that the program learn and forget data/rules/information at run-time. For this, similar to "assert" and "retract" of Prolog we have introduced two directives called "learn" and "forget." In CμLOG there exist no specific data structures like you would see in Java or Python, however rules and facts can be added to the program dynamically, which allows programs to remember data in a much more natural way since the data simply becomes part of the running code.

The simulator discussed could be modified to be used in other with other simulation environments, such as in a three-dimensional grid simulation with several agents–such as a flight simulation. Alternatively, the interpreter could be used in a real environment like the movement of pick and place robots in a warehouse. The language could be used to define the warehouse environment and agent programs for robots, and a replacement for the simulator would feed live information in to the programs in the form of facts, similarly to how the simulation feeds its state information to agents now.

1.2 Goals

The language and simulator presented here attempts to fulfill the following requirements:

Generic Games are defined mostly by the environment application.

Composable Individual behaviors can be written simply and easily, then combined to obtain high-level actions and reasoning

Declarative Programmers can specify what they want entities to do rather than how

Controlled Communication Data in the system is frequently made up of nearly-atomic bits of data many of which can be used both on their own and composed as complex data. This means that subsets and smaller pieces of data can be communicated between entities without loosing meaning.

High-level libraries Due to the flexibility and composibility of the language, high-level algorithms-such as path finding-can be easily implemented in libraries, allowing further, domain-specific intelligence to be written in the programs.

2 Tutorial

Logic programming is a kind of computer programming using mathematical logic. Specifically, it is based on the the idea of applying a theorem-prover to declarative sentences and deriving implications. Compared with procedural languages, logic programming solves the problem by setting rules with which solutions must fit. We can represent logic programming by the formula:

$Facts + Rules = Solutions$

Logic programming languages are inherently high level languages, allowing programmers to specify problems in a declarative manner, leaving some or all of the details of solving the problem to the interpreter.

Both the programming and data structures in both prolog and CμLOG can be very simple- such as facts. The relationship between code and data is also of note. CμLOG uses the Von Neumann style (vs. Harvard architecture) wherein data is code. It is therefore possible (and inherently necessary) for programs to be introspective and self-modifying. In other words, it is easier for programs to learn and adapt.

2.1 Variables

Variables represent a value to be solved for. They don't have a fixed datatype, but match to the refered type. All variables are scoped to the rule, so that variable solutions can be shared between sub-blocks.

Variables are represented by a dollar sign ($) then the variable name. The name must start with a letter, and is composed of letters, numbers, and underscores. There is a special variable called the anonymous variable which is represented simply by a question mark (?).

```
Example variable names:
   \$foo  \$bar\_  \$f1o2o3

The following are not valid variables:
  foo  \$\_foo  \$1bar
```

2.2 Statements

These are conditional statements which give output as true or false only and are frequently used to constrain variables. They are of two types, comparison and evaluation statements.

Comparison statements are used to compare variables against constants:

```
Example comparisons:
  \$a>1+3-4; //means that variable 'a' is always greater than 0
  \$boo <= 5; // means that variable 'boo' is less than or equal to 5
```

Evaluation or eval statements are used to query the program for solutions:

```
  boofar(\$s,\$d,7); //from all the possible matches in the program's
     //graph it returns various possible values for the pair s and d,
     //and constrains those values in their scope appropriately, as
     //defined by the block in which the statement is contained
```

2.3 Facts

Facts are terminal nodes of the solution search which are always true. Facts help us define constant information in the program like the position of a wall.

```
Syntax: id(parameter1, parameter2 ....);


Examples:
  wall(2,3);  //This means that a wall is defined for 2 and 3

  fire(4,a); //Symbols like 'a' can also be a parameter.
             // Here, fire of 4 and 'a' evaluates to be true.
```

2.4 Rules

Rules are similar to facts, but are only conditionally true. These conditions are defined inside a block. The defination or declaration of rules suggests that the solution tree is about to branch out to search for new solutions.

```
  syntax: id(parameter1, parameter2....) {conditions}
```

The block is "conditions" in the above syntax. Block can be of 2 types, namely 'AND' and 'OR' block. AND blocks evalute true iff all the conditions inside the block are true. Similarly, the OR block is true if any one of the conditions is true. If no reduction method is specified (i.e. AND or OR is written), by default AND is used.

To define a OR block we use the following construct:

```
  {OR:
    foo();
    bar();
  }
```

The AND block is written similarly:

```
wall(2,3) {AND:
  foo();
  bar();
  {OR: barfoo(); foobar();}
}
```

Here "OR: barfoo(); foobar();" is a sub-block. wall(2,3) is true if foo() and bar() are true and if either of barfoo() or foobar() are true.

2.5 Directives

Three interpreter directives are supported; print, learn and forget. print is used to output strings and results during runtime. the learn and forget directives are used for database modification. They function similar to assert and retract of prolog.

```
Syntax: @directive_name(parameters);


Example:
     //prints "hello world: " then whatever constraints exist on $foo
  @print("hello world:", $foo);

    //adds a fact to the database that 'fire' is true for 4,5
  @learn(fire(4,5););

    //erases the fact from the database that tree is true for 3,9.
  @forget(tree(3,9););
```

2.6 Simulator

Now for the user to be able to run a simulation or play a game in Cμ, they will have to use a simulator which interacts with the logic engine of the langauge to produce required results. For demonstration we have done so already. This simulator defines a class of games or simulations described as follows:

The environment is grid based and defined by a CμLOG program. It potentially includes obstacles and a goals which the agent must reach, however the game is defined mostly by the environment program. Every object (i.e. agents, walls, switches, goals) in the environment is defined by grid positions. The environment specifies the representations of the entities to the simulator. The simulator re-evaluates the various object rules during each turn when it renders the grid, so the contents of the grid can be dynamically defined based on the state of the simulation or the contents of the program (which can be changed by the program.) For example based on the grid position of the agent the environment might remove or insert a wall. The agent program decides the next move based on previous moves and obstacle data.

The simulation of the agent program is also turn based. Each time the agent makes a move it sends its new coordinates to the simulator. The new coordinates become part of the simulation's state which are exposed to the environment when it is solved to render the scene.

```
Example 1:
  Size(5,5);  //defines the grid size of 5 by 5
  wall(2,3);  //a fact where wall is present at coordinates 2,3
  wall(4,2);
  goal(3,3);  //a fact which defines the goal to be achieved by the player

  igo("UP");  //move($dir) would be true for all the values of $dir
              // for which igo($dir) is true

  move($dir){
        //causes the interpreter to remove igo("UP") from its database.
    @forget(igo($dir););

        //Fetch the next movement
    igo($dir);
  }

The output of the above program is:
X


. . . . .
. . . . .
. | # . .
x . . | .
. . . . .
```

In the above example, 'size', 'goal', 'wall' and 'move' are keywords for the simulator. Size(5,5) defines the grid in which walls (shown by the pipe symbol) are placed at coordinates (2,3) and (4,2). A goal object (shown by #) is placed at (3,3). The game simulation ends when the agent either hits a wall, moves out of the grid or reaches the goal.

In order to run code through the simulator, put your code in a file with a ".ul" extension (this extension is a convention only) then invoke the simulator, passing it the name of your code file:

```
./simulator mySimulation.ul
```

2.7 Program Modification

Now let us look at example using one of our program modification directives.

```
Example 2:
  size(5, 5);
  wall(2, 3);
  wall(4,2);
```

```
goal(3, 3);
imove("UP");
imove("RIGHT");
imove("RIGHT");
imove("UP");

move($dir) {
  @forget1( imove($dir); );
  imove($dir);
}
```

OUTPUT:

```
==== Turn 1 ====
. . . . .
. . . . .
. | # . .
x . . | .
. . . . .

x: Moving RIGHT

==== Turn 2 ====
. . . . .
. . . . .
. | # . .
. x . | .
. . . . .

x: Moving RIGHT

==== Turn 3 ====
. . . . .
. . . . .
. | # . .
. . x | .
. . . . .

x: Moving UP

Simulation over: x wins!!!Successfully reach the goal at position (3,3)
```

In the above code we see a carefully drafted route through the grid can make you win the game. Each step of the simulation is displayed. In this example, the 'imove' facts are used as a stack of moves which are queried for each turn, and removed from the stack after using it. The "@forget1" directive shown in this example removes only one fact from the program instead of all the facts which match the pattern.

2.8 Breakout

Although we can define an agent's actions within the environment program, it is typically more desirable to specify a separate agent file so that multiple agents can operate in the same environment. In the next example, we use a separate agent which queries the environment for the movements it should take–sort of like asking for directions.

```
Environment Program:
  size(10, 10);
  wall(?, 7);
  goal(6, 6);

  imove("UP");
  imove("RIGHT");
  imove("RIGHT");
  imove("RIGHT");
  imove("RIGHT");
  imove("RIGHT");

  agent("d", "tests/agents/delg_to_env.ul");

Agent Program:
  move($dir) {
    @print($e, " says move ", $dir);
    $e.@forget( imove($dir) );
    env($e);
    $e.imove($dir);
  }
```

3 Language Reference Manual

3.1 Lexical

```
[' ' '\t' '\r' '\n'] WS
"/*"     OPENCOMMENT
"*/"     CLOSECOMMENT
"//"     COMMENT
```

```
'('      LPAREN
')'      RPAREN
'{'      LBRACE
'}'      RBRACE
';'      SEMICOLON
','      COMMA
'+'      PLUS
'-'      MINUS
'*'      TIMES
'/'      DIVIDE
"=="     EQ
'<'      LT
"<="     LEQ
">"      GT
">="     GEQ
'@'      AT
'.' DOT
'"'      QUOTE
'?'      QUESTION
'!'      NOT
'$'['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* Variable
['0'-'9']+ Number
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* Identifier
```

3.2 Facts

Facts define factual relationships. They have a very similar syntax to rules, except they have no code block to make them conditionally true. Any query which matches a fact is simply true. Another way to think of facts is as terminal nodes in the solution search.

Each fact is composed of a name, and a comma separated list of parameters, each of which may be a constant, or a variable. Using any variable except the anonymous variable doesn't make much sense in a fact, but is allowable.

```
Example:
  foo(4, symA);  //Foo of 4 and symA is always true
  foo(4, symA, ?); //Foo of 4, symA, and anything (wildcard) is always true
  wall(4, 5); //In an environment might mean: there is a wall present at (4,5)

Grammar:
  Fact -> Identifier ( ParamList );
  ParamList -> Param | ParamList , Param
  Param -> Variable | Number | String | Identifier
```

3.3 Rules

Rules define relationships which are conditionally true. They are similar to facts, but instead of ending with a semicolon, they contain have a block, which defines the conditions upon which the rule should be evaluated as true. Another way to think of a rules is as a node in the solution search which may branch, or be a leaf, depending on the contents of the condition block. Each rule is composed of a name, a comma separated list of parameters, and a block.

```
Example:
  foo(4) { bar(5); }  //Foo of 4 is true if bar(5) is true
  foo(4) { bar(6); }  //Foo of 4 is true if bar(6) is true
The two above rules are together equivalent to:
  foo(4) {OR: bar(5); bar(6); }
```

```
Grammar:
  Fact -> Identifier ( ParamList ) Block
```

3.4 Variables

Variables represent a value to be solved for. During rule matching, they will match any value or type, but can be constrained in an associated block. All variables are scoped to the rule, so that variable solutions can be shared between subblocks. Variables are represented by a dollar sign ($) then the variable name. The name must start with a letter, and is composed of letters, numbers, and underscores. There is a special variable called the anonymous variable which is represented simply by a question mark (?). It cannot be referenced in the block, and simply matches anything.

```
Example:
  foo($X, $y, $foo_bar, $bar9, ?) { }
```

```
Grammar:
  Variable -> $[a-zA-Z][a-zA-Z0-9_]* | ?
```

3.5 Blocks

Blocks contain a list of statements (conditions) to determine truth, and specify a reduction method for those statements. Each block will reduce all of its statements using the same reduction method (usually AND or OR), but may contain sub-blocks. If the reduction method is omitted, AND is assumed. The syntax allows for other reduction methods to be allowed (such as xor, or a user-specified method), however the language does not yet support this.

```
Examples:
 {
    foo();
    bar();
```

```
}
//True if foo and bar are both true.

{AND:
   foo();
   bar();
}
//True if foo and bar are both true.

{OR:
   foo();
   bar();
}
//True if foo or bar are true.
```

```
Grammar:
  Block -> { (Identifer:)? StatementList }
  StatementList -> Statement | StatementList Statement
```

3.6 Statements

Statements are boolean qualifiers which are used inside of blocks. They can be any one of three types: comparisons, evaluations, or blocks. Comparisons are used to constrain variables. Only values of the same type can be compared, and certain comparisons only work on certain types, so comparisons can be used to constrain variables by type. Evals are used to query the program, and have a similar syntax as facts. They can be thought of as a branch in the solution search. Blocks are considered a statement to support sub-blocks. They are evaluated and the reduced result is used. Comparisons and evals are both terminated by semicolons.

```
Examples:
  $X < 10;  // A comparison
  range($X, $Y, 7);   // An eval
  !range($X, $Y, 7);   // This must not evaluate to true
  {OR: $X > 10; $X < 0; }  //A sub-block with two binary comparisons
```

```
Grammar:
  Statement -> Block | Eval ; | Comparison ;
  Eval -> (!)? Identifier ( ExprList );
  ExprList -> Expression | ExprList , Expression
  Comparison -> Expression ComparisonOp Expression | Expression ComparisonOp Comparison
  ComparisonOp -> EQ | NEQ | LT | LEQ | GT | GEQ
```

3.7 Comparisons

Expressions are used to constrain variables. One side of the comparison must be a variable, and the other a constant. Depending on the type of the constant, only certain comparisons are allowed.

```
Examples:
  $r < 10;  // a comparison

Grammar:
  Comparison -> Expression CompOp Expression
  CompOp -> EQ | LT | LEQ | GT | GEQ
  Expression -> Number | String | Variable | Expression Op Expression | ( Expression )
  Op -> PLUS | MINUS | TIMES | DIVIDE
```

3.8 Types

The following types are supported: integers, strings, symbols, and entities. Strings in CμLOG are currently atomic, so no string processing such as splitting, joining, or searching is supported. They are primarily used for interaction with the rest of the system (printing, specifying files, ect.). Symbols are simply identifiers and can only be compared with equals. Entities are used to represent other programs (typically agents) and are used for interaction. In addition to equals and not equals comparison operators, they support the dot operator for interaction (discussed later.)

3.9 Directives

CμLOG supports a special syntax for interpreter directives. This allows programs to interact with the interpreter while avoiding symbol collisions. The syntax is similar to that of a fact's, but an at sign (@) is prepended. Three directives are currently supported: print, learn, and forget. Print is used to output strings, and results of searches during runtime. Learn and forget are discussed in the next section.

```
Examples:
  @print("Hello, world!");

Grammar:
  Directive -> @ Identifier ( ParamList );
```

3.10 Program Modification

The two directives learn and forget are used to modify a program at runtime. This is the only way in which CμLOG supports non-volatile storage. Learn is used to add a fact to a program, and forget is used to remove a fact. The synatax for these two directives is special, consisting of the usual directive syntax, except contained inside the parenthesis is a fact definition. Any non-anonymous variables in this fact definition are filled in with solutions found for those variables, and the learn or forget is “executed” once for each solution. They are similar to Prolog's assert and retract.

```
Examples:
  @learn( wall(4,5); );  //Remember that there is a wall at (4,5)
  @forget( agent(8, 10); ); //Forget about the agent at (8, 10)

Grammar:
  Directive -> @ (learn|forget) ( Fact List );
```

3.11 Interaction- The Dot Operator

If a variable or symbol represents another program (entity), then it supports the dot operator. After appending a dot (.) to the reference, one can put an eval, a learn, or a forget, and that action will take place in the other entity's namespace. This can be used to ask for information from another program (such as the environment program or another agent) or to modify the other program–perhaps to teach another agent, to trick a competitor, or to change the operating environment. Future versions of CμLOG could likely support some sort of access rules in the destination program, allowing it to control who is allowed to access what data, and who is allowed to change its program, and how. These access rules could potentially modify any queries or changes, perhaps revealing an entirely fake namespace to the other agent. Such access rules are beyond the scope of CμLOG initially, however.

```
Example:
  $agent.@learn( wall(4,5); );  //Tell agent2 that there is a wall at (4,5)
  env.view($X, $Y, $obj); //Query the environment, find out what is at ($X, $Y)

Grammar:
  DotOp -> Directive | Statement
  Dot -> Variable . DotOp | Identifier . DotOp
```

4 Project Plan

4.1 Responsibilities

It was the responsibility of each team member to complete and help complete the individual parts of the interpreter. Specifically, initially the scanner and parser were developed by Devesh Dedhia and Nishant Shah. The AST file was done by Cheng Cheng. The interpreter and translator were completed by John Demme. Nishant Shah and Cheng Cheng developed the simulator together. Testing each phase and testing the whole system was not assigned to any particular person as it requires as much man power as available. Testing was done by every group member.

4.2 Timeline

The following were the timelines we decided on at the start of the semester:

As we started working on the project, it was soon realized that the above deadline are not what our aim should be as, it is not a start-end process. The development process was more like evolution. So every

Table 1: Project Deadlines

Language Features design	Oct 20
LRM	Oct 22
Scanner/Parser	Nov 5
Translator	Nov 15
Interpreter	Nov 22
Simulator	Nov 27
Cold Freeze	Dec 12
Testing	Dec 18

section was up and running by Nov 15th, i.e. by then we were able to print ”hello world” in our language. After that we have been adding features and for that support is needed on every level, including the scanner, parser, ast file, translator, interpreter and the simulator. All members have been simultaneously working on the development and also testing the features at the same time.

4.3 Software Development Environment

The project will be developed on Ubuntu using Objective Caml 3.1.0. The scanner was developed using Ocamllex v3.1.0. The parser was developed using Ocamlyacc v3.1.0. We will use Python to run our tests and compare it with the expected output. Version control,managing multiple revisions of files, not only program source files, but any type of files is done using Subversion. We are using Google Code for issue tracking and Subversion hostin, plus Google groups(”pltsim”) for communicating within ourselves.

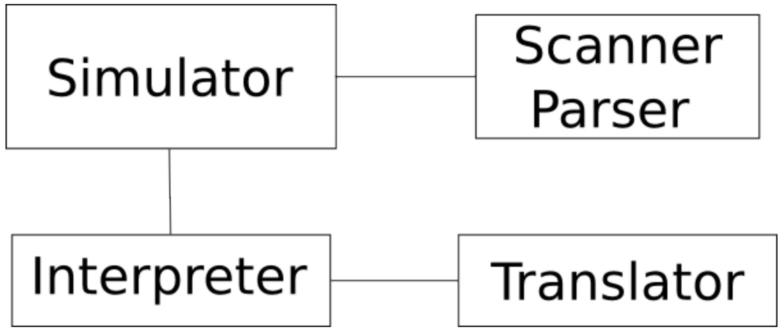
5 Architecture

The language CμLOG we have designed will be used for communication between agents and an environment, as well as to determine behavior of said entities. Every agent program communicates with the environment program through a simulator. The simulator runs a CμLOG logic solver and interpreter which functions on a set of rules and facts defined and modified by the environment and agents then provides solutions representing the actions to be taken by the agents.

The cmulog interpreter consists of several major blocks: scanner, parser, translator and interpreter. The relationship between these components is demonstrated in figure *. The simulator loads each program by reading in each necessary “.ul” file through the scanner and parser, resulting in an AST. The each AST is then passed into the interpreter, resulting in a database of rules and facts in the interpreter's internal format. The interpreter does not operator directly out of the AST, however: it uses the translator to convert the AST to a “translated synatax tree” (TST) first.

While the AST directly represents the structure of a CμLOG, there are a number of static transformations which do not change the meaning of the program, but make interpreting it much easier for the interpreter. The TST represents a simplier version of the program. The translator removes all the variable names and indexes them to a list and then each of them are identified by the number rather than the name. It also partitions all the statements with and without side-effects and runs all the ones without side-effects once for

Figure 1: Architecture Block Diagram



each solution. It performs all possible arithmetic to reduce each statement into its simplest form. It brings the unknown variable to the leftmost side by making all the necessary changes. For $(3 + 4 > \$x - 1)$ will reduce to $(\$x < 6)$. Lastly, all the static semantic checking is also done in the translator.

Using the database reference returned from the interpreter to the simulator, the simulator (or any other program for that matter) can query the database using the interpreter's "query" method, passing in the name to query, and the number of parameters. From this query method, the solution solver lazily evaluates the query by returning either "NoSolution" or "Solution". "NoSolution" indicates the all the results have already been returned. "Solution" is composed of a list of constraints (one per parameter) and a function to generate the next solution. Each solution is computed when the next function is called, so if there are an infinite number of solutions, the query function will not block forever. The caller may iterate through all of the solutions, or use only the first one.

Finally, the simulator uses the interpreter to query various terms for each turn, modifying its state and printing the output. For instance, it queries "wall" with two parameters (for each coordinate) each turn, iterates through all of the solutions, and puts a wall at each solution. For each agent, it queries the "move" term and uses only the first solution to move each agent. Before the first move, the simulator even queries the environment for the "agent" term to get the location of the agent program and its symbol!

Other programs such as the stand-alone "culog.ml" interpreter use the same interpreter interface to parse programs into databases and query these programs using different terms to invoke different behaviors. The stand-alone interpreter queries and "main" term and iterates through the results, and is generally used for testing of the interpreter. Other programs could use their own terms to generate different behaviors.

6 Test Plan

6.1 Testing Script

The python script shown in listing 1 is used to run our tests. Each test can be one of three different types: a parsing test, an interpreter test, or a full simulation test.

Listing 1: Testing Script

```
#!/usr/bin/python
```

```python

import os.path
import os
import glob
import sys

def run_and_compare(cmd, testOut):
    pipe = os.popen(cmd)
    to = open(testOut)
    for line in pipe.readlines():
        toLine = to.readline()
        if toLine != line:
            to.close()
            pipe.close()
            return False

    pipe.close()
    if to.readline() != '':
        to.close()
        return False
    to.close()
    return True

tests = glob.glob("tests/*.ul")
tests.sort()
for test in tests:
    testOut = test.replace(".ul", ".out")
    sys.stdout.write("Running %-35s..." % (test))
    sys.stdout.flush()

    if not os.path.exists(testOut):
        print "No Output"
        continue

    if (test.find("/pr") != -1):
        prog = "print <"
    else:
        if (test.find("/sim") != -1):
            prog = "simulator"
        else:
            prog = "culog"
    if run_and_compare("./%s %s 2>&1" % (prog, test), testOut):
        print "OK"
    else:
        print "FAIL!"
```

Listing 1: Testing Script

6.2 Test Case Rationale

Most of our test cases are written to test a specific feature. For instance, "andTest.ul" is designed to test AND blocks. To whatever extent possible, these tests avoid testing other features. This makes it easier to determine what feature has been broken when tests start failing. Other tests are designed to fail to make sure that various parts of the system fail properly. Still other tests are composite tests and are designed to test the system as a whole- they test multiple features at once to ensure that there are not bizarre interactions between various parts of the system.

6.3 Testing Results

As of the writing of this report, the testing results are shown below. Each of the test inputs and outputs can be found in the test cases appendix.

```
Running tests/andTest.ul                 ...OK
Running tests/facts.ul                   ...OK
Running tests/learnForget1.ul            ...OK
Running tests/main.ul                    ...OK
Running tests/main_fall_through.ul       ...OK
Running tests/mult-main.ul               ...OK
Running tests/neq.ul                     ...FAIL!
Running tests/not1.ul                    ...OK
Running tests/plist-twice.ul             ...OK
Running tests/printer_test.ul            ...OK
Running tests/prsimple.ul                ...OK
Running tests/prstrings.ul               ...OK
Running tests/range.ul                   ...OK
Running tests/sim_dot1.ul                ...OK
Running tests/sim_dot2.ul                ...OK
Running tests/sim_my_loc.ul              ...OK
Running tests/sim_ndot2.ul               ...OK
Running tests/sim_two_test.ul            ...OK
Running tests/simulator_test.ul          ...OK
Running tests/sprint1.ul                 ...OK
```

7 Lessons Learned

7.1 From Devesh

Programming languages and translators project introduced me to a whole new world of programming both logic and functional. In the first project meeting we envisioned a programming language to simulate agent movement in a grid based environment. The most obvious choice was having a logic level programming language. The learning course started with choosing a convinient and accurate grammar. After that working on the parser and scanner made me realize the power of type checking in ocaml. I also learnt how to logically reason and resolve the shift reduce and reduce reduce conflicts.

Implementation of the interpreter introduced me the power of recursive programming. Writing tests help me find bugs in the language,also it taught me that the testing is an ongoing process. Working in a team and meeting self made deadlines was also part of my learning.

After the working on this project I found it much easier to understand the syntax and program in "Murphy" a formal verification language. Along with programming languages it also exposed me to shell scripts, Makefiles, Svn repositories.

7.2 From John

A few rules to live by:

Tools Don't use the wrong tools for the right job or the right tools for the wrong job. Even parity is required! The interpreter is written to lazily evaluate queries both to reduce memory usage and avoid infinite loops in cases of infinite solutions. This lazy evaluation stragety would have been much easier to implement either with co-routines or lazy evaluation (a la Haskell.) Since OCaml offers neither of these features, I implemented lazy evaluation by hand, and it made everything harder by an order of magnitude. Since I didn't have the proper tools available, I shouldn't have written lazy evaluation. I should have created a logic solver would could only operate on a smaller class of programs, and would return all the results at once.

Testing Everyong tests while they write code. You have to. Frequently, you write tests in a temporary file and discard when the feature is "working." Don't do this. It's almost always worth the extra time to set up a test bed and put your tests in it. Then keep them. Run them often. Passing tests gives you a warm, fuzzy feeling which grows with the number of tests. So, keep all your "temporary" tests and you'll not only feel better about yourself, but you'll be ensuring long-term quality.

Refactoring At first, you don't know the features of the library and language. You'll write an AST with unnecessary boxing and unboxing. You'll re-write List.filter, and use lambdas when currying would have done the job. Realizing it later on is the mark of a good programmer. Refactoring this code is the mark of a diligent one.

BYOT Bring your own tools! Tools are what separate the chaff from the wheat. If you need something done and a tool can do the job, write the tool. Scripting languages are great for this, so one of the best time savers is intimate knowledge of a scripting language. Any language will do, but you can't be afraid of using it. Python is my pocket knife of choice, and as far as I'm concerned, there's no such thing as abuse!

Recycling Does the code you're writing right now look at lot like some of the code you wrote yesterday? Don't write it again, refactor the old code into a more generic function. I won't claim to be an angel, but I'm sure that "copy and paste" are tools of the devil.

7.3 From Nishant

Programming Languages and Translators, is my first ever core CS course. Being an EE student it was difficult but seemed interesting enough a course to be taken. The motivation behind taking this course was to learn about programming languages and the working of a translator and the various components a language interpreter/translator/compiler is made of. Another reason was to get involved in a programming project to get a feel of programming and think as a system programmer, designing stuff for the end user.

After taking the class and brain storming with the group members, the thought was to create this language and looked unsurmountable to me. As a niche programmer I learnt a few valuable lessons in regards of thinking as a system programmer. This class and the project has introduced me to many different types of langauges, like Prolog (logical) and obviously OCaml(functional). Learning how to program in OCaml took some doing. Handling the return types and recursion was not easy. But after a full course and the project it is possible to write in OCaml. Adapting to a new language, was a very useful thing learnt as well. While using a system modelling language called Promela for another class, I found it extremely easy to adapt to it. The other most important thing I learnt was errors, their types and their origin. To conclude, this project has taught me the tricks of the trade to "program" and given me the tool, "OCaml".

7.4 From Cheng

From this project, i have learned not only how to contruct an interpreter step by step but also how to collaborate with my teammates.

Basically, there are three most important things i have learned from this project. First, I have acquired a deep understanding of basic concepts covered in the class. This is definitely helpful when i am trying to study a new programming language. Specifically, I can quickly learn how to program using this language and solve the compiling errors as soon as i know about the general features of its compilor (such as naming and scoping rules). Moreover, the success of the project is largely determined by its design. With a good design, coding part will be more easily. However, with a bad design, it will be a painful experience. So it is wise to spend most time on the software design. Finally, teamwork plays a key role in software development. In this project, I have improved my skills of communication. With better communication, the whole team can work more efficiently.

A Appendix: Test Cases

A.1 andTest

Listing 2: andTest Test Case Input

```
wall(4,5);
wall(6,7);

wall($x, 5) {AND:
    $x < 7;
    $x > 2;
}

wall($x, $y) {AND:
         $x < 15;
         $y < 2;
}

main() {
    @print("(", $x, ",", $y, ")");
    wall($x, $y);
}
```

Listing 2: andTest Test Case Input

Listing 3: andTest Test Case Output

```
(4,5)

 ^^^  Solution  ^^^

(6,7)

 ^^^  Solution  ^^^

(2..7,5)

 ^^^  Solution  ^^^

(<15,<2)

 ^^^  Solution  ^^^

No more solutions
```

Listing 3: andTest Test Case Output

A.2 facts

Listing 4: facts Test Case Input

```
foo(4,4);

```

```
foo(symA);

bar($name);

foo(){wall(3);}

main() {OR:
    @print($a, " ", $b);
    foo($a);
    foo($a, $b);
    bar("$a");
}
```

Listing 4: facts Test Case Input

Listing 5: facts Test Case Output

```
symA Any

 ^^^ Solution ^^^

4 4

 ^^^ Solution ^^^

Any Any

 ^^^ Solution ^^^

No more solutions
```

Listing 5: facts Test Case Output

A.3 learnForget1

Listing 6: learnForget1 Test Case Input

```
stack("s1");
stack("s2");
stack("s3");

f() {
    @print("Removing: ", $s);
    @forget( stack($s); );

     $s == "s1";
}

l() {
    @print("Learning: ", $s);
    @learn( stack($s); );

```

```
    $s == 6;
}

main() {OR:
    @print($s);
    f();
    l();
    stack($s);
}
```

Listing 6: learnForget1 Test Case Input

Listing 7: learnForget1 Test Case Output

```
Removing: 's1'
Any

 ^^^ Solution ^^^

Learning: 6
6

 ^^^ Solution ^^^

's2'

 ^^^ Solution ^^^

's3'

 ^^^ Solution ^^^

No more solutions
```

Listing 7: learnForget1 Test Case Output

A.4 main

Listing 8: main Test Case Input

```
/* test to find the no. solutions one gets under given constraints*/

wall(3,4);
wall(4,8);
wall(6,8);
wall($X,$Y) {AND:
    $X>=10;
    $X<=15;
    $Y<=8;
    $Y>=2;
}
wall(){}
```

```

main ()
{
    @print("Wall: ", $x, ",", $y);
    wall($x, $y);
}
```

Listing 8: main Test Case Input

Listing 9: main Test Case Output

```
Wall: 3,4

 ^^^  Solution  ^^^

Wall: 4,8

 ^^^  Solution  ^^^

Wall: 6,8

 ^^^  Solution  ^^^

Wall: 9..16,1..9

 ^^^  Solution  ^^^

No more solutions
```

Listing 9: main Test Case Output

A.5 main fall through

Listing 10: main fall through Test Case Input

```
main() {
    @print("ERROR");
    noexist($y);
}

main() {
    @print("Success");
}
```

Listing 10: main fall through Test Case Input

Listing 11: main fall through Test Case Output

```
Success

 ^^^  Solution  ^^^

```

```
No more solutions
```

Listing 11: main fall through Test Case Output

A.6 mult-main

Listing 12: mult-main Test Case Input

```
main(a,b);

main()
{
    main(a,b);
}

main();
```

Listing 12: mult-main Test Case Input

Listing 13: mult-main Test Case Output

```

 ^^^ Solution ^^^


 ^^^ Solution ^^^

No more solutions
```

Listing 13: mult-main Test Case Output

A.7 neq

Listing 14: neq Test Case Input

```
main() {
    @print($x);
    $x != 8;
}
```

Listing 14: neq Test Case Input

Listing 15: neq Test Case Output

```
!=8

 ^^^ Solution ^^^
```

Listing 15: neq Test Case Output

A.8 not1

Listing 16: not1 Test Case Input

```
wall(10);

wall($y) {
    4 < $y;
    $y < 6;
}

main() {AND:
    @print($y);
    {AND:
        $y > 1;
        $y < 15;
    }
    !wall($y);
}
```

Listing 16: not1 Test Case Input

Listing 17: not1 Test Case Output

```
11..15

 ^^^  Solution  ^^^

1..4

 ^^^  Solution  ^^^

6..9

 ^^^  Solution  ^^^

No more solutions
```

Listing 17: not1 Test Case Output

A.9 plist-twice

Listing 18: plist-twice Test Case Input

```
foo($y, $u, "hello", $y, 6);
```

Listing 18: plist-twice Test Case Input

Listing 19: plist-twice Test Case Output

```
Fatal error: exception Failure("You cannot list the same variable twice in a parameter list
```

Listing 19: plist-twice Test Case Output

A.10 printer test

Listing 20: printer test Test Case Input

```
/*This is a test case   */

/*
  environ1.ul
  The environment being operated in is the list of the
  simulator's facts, then the facts and rules below
*/

/*This is a sample 15x15 environment*/
size(15,15);

@attach("geometry.ul");
/*A wall segment at (5,5)*/
wall(5,5);
/*A wall segment from (1,10) to (5,10)*/
wall($X,$Y) {
        $X > 0;
        $X <= 5;
        $Y == 10;
}
/*A wall that only appears when an agent is at (1,2) or (1,4)*/
wall(1,3) {OR:
        object(1, 2, agent1);
        object(1, 4, agent1);
}

/* A wall that only appears when an agent is at (2,2) or (2,4),
  but stays there after the agent leaves*/
wall(2,3) {
        {OR:
                object(2, 2, agent1);
                object(2, 4, agent1);
        }
        @learn( wall(2,3); );
}
/* An invisible switch appears at (3,3) and dissolves the wall
   at (2,3) when the agent steps on it */
object(3, 3, switchObject) {
        object(3, 3, agent1);
        @forget( wall(2,3); );
}
/*The objective is at (15,15)*/
object($x, $y, wallObject) {
        wall($x, $y);
}

/* These are the icons for each object*/
repr(wallObject, "pix/wall.png");
```

```
repr(switchObject, "pix/switch.png");
repr(goalObject, "pix/goal.png");

/* Agent success if it reaches (15, 15)*/
finish(SuccessAgent1) {
        object(15, 15, agent1);
}

finish(SuccessAgent2) {
        object(13, 15, agent2);
}

/* Fail the simulation if the agent hits a wall*/
finish(Failure) {
        object($x, $y, agent1);
        wall($x, $y);
}
/* Load agent1*/
repr(agent1, "agent1.sl");

/*Place at (1,1) then forget about the agent,
 so the simulator will take over agent management*/
object(1, 1, agent1) {
        @forget( object(1, 1, agent1); );
}

viewRange($x, $y, $viewer, $obj, $rangeMax) {
        object($ViewerX, $ViewerY, $viewer);
        range($x, $y, $ViewerX, $ViewerY, $range);
        0 <= $range ;
        $range <= $rangeMax;
        object($x, $y, $obj);
}
viewAccessRule(agent1);
/*How far can agents see?
 This is defined in geometry.ul*/
view($x, $y, $viewer, $obj) {
        viewRange($x, $y, $viewer, $obj, 1);
}

repr(agent2, "agent2.ul");

peers(agent1);
peers(agent2);
```

Listing 20: printer test Test Case Input

Listing 21: printer test Test Case Output

```
size(15,15);
@attach("geometry.ul");
wall(5,5);
```

```
wall($X,$Y) {AND:
$X>0;
$X<=5;
$Y=10;
}
wall(1,3) {OR:
object(1,2,agent1);
object(1,4,agent1);
}
wall(2,3) {AND:
{OR:
object(2,2,agent1);
object(2,4,agent1);
}
@learn(wall(2,3);)
}
object(3,3,switchObject) {AND:
object(3,3,agent1);
@forget(wall(2,3);)
}
object($x,$y,wallObject) {AND:
wall($x,$y);
}
repr(wallObject,"pix/wall.png");
repr(switchObject,"pix/switch.png");
repr(goalObject,"pix/goal.png");
finish(SuccessAgent1) {AND:
object(15,15,agent1);
}
finish(SuccessAgent2) {AND:
object(13,15,agent2);
}
finish(Failure) {AND:
object($x,$y,agent1);
wall($x,$y);
}
repr(agent1,"agent1.sl");
object(1,1,agent1) {AND:
@forget(object(1,1,agent1);)
}
viewRange($x,$y,$viewer,$obj,$rangeMax) {AND:
object($ViewerX,$ViewerY,$viewer);
range($x,$y,$ViewerX,$ViewerY,$range);
0<=$range;
$range<=$rangeMax;
object($x,$y,$obj);
}
viewAccessRule(agent1);
view($x,$y,$viewer,$obj) {AND:
viewRange($x,$y,$viewer,$obj,1);
}
```

```
repr(agent2,"agent2.ul");
peers(agent1);
peers(agent2);
```

Listing 21: printer test Test Case Output

A.11 prsimple

Listing 22: prsimple Test Case Input

```
foo(4,5);
@learn("$x");
foo(){OR:
@learn(wall(4,5););
//@forget(  wall(2,3);  );
}
```

Listing 22: prsimple Test Case Input

Listing 23: prsimple Test Case Output

```
foo(4,5);
@learn("$x");
foo() {OR:
@learn(wall(4,5);)
}
```

Listing 23: prsimple Test Case Output

A.12 prstrings

Listing 24: prstrings Test Case Input

```
foo(4, "asdf");
foo("#@!$%");
foo(4, "//as oiuwer//2356 asdoiulkj ouweoij:::;popi%$%^_)+(*^%&$$$^%&(*_^%&$@#^%$&");
```

Listing 24: prstrings Test Case Input

Listing 25: prstrings Test Case Output

```
foo(4,"asdf");
foo("#@!$%");
foo(4,"//as oiuwer//2356 asdoiulkj ouweoij:::;popi%$%^_)+(*^%&$$$^%&(*_^%&$@#^%$&");
```

Listing 25: prstrings Test Case Output

A.13 range

Listing 26: range Test Case Input

```
/* test to find how the range cases work*/

foo($x,$y) {
    10 >= $x;
    1 <= $x;
     $y>9;
    19 >$y;
}

bar($z) {
    $z < 50;
    10 < $z;
}

main() {
    {OR:
        foo($x,$y);
        bar($z);
        @print("the solutions for $x ",$x,
               " the solutions for $y ",$y,
               " the solutions for $z ",$z);
    }


}
/* conclusions: If an infinte range is given the interpretor gives no solution
                The values returned are exclusive in a range
                so for  10 >= $x;
                        1 <= $x;
                the interpreter returns a range 0..11
*/
```

Listing 26: range Test Case Input

Listing 27: range Test Case Output

```
the solutions for $x 0..11 the solutions for $y 9..19 the solutions for $z Any

 ^^^  Solution  ^^^

the solutions for $x Any the solutions for $y Any the solutions for $z 10..50

 ^^^  Solution  ^^^

No more solutions
```

Listing 27: range Test Case Output

A.14 sim dot1

Listing 28: sim dot1 Test Case Input

```
size(10, 10);
wall(?, 7);
goal(6, 6);

imove("UP");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");

move($dir) {
    imove($dir);
}

move("UP");

agent("d", "tests/agents/dot1.ul");
```

Listing 28: sim dot1 Test Case Input

Listing 29: tests/agents/dot1.ul

```
move($dir) {
    $e.@forget1( imove($dir); );
    @print($e, " says move ", $dir);
    env($e);
    $e.move($dir);
}
```

Listing 29: tests/agents/dot1.ul

Listing 30: sim dot1 Test Case Output

```
Agent says move 'UP'
d: Moving UP

==== Turn 1 ====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
d . . . . . . . . .
. . . . . . . . . .

```

```
Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 2 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. d . . . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 3 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . d . . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 4 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . d . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 5 =====
. . . . . . . . . .
. . . . . . . . . .
```

```
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . d . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 6 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

===== Turn 7 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

===== Turn 8 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .
```

```
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

==== Turn 9 ====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . d . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

Simulation over: d wins!!! Successfully reach the goal at position (6,6)
```

Listing 30: sim dot1 Test Case Output

A.15 sim dot2

Listing 31: sim dot2 Test Case Input

```
size(10, 10);
wall(?, 7);
goal(6, 6);

imove("UP");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");

move($dir) {
    @forget1( imove($dir); );
    imove($dir);
}

move("UP");

agent("d", "tests/agents/delg_to_env.ul");
```

Listing 31: sim dot2 Test Case Input

Listing 32: tests/agents/delg_to_env.ul

```
env(yo);

move($dir) {
    @print($e, " says move ", $dir);
    env($e);
    $e.move($dir);
}
```

Listing 32: tests/agents/delg_to_env.ul

Listing 33: sim dot2 Test Case Output

```
Agent says move 'UP'
d: Moving UP

==== Turn 1 ====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
d . . . . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

==== Turn 2 ====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. d . . . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

==== Turn 3 ====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
```

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . d . . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 4 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . d . . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 5 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . d . . . . .
. . . . . . . . . .

Agent says move 'RIGHT'
d: Moving RIGHT

===== Turn 6 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .

```

```
Agent says move 'UP'
d: Moving UP

===== Turn 7 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

===== Turn 8 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . d . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

===== Turn 9 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . d . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

Agent says move 'UP'
d: Moving UP

Simulation over: d wins!!! Successfully reach the goal at position (6,6)
```

Listing 33: sim dot2 Test Case Output

A.16 sim my loc

Listing 34: sim my loc Test Case Input

```
size(10, 10);
wall(?, 7);
goal(6, 6);

move("UP") {
    @print("My location: ", $x, ",", $y);
    loc($x, $y);
    $y < 6;
}

move("RIGHT");

main() {
    @print($dir);
    move($dir);
}
```

Listing 34: sim my loc Test Case Input

Listing 35: sim my loc Test Case Output

```
My location: 1,1
x: Moving UP

===== Turn 1 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
x . . . . . . . . .
. . . . . . . . . .

My location: 1,2
x: Moving UP

===== Turn 2 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
x . . . . . . . . .
```

```
. . . . . . . . . .
. . . . . . . . . .

My location: 1,3
x: Moving UP

===== Turn 3 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
. . . . . . . . . .
x . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

My location: 1,4
x: Moving UP

===== Turn 4 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . . # . . . .
x . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

My location: 1,5
x: Moving UP

===== Turn 5 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
x . . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

x: Moving RIGHT

===== Turn 6 =====
```

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. x . . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

x: Moving RIGHT

===== Turn 7 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . x . . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

x: Moving RIGHT

===== Turn 8 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . x . # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .

x: Moving RIGHT

===== Turn 9 =====
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
| | | | | | | | | |
. . . . x # . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

```
. . . . . . . . . . .

x: Moving RIGHT

Simulation over: x wins!!! Successfully reach the goal at position (6,6)
```

Listing 35: sim my loc Test Case Output

A.17 sim ndot2

Listing 36: sim ndot2 Test Case Input

```
size(3,3);

wall(3,3);

goal(3,1);

disallow("DOWN");
disallow("LEFT");
disallow("UP");

agent("x", "tests/agents/ndot2.ul");
```

Listing 36: sim ndot2 Test Case Input

Listing 37: tests/agents/ndot2.ul

```
move($dir) {
    {OR:
        $dir == "UP";
        $dir == "DOWN";
        $dir == "LEFT";
        $dir == "RIGHT";
    }

    @print("Moving: ", $dir);
    env($e);
    !$e.disallow($dir);
}
```

Listing 37: tests/agents/ndot2.ul

Listing 38: sim ndot2 Test Case Output

```
Moving: 'RIGHT'
x: Moving RIGHT

==== Turn 1 ====
. . |
. . .
. x #
```

```

Moving: 'RIGHT'
x: Moving RIGHT

Simulation over: x wins!!! Successfully reach the goal at position (3,1)
```

Listing 38: sim ndot2 Test Case Output

A.18 sim two test

Listing 39: sim two test Test Case Input

```
/*this is test of simulator, it output the walls and trace of agent into agent1.dat*/

/*   ENV CODE... PLAYER——DON'T CHANGE OR LOOK AT ME!!!*/
goal(4,4);
size(20,20);
wall(12,4);
wall(4,9);
wall(6,8);
wall($X,$Y) {AND:
        $X>=4;
        $X<=6;
        $Y<=15;
        $Y>=10;
        }


agent("x", "tests/agents/agent1.ul");
agent("y", "tests/agents/agent2.ul");
```

Listing 39: sim two test Test Case Input

Listing 40: tests/agents/agent1.ul

```
imove("UP");
imove("UP");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");

move($dir) {
    @forget1( imove($dir); );
    imove($dir);
}

move("DOWN");

main() {
    @print($d);
    move($d);
```

```
}
```

Listing 40: tests/agents/agent1.ul

Listing 41: tests/agents/agent2.ul

```
imove("RIGHT");
imove("RIGHT");
imove("LEFT");
imove("UP");
imove("RIGHT");
imove("UP");
imove("UP");
imove("RIGHT");

move($dir) {
    @forget1( imove($dir); );
    imove($dir);
}

move("DOWN");
```

Listing 41: tests/agents/agent2.ul

Listing 42: sim two test Test Case Output

```
x: Moving UP
y: Moving RIGHT

==== Turn 1 ====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
x . . . . . . . . . . . . . . . . . . .
. y . . . . . . . . . . . . . . . . . .

x: Moving UP
```

```
y: Moving RIGHT

===== Turn 2 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
x . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . y . . . . . . . . . . . . . . . . .

x: Moving RIGHT
y: Moving LEFT

===== Turn 3 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. x . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. y . . . . . . . . . . . . . . . . . .

x: Moving RIGHT
y: Moving UP
```

```

===== Turn 4 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . x . . . . . . . . . . . . . . . . .
. y . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT
y: Moving RIGHT

===== Turn 5 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . x . . . . . . . . . . . . . . . .
. . y . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT
y: Moving UP

```

```
===== Turn 6 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . y . x . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving DOWN
y: Moving UP

===== Turn 7 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . y # . . . . . . . | . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . x . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving DOWN
y: Moving RIGHT

Simulation over: y wins!!! Successfully reach the goal at position (4,4)
```

Listing 42: sim two test Test Case Output

A.19 simulator test

Listing 43: simulator test Test Case Input

```
/*this is test of simulator, it output the walls and trace of agent into agent1.dat*/

/*   ENV CODE... PLAYER——DON'T CHANGE OR LOOK AT ME!!!*/
goal(4,4);
size(20,20);
wall(12,4);
wall(4,9);
wall(6,8);
wall($X,$Y) {AND:
        $X>=4;
        $X<=6;
        $Y<=15;
        $Y>=10;
        }


agent("x", "tests/agents/agent1.ul");
```

Listing 43: simulator test Test Case Input

Listing 44: tests/agents/agent1.ul

```
imove("UP");
imove("UP");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");
imove("RIGHT");

move($dir) {
    @forget1( imove($dir); );
    imove($dir);
}

move("DOWN");

main() {
    @print($d);
    move($d);
}
```

Listing 44: tests/agents/agent1.ul

Listing 45: simulator test Test Case Output

```
x: Moving UP

===== Turn 1 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
x . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving UP

===== Turn 2 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
x . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT

```

```
===== Turn 3 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. x . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT

===== Turn 4 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . x . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT

===== Turn 5 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
```

```
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . x . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving RIGHT

===== Turn 6 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . . x . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving DOWN

===== Turn 7 =====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
```

```
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . x . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .

x: Moving DOWN

==== Turn 8 ====
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | | | . . . . . . . . . . . . . .
. . . | . . . . . . . . . . . . . . . .
. . . . . | . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . # . . . . . . . | . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . .
. . . . x . . . . . . . . . . . . . . .

x: Moving DOWN

Simulation over: x hits the x margin and Game over!!!
```

Listing 45: simulator test Test Case Output

A.20 sprint1

Listing 46: sprint1 Test Case Input

```
wall(4,4);
```

```
wall(6,3);
move("UP");
move("UP");
move("UP");
move("RIGHT");
move("RIGHT");
move("RIGHT");
face("foo");

face(symA);

main() {OR:
     @print("Wall: ", $x, ", ", $y);
     wall($x, $y);
     face($y);
     move($y);
}
```

Listing 46: sprint1 Test Case Input

Listing 47: sprint1 Test Case Output

```
Wall: 4, 4

 ^^^ Solution ^^^

Wall: 6, 3

 ^^^ Solution ^^^

Wall: Any, 'foo'

 ^^^ Solution ^^^

Wall: Any, symA

 ^^^ Solution ^^^

Wall: Any, 'UP'

 ^^^ Solution ^^^

Wall: Any, 'RIGHT'

 ^^^ Solution ^^^

No more solutions
```

Listing 47: sprint1 Test Case Output

B Appendix: Code Listings

B.1 parser.mly

Listing 48: CμLOG Parser

```
/* Original author:Devesh Dedhia*/

%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE NOT ASSIGN EOF COMMENT
%token LBRACE RBRACE LPAREN RPAREN
%token ARROPEN ARRCLOSE AT DOT
%token SEMICOLON OR AND COMMA COLON QUOTE QUESTION
%token <string> ID VARIABLE STRING
%token <int> DIGIT

/* Comparison tokens */
%token EQ GT LT GEQ LEQ NEQ

%nonassoc EQ
%left NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE


%start program
%type < Ast.program> program

%%

program:
  main { Program($1) }

main:
  EOF { [] }
| top main { $1 :: $2 }

top:
  culogRule            { $1 }  /* the program consists of facts rules and directives*/
| culogFact            { $1 }
| culogDirective       { $1 }

culogFact:                        /*wall(2,2);*/
  ID LPAREN param_list RPAREN SEMICOLON          { Fact($1, Params(List.rev $3) ) }


culogRule:                        /* wall(){}*/
  ID LPAREN param_list RPAREN block              { Rule($1, Params(List.rev $3), $5 ) }

culogDirective:                   /*@print(" these are global directives");*/
```

```
 AT ID LPAREN param_list RPAREN SEMICOLON        { GlobalDirective($2, Params(List.rev $4))


param_list:
  {[]}
| param                          {[$1]}
| param_list COMMA param         {$3::$1} /* Params seprated by Commas*/

param:
  VARIABLE               { Var($1) }      /* $x,$agent*/
| ID                     { Sym($1) }      /* symbA */
| DIGIT                  { Lit($1) }      /* 0...9*/
| PLUS DIGIT             { Lit($2) }      /* +0...+9*/
| MINUS DIGIT            { Lit(-1*$2) }   /* -0...-9*/
| STRING                 { Str($1) }      /* "STRINGS"*/
| array                  { Arr($1) }      /* [$x,$y]*/
| QUESTION               { Ques }         /* ?- to indicate Anonymous variables */

array:
  ARROPEN param_list ARRCLOSE { Array( List.rev $2 ) }

block:
  LBRACE stmt_list RBRACE { Block("AND", Stmts( $2 ) ) } /* Default reduction operator is A
| LBRACE ID COLON stmt_list RBRACE{ Block($2, Stmts( $4 )) } /* any operator can be used*/

stmt_list:
  /*nothing*/             { [] }
 | statement stmt_list  { $1 :: $2 }

statement:               /* statements can be sub-blocks, facts,comparison statements */
                         /* directives statements or dot operator statements*/
  block {$1}
| ID LPAREN param_list RPAREN SEMICOLON                       {Eval($1, Params(List.rev $3)) }
| NOT ID LPAREN param_list RPAREN SEMICOLON                   {NEval($2, Params(List.rev $4)) }
| VARIABLE DOT ID LPAREN param_list RPAREN SEMICOLON          {Dot2($1,$3,Params(List.rev $5))}
| NOT VARIABLE DOT ID LPAREN param_list RPAREN SEMICOLON{NDot2($2,$4,Params(List.rev $6))}
| VARIABLE DOT AT ID LPAREN direc_list RPAREN SEMICOLON {Dot1($1,$4,(List.rev $6))}
| expr EQ expr  SEMICOLON                                     {Comp($1,Eq,$3)}
| expr NEQ expr SEMICOLON                                     {Comp($1,Neq,$3)}
| expr GT expr  SEMICOLON                                     {Comp($1,Gt,$3)}
| expr LT expr  SEMICOLON                                     {Comp($1,Lt,$3)}
| expr GEQ expr SEMICOLON                                     {Comp($1,Geq,$3)}
| expr LEQ expr SEMICOLON                                     {Comp($1,Leq,$3)}
| AT ID LPAREN param_list RPAREN SEMICOLON                    {Directive($2, Params(List.rev $4)
| AT ID LPAREN direc_list_first RPAREN SEMICOLON              {DirectiveStudy($2,(List.rev $4))}

direc_list_first:
 directive SEMICOLON direc_list                               { $1 :: $3 }

direc_list:
  {[]}
```

```
| directive SEMICOLON direc_list                    { $1 :: $3 }

directive:
 ID LPAREN param_list RPAREN                        {($1,Params(List.rev $3))}


expr:
    expr PLUS   expr    { Binop($1, Plus,   $3) }    /* $X+4*/
  | expr MINUS  expr    { Binop($1, Minus,  $3) }    /*3-4*/
  | expr TIMES  expr    { Binop($1, Mult,   $3) }    /* $x*4 */
  | expr DIVIDE expr    { Binop($1, Divide, $3) }    /* $x/$y */
  | DIGIT               { ELit($1) }
  | MINUS DIGIT         { ELit(-1*$2) }
  | PLUS DIGIT          { ELit($2) }
  | VARIABLE            { EVar($1) }
  | STRING              { EStr($1) }
  | ID                  { EId($1) }
```

Listing 48: CμLOG Parser

B.2 scanner.mll

Listing 49: CμLOG Scanner

```
{ open Parser }
rule token = parse
    [' ' '\t' '\r' '\n'] { token lexbuf }
  | "/*"      { comment lexbuf }
  | "//"      { linecomment lexbuf }
  | '('       { LPAREN }
  | ')'       { RPAREN }
  | '{'       { LBRACE }
  | '}'       { RBRACE }
  | ';'       { SEMICOLON }
  | ','       { COMMA }
  | '+'       { PLUS }
  | '-'       { MINUS }
  | '*'       { TIMES }
  | '/'       { DIVIDE }
  | "==" | "="  { EQ }
  | "!="      { NEQ }
  | '<'       { LT }
  | "<="      { LEQ }
  | ">"       { GT }
  | ">="      { GEQ }
  | '@'       { AT }
  | '.'       { DOT }
  | ':'       { COLON }
  | '['       { ARROPEN }
  | ']'       { ARRCLOSE }
  | '"'       { QUOTE}
```

```
  | '?'        { QUESTION }
  | '!'        { NOT}
  | '$'['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as var { VARIABLE(var) } (*variables
  | ['0'-'9']+ as lxm { DIGIT(int_of_string lxm) }
  | [ 'a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* as lxm { ID(lxm) } (* An ID must star
  | '"'([^ '"''\t' '\r' '\n']+ as lxm) '"'{STRING(lxm)}                   (* any thing declar
                                                                            quotes is a string
  | eof        { EOF }

and  comment = parse
     "*/" { token lexbuf }
  | _  { comment lexbuf }

and linecomment = parse
     ['\r' '\n'] {token lexbuf}
  | _ {linecomment lexbuf}
```

Listing 49: CμLOG Scanner

B.3 ast.mli

Listing 50: CμLOG AST

```
(* Original author: Cheng Cheng
   Edited          : Devesh Dedhia
   support added to include directives *)

type operator = Plus | Minus | Mult | Divide
type compoperator = Lt | Leq | Gt | Geq | Eq | Neq

(* type study = learn | forget *)

type param =
     Lit   of int                          (* 0...9*)
  | Sym   of string                        (* sym1*)
  | Var   of string                        (* $X *)
  | Str   of string                        (*"asdf"*)
  | Arr   of params                        (* [2,$x,symb1]*)
  | Ques

and params =
     Params of param list
  | Array of param list

type expr =
     Binop of expr*operator*expr           (* 0>$X>=5  $X==$Y  5!=4*)
  | ELit  of int                           (* 0...9*)
  | EVar  of string                        (* $X *)
  | EStr  of string                        (*"asdf"*)
  | EId   of string                        (* sym1*)

```

```
type eval = string*params

type stmt =
    Block of string*stmts                       (*  {.....}  *)
  | Comp of expr*compoperator*expr              (*  $5+5<$4   $a=5,$b=6;  *)
  | NEval of string*params                      (*!wall(4,5)  *)
  | Eval of eval                                (*wall(4,5)  *)
  | DirectiveStudy of string*(eval list)        (*@learn(wall(4,5);)*)
  | Directive of string*params                  (*@print("dfdsf");*)
  | Dot1 of string*string*(eval list)           (*$agent.@learn(wall(4,5);)  *)
  | Dot2 of string*string*params                (*  $env.view($X,$Y,$Obj)*)
  | NDot2 of string*string*params               (*  !env.view($X,$Y,$Obj)*)

and stmts=Stmts of stmt list   (*  statment1;statment2;statement3;  *)


type ruleFact =
    Rule of string * params * stmt        (*  wall(3,4){AND:  ....}*)
  | Fact of string * params               (*  wall(2,2);*)
  | GlobalDirective of string*params (*@attach("dfsfsa")*)  (*@print("ddafafa")*)


type program = Program of ruleFact list
```

Listing 50: CμLOG AST

B.4 printer.ml

Listing 51: CμLOG AST Printer

```
(*
*   printer.ml
*
*   Originally written in group, typed by John Demme
*   Updated as necessary by everyone
*)

open Ast

let string_of_compoperator = function
    Lt  -> "<"
  | Leq -> "<="
  | Gt  -> ">"
  | Geq -> ">="
  | Eq  -> "="
  | Neq -> "!="

let string_of_operator = function
    Plus   -> "+"
  | Minus  -> "-"
  | Mult   -> "*"
```

```
  | Divide -> "/"


let rec string_of_expr = function
    Binop(e1, o, e2) -> (string_of_expr e1) ^ (string_of_operator o) ^ (string_of_expr e2)
  | ELit(i) -> string_of_int i
  | EVar(s) -> s
  | EStr(s) -> s
  | EId(s)  -> s

let rec string_of_param = function
    Lit(i) -> string_of_int i
  | Sym(s) -> s
  | Var(s) -> s
  | Str(s) -> "\""^s^"\""
  | Arr(a) -> "["^string_of_params a^"]"
  | Ques   -> "?"


and string_of_params = function
    Params(pList) -> String.concat "," (List.map string_of_param pList)
  | Array(pList) -> String.concat "," (List.map string_of_param pList)

let rec string_of_stmts = function
    Stmts(sList) -> String.concat "\n" (List.map string_of_stmt sList)
and string_of_stmt = function
    Block(red, stmts) -> "{" ^ red ^ ":\n" ^ (string_of_stmts stmts) ^ "\n}"
  | Comp(e1, c, e2) -> (string_of_expr e1) ^ (string_of_compoperator c)
      ^ (string_of_expr e2) ^ ";"
  | Eval(name, ps) -> name ^ "(" ^ (string_of_params ps) ^ ");"
  | NEval(name1, ps1) -> "!" ^ name1 ^ "(" ^ (string_of_params ps1) ^ ");"
  | DirectiveStudy(name,stmts) -> "@"^name ^ "(" ^
      (string_of_stmts (Stmts (List.map (fun a -> Eval(a)) stmts))) ^ ")"
  | Directive(name,params) -> "@"^name^"("^(string_of_params params)^")"
  | Dot1(str1,str2,stmts) -> str1^"."^"@"^str2^"("^
      (string_of_stmts (Stmts (List.map (fun a -> Eval(a)) stmts))) ^ ")"
  | Dot2(str1,str2,ps) -> str1^"."^str2^"("^(string_of_params ps)^");"
  | NDot2(str1,str2,ps) -> "!"^str1^"."^str2^"("^(string_of_params ps)^");"


let string_of_ruleFact = function
    Rule(name, params, stmt) -> name ^ "(" ^ (string_of_params params) ^ ") "
      ^ (string_of_stmt stmt)
  | Fact(name, params)      -> name ^ "(" ^ (string_of_params params) ^ ");"
  | GlobalDirective(name,ps) ->"@"^name ^ "(" ^ (string_of_params ps) ^ ");"

let string_of_program = function
    Program(ruleList) -> String.concat "\n" (List.map string_of_ruleFact ruleList) ^ "\n"
```

Listing 51: CμLOG AST Printer

B.5 tst.mli

Listing 52: CμLOG Translated Syntax Tree

```
(*
  This is a simpler, much more restrictive version of the AST.
  It is much easier for the interpreter to deal with, and is relatively
  easy to obtain given an AST.  The trans.ml module translates from the AST
  to this TST.

  Copied and modified from ast.mli

  Written by John Demme

*)
type param =
    Lit   of int
  | Sym   of string
  | Var   of int
  | Anon
  | Str   of string
  | Arr   of param list

and params = param list

type expr =
  | ELit  of int

type eval = string*params
type var  = int

type stmt =
    Block of string*stmts                         (* {.....} *)
  | Comp of var*Ast.compoperator*expr             (* $5+5<$4   $a=5,$b=6; *)
  | StrComp of var*string
  | SymComp of var*string
  | NEval of eval                                 (*!wall(4,5) *)
  | Eval of eval                                  (*wall(4,5) *)
  | DirectiveStudy of string*(eval list)          (*@learn(wall(4,5);)*)
  | Directive of string*params                    (*@print("dfdsf");*)
  | Dot1 of int*string*(eval list)                (*$agent.@learn(wall(4,5);) *)
  | Dot2 of int*string*params                     (* env.view($X,$Y,$Obj)*)
  | NDot2 of int*string*params                    (* !env.view($X,$Y,$Obj)*)

and stmts=stmt list  (* statment1;statment2;statement3; *)

type ruleFact =
  | Rule of string * params * int * stmt * stmt list
  | Fact of string * params


type program = ruleFact list
```

Listing 52: CμLOG Translated Syntax Tree

B.6 trans.ml

Listing 53: CμLOG AST to TST Translator

```
(* Functions to modify the AST slightly
   to make parsing it easier for the interpreter.

   Static checking happens here as well.

   John Demme
*)

open Ast

module StringMap = Map.Make(String);;

(* Give me the number of items in a StringMap *)
let map_length sMap =
  let fLength k a b =
    b + 1
  in
    StringMap.fold fLength sMap 0
;;

(* Use me with List.fold to get a maximum index *)
let max_index s i l =
  if i > l
  then i
  else l
;;

(* Print all items in a StringMap *)
let smPrint key a =
  Printf.printf "%s: %d\n" key a; ()
;;

(* Get a variable name to variable number binding from a rule *)
let getBindings mRule =
  (* TODO: Many of these functions could be made nicer using stuff like List.fold *)
  let add_binding var bindings =
    if (StringMap.mem var bindings) then
      bindings
    else
      (StringMap.add var (map_length bindings) bindings)
  in
  let rec get_params_var_mapping params bindings =
    let len = map_length bindings in
```

```
      match params with
          [] -> bindings
        | Var(name) :: tail ->
            if (StringMap.mem name bindings)
            then failwith "You cannot list the same variable twice in a parameter list"
            else get_params_var_mapping tail (StringMap.add name len bindings)
        | i :: tail ->
            get_params_var_mapping tail (StringMap.add (string_of_int len) len bindings)
  in
  let rec get_eval_var_mapping params bindings =
    match params with
        [] -> bindings
      | Var(name) :: tail ->
          get_eval_var_mapping tail
            (add_binding name bindings)
      | _ :: tail -> get_eval_var_mapping tail bindings
  in
  let rec get_expr_var_mapping e bindings =
    match e with
        EVar(name) -> add_binding name bindings
      | Binop(a, op, b) -> get_expr_var_mapping a (get_expr_var_mapping b bindings)
      | _ -> bindings
  in
  let rec get_stmts_var_mapping stmts bindings =
    match stmts with
        [] -> bindings
      | Block(redOp, Stmts(stmts)) :: tail ->
          get_stmts_var_mapping tail (get_stmts_var_mapping stmts bindings)
      | Comp(expr1, compOp, expr2) :: tail ->
          get_stmts_var_mapping tail
            (get_expr_var_mapping expr1 (get_expr_var_mapping expr2 bindings))
      | Eval(name, Params(params)) :: tail ->
          get_stmts_var_mapping tail (get_eval_var_mapping params bindings)
      | Directive(name, Params(params)) :: tail ->
          get_stmts_var_mapping tail (get_eval_var_mapping params bindings)
      | _ :: tail ->
          get_stmts_var_mapping tail bindings
  in
    match mRule with
        Rule(name, Params(params), stmt) ->
          get_stmts_var_mapping [stmt] (get_params_var_mapping params StringMap.empty)
      | Fact(name, Params(params)) ->
          (get_params_var_mapping params StringMap.empty)
      | _ -> StringMap.empty
;;

(* Translate a rule or fact from AST to TST *)
let translate_rule mRule =
  let bindings = getBindings mRule in
  let bget name =
    StringMap.find name bindings
```

```
in
  (* Translate paramaters using these bindings *)
let translate_params params =
  let translate_param param =
    match param with
        Var(name) -> Tst.Var(bget name)
      | Lit(i)    -> Tst.Lit(i)
      | Sym(s)    -> Tst.Sym(s)
      | Str(s)    -> Tst.Str(s)
      | Arr(prms) -> failwith "Sorry, arrays are unsupported"
      | Ques      -> Tst.Anon
  in
    List.map translate_param params
in
let rec translate_stmts stmts =
  (* Move the variable to one side, and simplyify to a constant on the other *)
  let translate_comp expr1 op expr2 =
    (* Can this expression be numerically reduced? *)
    let rec can_reduce expr =
      match expr with
          ELit(i) -> true
        | Binop(e1, op, e2) -> (can_reduce e1) && (can_reduce e2)
        | _ -> false
    in
    (* Give me the reverse of an operator *)
    let rev_op op =
      match op with
          Lt -> Gt
        | Gt -> Lt
        | Eq -> Eq
        | Neq -> Neq
        | Geq -> Leq
        | Leq -> Geq
    in
      (* Translate a comparison where the variable is on the LHS *)
    let translate_comp_sv var_expr op expr =
      (* Reduce a constant expression to a literal *)
      let reduce expr =
          let rec num_reduce expr =
            match expr with
                ELit(i) -> i
              | Binop(e1, op, e2) ->
                  (let re1 = num_reduce e1 in
                   let re2 = num_reduce e2 in
                     match op with
                         Plus -> re1 + re2
                       | Minus -> re1 - re2
                       | Mult -> re1 * re2
                       | Divide -> re1 / re2)
              | _ -> failwith "Internal error 8"
          in
```

```
              Tst.ELit(num_reduce expr)
        in
          match var_expr with
              EVar(name) ->
                (* Can we numerically reduce the RHS? *)
                if not (can_reduce expr)
                then
                  (* If not, if better be a simple string of symbol comparison *)
                  match (op, expr) with
                      Eq, EStr(s) -> Tst.StrComp(bget name,s)
                    | Eq, EId (s) -> Tst.SymComp(bget name,s)
                    | _ -> failwith "Unsupported comparison"
                else
                  Tst.Comp(bget name, op, reduce expr)
            | _ -> failwith "Comparison unsupported"
      in
        (* Does this expression have a variable *)
      let rec has_var expr =
        match expr with
            EVar(i) -> true
          | Binop(e1, op, e2) -> (has_var e1) || (has_var e2)
          | _ -> false
      in
        (* Check each expression for variables *)
      let ev1 = has_var expr1 in
      let ev2 = has_var expr2 in
        if ev1 && ev2
        then failwith "Comparisons with multiple variables are unsupported."
        else if (not ev1) && (not ev2)
        then failwith "Error: Comparison is constant"
        else if ev1
        then translate_comp_sv expr1 op expr2
        else translate_comp_sv expr2 (rev_op op) expr1
    in
      (* translate a list of evals *)
    let mapEvList evList =
      List.map
        (fun ev ->
           match ev with
               (name, Params(plist)) ->
                 (name,
                  translate_params plist)
             | (name, Array(alist)) ->
                 failwith "Syntax error, arrays not permitted as params")
        evList
    in
      (* Translate a single statement *)
    let rec replace_stmt stmt =
      match stmt with
          Block(redOp, Stmts(stmts))->
            Tst.Block(redOp, translate_stmts stmts)
```

```
        | Comp(expr1, compOp, expr2) ->
            translate_comp expr1 compOp expr2
        | Eval(name, Params(params)) ->
            Tst.Eval(name, translate_params params)
        | NEval(name, Params(params)) ->
            Tst.NEval(name, translate_params params)
        | Dot2(vname, pred, Params(params)) ->
            Tst.Dot2(bget vname, pred, translate_params params)
        | NDot2(vname, pred, Params(params)) ->
            Tst.NDot2(bget vname, pred, translate_params params)
        | Directive(n, Params(params)) ->
            Tst.Directive(n, translate_params params)
        | DirectiveStudy(n, evList) ->
            Tst.DirectiveStudy(n, mapEvList evList)
        | Dot1(vname, n, evList) ->
            Tst.Dot1(bget vname, n, mapEvList evList)
        | _ -> failwith "Unsupported statement"
    in
      List.map replace_stmt stmts
  in
    (* Given a list of TST statements, prune out the ones
       which have no effect on the solutions *)
  let rec filterSE stmts =
    match stmts with
        [] -> []
      | Tst.Block(redOp, stmts) :: tail ->
          Tst.Block(redOp, filterSE stmts) :: filterSE tail
      | Tst.Directive(_, _) :: tail ->
          filterSE tail
      | Tst.DirectiveStudy(_, _) :: tail ->
          filterSE tail
      | Tst.Dot1(_, _, _) :: tail ->
          filterSE tail
      | head :: tail ->
          head :: filterSE tail
  in
    (* Given a list of TST statements, prune out the ones
       which have some effect on the solutions *)
  let rec filterNSE stmts =
    match stmts with
        [] -> []
      | Tst.Block(redOp, stmts) :: tail ->
          List.append (filterNSE stmts) (filterNSE tail)
      | Tst.Directive(n, p) :: tail ->
          Tst.Directive(n, p) :: filterNSE tail
      | Tst.DirectiveStudy(n, p) :: tail ->
          Tst.DirectiveStudy(n, p) :: filterNSE tail
      | Tst.Dot1(v, n, p) :: tail ->
          Tst.Dot1(v, n, p) :: filterNSE tail
      | head :: tail ->
          filterNSE tail
```

```
  in
    (* This is the entry point for translate_rule
          ... It's been awhile, so I figured you might need a reminder *)
    match mRule with
        Rule(name, Params(params), stmt) ->
          let replacedStmts = translate_stmts [stmt] in
            Tst.Rule(name,
                     (translate_params params),
                     1 + (StringMap.fold max_index bindings (-1)),
                     List.hd (filterSE replacedStmts),
                     filterNSE replacedStmts)
      | Fact(name, Params(params)) ->
          Tst.Fact(name, (translate_params params))
      | _ -> failwith "Unsupported global directive"
;;


let translate prog =
  match prog with
      Program (rfList) ->
        let newProgram = List.map translate_rule rfList in
          (*print_string (Printer.string_of_program newProgram);*)
          newProgram
;;
```

Listing 53: CμLOG AST to TST Translator

B.7 culog.ml

Listing 54: CμLOG "General Purpose" Interpreter

```
(*
*  culog.ml
*
*  Made by (John Demme)
*  Login   <teqdruid@teqBook>
*
*  Started on  Mon Nov 24 16:03:20 2008 John Demme
*  Last update Mon Nov 24 16:03:27 2008 John Demme
*)

open Interp

let rec iter_sols nxt =
  match nxt with
      NoSolution -> print_string "No more solutions\n"
    | Solution(c,n) ->
        (print_string "\n ^^^  Solution  ^^^ \n\n");
        iter_sols (n ())
;;

```

```
let myDBD db =
  print_string "Database_dump:\n";
  dump_db !db;
  print_string "\n";;

let _ =
  let lexbuf = Lexing.from_channel (open_in Sys.argv.(1)) in
  let program = Parser.program Scanner.token lexbuf in
  let pDB = parseDB(program) in
    (* myDBD pDB; *)
    (let sGen = query pDB (ref []) "main" 0 in
       iter_sols sGen);
    (* myDBD pDB; *)
;;
```

Listing 54: CμLOG “General Purpose” Interpreter

B.8 simulator.ml

Listing 55: CμLOG Simulator

```
(* simulator.ml
 * This is a simulator for entities interaction.
 * Specifically, simulator comprises of two parts.
 * Simulator is divided into 2 parts, one for obataining
 * information about agents and environment(grids) and simulate
 * their behaviors and the other being output driver.
 *
 * Original authors:
 *   Cheng Cheng
 *     (worked on communicating with interpreter and
 *          simulating the interaction)
 *   Nishant Shah
 *     (worked on integrating the output driver with the simulator)
 *
 * Support for loading multiple agents and separating the programs
 *   added by John Demme
 *)

open Interp
open Ast

(* define global references to the parameters of environment*)
let grid_size_ref=ref 1;;
let grid_x_size_ref=ref 1;;
let grid_y_size_ref=ref 1;;
let goal_x_ref=ref 1;;
let goal_y_ref=ref 1;;

(* define data structure of agent*)
type sim_agent = {
```

```
  x    : int;
  y    : int;
  sym  : char;
  db   : database
}

(*define a global array to restore information of wall and positions of agents *)
(* maximum environment size is 100*100 *)
(*  '.'represents empty grid,'|' represents wall*)
let record=
  let f index='.' in
    Array.init 10000 f ;;

let clear_array a=
  for index=0 to (Array.length a)-1 do
    if index= (!grid_y_size_ref- !goal_y_ref)* !grid_x_size_ref+ !goal_x_ref-1 then
      begin
        a.(index)<-'#'
      end
    else a.(index)<-'.'
  done
;;

let sim_exit s =
  Printf.printf "\nSimulation over: %s\n\n" s;
  exit(1)
;;

(*set the size of environment *)
let rec set_size nxt=
  match nxt with
      NoSolution-> ()
    | Solution(c,n)->
        (match c with
              [CEqlInt(x);CEqlInt(y)]-> if x<1||x>100 then failwith "the length of grid is 
              else if y<1||y>100 then failwith "the width of grid is not illegal!!! "
              else
                begin
                  grid_x_size_ref:=x;
                  grid_y_size_ref:=y;
                  grid_size_ref:=x*y
                end
            |_-> ())
;;

(* set the goal agents try to reach*)
let rec set_goal nxt=
  match nxt with
      NoSolution-> ()
    | Solution(c,n)->
        (match c with
```

```
            [CEqlInt(x);CEqlInt(y)]-> if x<1||x> !grid_x_size_ref then failwith "illegal
            else if y<1||y> !grid_x_size_ref then failwith "illegal goal y position"
            else
              begin
                goal_x_ref:=x;
                goal_y_ref:=y;
              end
          |_->())
;;

(* display and output the results after interactions*)
let print_grid oc arr =
  for a=0 to !grid_y_size_ref-1
  do
    for j= !grid_x_size_ref*(a) to !grid_x_size_ref*(a+1)-1
    do
      Printf.fprintf oc "%c " arr.(j)
    done;
    Printf.fprintf oc "\n"
  done
;;

let print_file j arr =
  let file = "Agent"^string_of_int(j)^".dat" in  (* Write message to file *)
  let oc = open_out file in    (* create or truncate file, return channel *)
    (print_grid oc arr;
     close_out oc)
;;

let print_stdout j arr =
  Printf.printf "\n===== Turn %d =====\n" j;
  print_grid stdout arr;
  print_string "\n"
;;

(* create wall in environment*)
let create_wall x_start x_end y_start y_end =
  if x_start<1 || x_end> !grid_x_size_ref then
    failwith "Creating Wall : x position of wall exceeds the grids"
  else if y_start<1 || y_end> !grid_y_size_ref then
    failwith "Creating Wall : y position of wall exceeds the grids"
  else if x_start>x_end || y_start>y_end then failwith "Creating Wall:wrong range!!!"
  else for i=x_start to x_end do
    for j=y_start to y_end do
      record.((!grid_y_size_ref-j)* !grid_x_size_ref+i-1)<-'|'
    done
  done
;;


(* obtain wall information from interpretor and create walls*)
```

```
let rec iter_wall nxt=
  match nxt with
      NoSolution -> ()
    | Solution(c,n) ->
        (match c with
              [Any; Any]-> create_wall 1 !grid_x_size_ref 1 !grid_y_size_ref
            |[CEqlInt(x);Any]-> create_wall x x 1 !grid_y_size_ref
            |[CLT(x);Any]-> create_wall 1 (x-1) 1 !grid_y_size_ref
            |[CGT(x);Any]-> create_wall (x+1) !grid_x_size_ref 1 !grid_y_size_ref
            |[CRange(x1,x2);Any]->create_wall (x1+1) (x2-1) 1 !grid_y_size_ref
            |[Any;CEqlInt(y)]-> create_wall 1 !grid_x_size_ref y y
            |[Any;CLT(y)]-> create_wall 1 !grid_y_size_ref 1 (y-1)
            |[Any;CGT(y)]-> create_wall 1 !grid_x_size_ref (y+1) !grid_y_size_ref
            |[Any;CRange(y1,y2)]-> create_wall 1 !grid_x_size_ref (y1+1) (y2-1)
            |[CEqlInt(x);CEqlInt(y)]-> create_wall x x y y
            |[CEqlInt(x);CLT(y)]-> create_wall x x 1 (y-1)
            |[CEqlInt(x);CGT(y)]-> create_wall x x (y+1) !grid_y_size_ref
            |[CEqlInt(x);CRange(y1,y2)]-> create_wall x x (y1+1) (y2-1)
            |[CLT(x);CEqlInt(y)]-> create_wall 1 (x-1) y y
            |[CLT(x);CLT(y)]-> create_wall 1 (x-1) 1 (y-1)
            |[CLT(x);CGT(y)]-> create_wall 1 (x-1) (y+1) !grid_y_size_ref
            |[CLT(x);CRange(y1,y2)]-> create_wall 1 (x-1) (y1+1) (y2-1)
            |[CGT(x);CEqlInt(y)]-> create_wall (x+1) !grid_x_size_ref y y
            |[CGT(x);CLT(y)]-> create_wall (x+1) !grid_x_size_ref 1 (y-1)
            |[CGT(x);CGT(y)]-> create_wall (x+1) !grid_x_size_ref (y+1) !grid_y_size_ref
            |[CGT(x);CRange(y1,y2)]-> create_wall (x+1) !grid_x_size_ref (y1+1) (y2-1)
            |[CRange(x1,x2);CEqlInt(y)]-> create_wall (x1+1) (x2-1) y y
            |[CRange(x1,x2);CLT(y)]-> create_wall (x1+1) (x2-1) 1 (y-1)
            |[CRange(x1,x2);CGT(y)]-> create_wall (x1+1) (x2-1) (y+1) !grid_y_size_ref
            |[CRange(x1,x2);CRange(y1,y2)]->create_wall (x1+1) (x2-1) (y1+1) (y2-1)
            | _ -> ());
        iter_wall (n ())
;;

(* agent moves towards to a direction*)
let agent_move a direction =
  Printf.printf "%c: Moving %s\n" a.sym direction;
  match direction with
      "UP"    -> {x = a.x; y = a.y + 1; db = a.db; sym = a.sym}
    | "DOWN" -> {x = a.x; y = a.y - 1; db = a.db; sym = a.sym}
    | "LEFT" -> {x = a.x - 1; y = a.y; db = a.db; sym = a.sym}
    | "RIGHT"-> {x = a.x + 1; y = a.y; db = a.db; sym = a.sym}
    | _ ->failwith "No such a direction!"
;;

(*simulator stores the information of agent's move  *)
(*if agent reaches the goal or hits wall, simulator terminates *)
let do_agent_move a =
  let array_index=(!grid_y_size_ref - a.y)* !grid_x_size_ref + a.x-1 in
    if a.x < 1|| a.x > !grid_x_size_ref then (*x position is beyond range *)
      begin
```

```
      let str=(Char.escaped a.sym)^" hits the y margin and Game over!!! " in
      sim_exit str
    end
  else if a.y < 1|| a.y > !grid_y_size_ref then (*y position is beyond range *)
    begin
      let str=(Char.escaped a.sym)^" hits the x margin and Game over!!! " in
      sim_exit str
    end
  else if Array.get record array_index = '|' then
    begin
      let str=(Char.escaped a.sym)^" hits the wall and Game over!!!" in
      sim_exit str
    end
  else if Array.get record array_index = '#' then
    begin
      let str=(Char.escaped a.sym)^" wins!!! Successfully reach the goal at position ("^s
^","^string_of_int(!goal_y_ref)^")"
      in
      sim_exit str
    end
  else if (Array.get record array_index) != '.' then
    begin
      let str="Game over!!! Agents crash!!! at position (" ^ string_of_int(a.x)^","^ strin
      sim_exit str
    end
  else record.(array_index)<- a.sym
;;

(* obtain current position of agent from interpretor and make it move*)
let iter_move agent nxt =
  match nxt with
      NoSolution -> failwith "No Solution"
    | Solution([CEqlStr(dir)], _ ) ->
        let new_agent = agent_move agent dir in
          ignore (do_agent_move new_agent);
          new_agent
    | _ -> failwith "Invalid (or no) move"
;;

(* load the databases of all agents in environment*)
let my_loc_db agent all env =
  ref ([Interp.Fact({name = "loc"; params = [CEqlInt(agent.x);CEqlInt(agent.y)]});
        Interp.Fact({name = "env"; params = [CEqlAgent(env)]})]
       @
        (List.map
            (fun other ->
               Interp.Fact({name = "agent"; params = [CEqlAgent(other.db)]}))
               (List.filter (fun a -> a != agent) all)))
;;
(* simulation function*)
let simulation envDB agents =
```

```
  let rec loop i agents =
    let sGen_size=query envDB (ref []) "size" 2 in
      set_size sGen_size;
      let sGen_goal=query envDB (ref []) "goal" 2 in
        set_goal sGen_goal;
        clear_array record;
        let sGen_wall=query envDB (ref []) "wall" 2 in
          iter_wall sGen_wall;
          let new_agents =
            List.map
              (fun agent ->
                 let sGen_move = query agent.db (my_loc_db agent agents envDB) "move" 1 in
                   iter_move agent sGen_move)
              agents
          in
            print_stdout i record;
            if i>100 then sim_exit "You lose! Can not reach the goal with in 100 steps"
            else loop (i+1) new_agents
  in loop 1 agents
;;


let load_agent db_loc =
  match db_loc with
      (c, s) ->
        let lexbuf1 = Lexing.from_channel (open_in s) in
        let program = Parser.program Scanner.token lexbuf1 in
          {x=1; y=1; sym = (String.get c 0); db = Interp.parseDB(program)}
;;

(*load database of rules and facts for a single agent*)
let load_db db_loc =
  let lexbuf1 = Lexing.from_channel (open_in db_loc) in
  let program = Parser.program Scanner.token lexbuf1 in
    {x=1; y=1; sym = 'x'; db = Interp.parseDB(program)}
;;


let get_agent_locs db =
  let rec gal_int res =
    match res with
        NoSolution -> []
      | Solution([CEqlStr(c); CEqlStr(s)], nxt) -> (c,s) :: (gal_int (nxt()))
      | _ -> failwith "Failed to load agent"
  in
    gal_int (query db (ref []) "agent" 2)
;;


let _ =
  let envDB = load_db Sys.argv.(1) in
```

```
  let agent_locs = get_agent_locs envDB.db in
    if 0 == (List.length agent_locs)
    then simulation envDB.db [envDB]
    else
      let agentDBs = List.map load_agent agent_locs in
        simulation envDB.db agentDBs
;;
```

Listing 55: CμLOG Simulator

B.9 interp.ml

Listing 56: CμLOG Interpreter

```
(*
 *  interp.ml
 *
 *  This guy is the interpreter... It "compiles" the TST to a bunch of OCaml
 *     functions to be run during a query.
 *
 *  You'll quickly be able to tell that this whole method is _begging_ for co-routines.
 *    Lazy evaluation could be beneficial here as well.
 *
 *  This whole guy is written for composability.  Each function takes a database and
 *  variable constraints and returns type "next".  Each composite function (like evals,
 *  and blocks) run their sub functions, look at the results, mutate then appropriately
 *  and return the results as "next"s.  All of this happens lazily.
 *
 *  John Demme
 *
 *)

(* Each variable can be constrained in any of these ways *)
type var_cnst =
    Any
  | FalseSol
  | CEqlSymbol of string
  | CEqlInt    of int
  | CEqlStr    of string
  | CLT        of int
  | CGT        of int
  | CRange     of int*int
  | CEqlAgent  of database

(* Constraints is a list of variables *)
and cnst = var_cnst list

and signature = {
  name   : string;
  params : cnst
}
```

```

(* This guy is how we do our lazy evaluation *)
and next =
    NoSolution
  | Solution of cnst * (unit -> next)

(* A list of these guys makes up our database *)
and rule_fact =
    Fact of signature
  | Rule of signature * (database -> database -> cnst -> next)

and database = rule_fact list ref
;;


let string_of_cnst = function
    Any -> "Any"
  | FalseSol -> "False"
  | CEqlSymbol(s) -> s
  | CEqlStr(s) -> "'" ^ s ^ "'"
  | CEqlInt(i) -> string_of_int i
  | CLT(i)      -> "<" ^ (string_of_int i)
  | CGT(i)      -> ">" ^ (string_of_int i)
  | CRange(a,b)-> (string_of_int a) ^ ".." ^ (string_of_int b)
  | CEqlAgent(a) -> "Agent"
;;

let string_of_eval name vars =
  name ^ "(" ^ String.concat "," (List.map string_of_cnst vars) ^")\n"
;;

(* AND two variable constraints together *)
let cAnd a b =
  let rec and_int a b t =
    match (a, b) with
        (Any, _) -> b
      | (_, Any) -> a
      | (CEqlAgent(a1),  CEqlAgent(a2))  when (a1 == a2) -> a
      | (CEqlSymbol(s1), CEqlSymbol(s2)) when (0 == String.compare s1 s2) -> a
      | (CEqlStr(s1),    CEqlStr(s2))    when (0 == String.compare s1 s2) -> a
      | (CEqlInt(i1),    CEqlInt(i2))    when (i2 == i2) -> a
      | (CEqlInt(i1),    CGT(i2))        when (i1 > i2) -> a
      | (CEqlInt(i1),    CLT(i2))        when (i1 < i2) -> a
      | (CLT(i1),        CLT(i2))        -> CLT(min i1 i2)
      | (CGT(i1),        CGT(i2))        -> CLT(max i1 i2)
      | (CGT(i1),        CLT(i2))        when (i1 < i2) -> CRange(i1, i2)
      | (CRange(l, u),   CEqlInt(i))     when (i > l && i < u) -> b
      | (CRange(l, u),   CGT(i))         when (i < u - 1) -> CRange((max l i), u)
      | (CRange(l, u),   CLT(i))         when (i > l + 1) -> CRange(l, (min u i))
      | (CRange(l1,u1),  CRange(l2,u2))  when (u1 > l2 && u2 > l1) -> CRange((max l1 l2),
      | (_, _) when t -> and_int b a false
```

```
      | (_, _) -> FalseSol
  in
    and_int a b true
;;

let range_to_int c =
  match c with
      CRange(l, u) when (l + 2 == u) -> CEqlInt(l+1)
    | _ -> c
;;

let int_to_range c =
  match c with
      CEqlInt(i) -> CRange(i-1, i+1)
    | _ -> c
;;

(* For each constraint, subtract the second from the first *)
(* TODO:  There are off-by-one errors in here... Fix when you have a clearer head *)
let cMinus b s =
  let cmi b s =
    (* Too many combinations and no play makes Johnny go something something *)
    match (b, s) with
        (_, Any) ->
          []
      | (Any, _) ->
          failwith "Unsupported subtraction - need != constraint"
      | (CEqlSymbol(s1), CEqlSymbol(s2)) when (0 != String.compare s1 s2) ->
          [b; s]
      | (CEqlStr(s1), CEqlStr(s2)) when (0 != String.compare s1 s2) ->
          [b; s]
      | (CEqlInt(i1), CEqlInt(i2)) when (i1 != i2) ->
          [b; s]
      | (CEqlAgent(a1), CEqlAgent(a2)) when (a1 != a2) ->
          [b; s]
      | (CRange(bl, bu), CRange(sl, su)) when (bl < sl && bu > su) ->
          [CRange(bl,sl); CRange(su, bu)]
      | (CRange(bl, bu), CRange(sl, su)) when (bl < sl && bu < su) ->
          [CRange(bl, min sl bu)]
      | (CRange(bl, bu), CRange(sl, su)) when (bl < sl && bu > su) ->
          [CRange(max bl su, bu)]
      | (CRange(bl, bu), CRange(sl, su)) when (bu > sl || bl > su) ->
          [b]
      | (CGT(bi), CLT(si)) -> [CGT(max bi si)]
      | (CLT(bi), CGT(si)) -> [CLT(min bi si)]
      | (CGT(bi), CGT(si)) when (bi < si) -> [CRange(bi, si)]
      | (CLT(bi), CLT(si)) when (bi < si) -> [CRange(si, bi)]
      | _ -> []
  in
    List.map range_to_int (cmi (int_to_range b) (int_to_range s))
;;
```

```
(* Why isn't this in the list module?
 *
 * This guy iterates through all the elements of a list like map,
 *  but the function emits a list which are all appended together.
 *)
let list_acc mapper list =
  let rec acc list ret =
    match list with
        [] -> ret
      | hd :: tl -> acc tl ((mapper hd) @ ret)
  in
    acc list []
;;

(* return the first n elements of list *)
let rec list_first n list =
  match list with
      [] -> []
    | hd :: tl when n > 0 -> hd :: list_first (n - 1) tl
    | _ -> []
;;

(* Return a list with 'number' items duplicated *)
let rec list_fill item number =
  if number <= 0
  then []
  else item :: (list_fill item (number - 1));;


(* Pad the shorter or the two lists to make them the
 * same size.  Pad with 'Anys'
 *)
let cnst_extend a b =
  let delta = (List.length a) - (List.length b) in
    if delta > 0
    then (a, List.append b (list_fill Any delta))
    else if delta < 0
    then (List.append a (list_fill Any (delta * -1)), b)
    else (a,b)
;;

(* Pad a constraint list to ensure it is length l *)
let cnst_extend_to a l =
  let delta = l - (List.length a) in
    if delta > 0
    then List.append a (list_fill Any delta)
    else a
;;

(* Extend aC and bC to be the same length and AND them *)
```

```
let cnstAndAll aC bC =
  let (aC, bC) = cnst_extend aC bC in
    List.map2 cAnd aC bC
;;

(* Does signature match the term 'name' with inputs 'vars'? *)
let match_signature signature name vars =
  let match_params param vars =
    let anded = cnstAndAll param vars in
      List.for_all (fun a -> a != FalseSol) anded
  in
    (String.compare signature.name name == 0) &&
      ((List.length signature.params) == (List.length vars)) &&
      (match_params signature.params vars)
;;

(*  *)
let remove_fact_all db pred cnsts =
  (* print_string ("Removing: "^ (string_of_eval pred cnsts) ^ "\n"); *)
  List.filter
    (fun curr ->
       match curr with
           Fact(signature) when
             match_signature signature pred cnsts -> false
         | _ -> true)
    db
;;

(* Remove a single matching fact from a database, returing the new DB *)
let rec remove_fact1 db pred cnsts =
  (* print_string ("Removing: "^ (string_of_eval pred cnsts) ^ "\n"); *)
  match db with
      [] -> []
    | Fact(sign) :: tl when match_signature sign pred cnsts -> tl
    | hd :: tl -> hd :: (remove_fact1 tl pred cnsts)
;;

(* Return a new list of constraints specified by a parameter list *)
let cnst_of_params params env =
  let param_to_cnst = function
      Tst.Lit(i) -> CEqlInt     (i)
    | Tst.Sym(s) -> CEqlSymbol (s)
    | Tst.Var(i) -> List.nth env i
    | Tst.Str(s) -> CEqlStr     (s)
    | Tst.Anon   -> Any
    | Tst.Arr(a) -> failwith "Arrays are not support yet"
  in
    (*print_string (string_of_eval "cop_env" env);
    print_string ("cop: " ^ (Printer.string_of_params (Tst.Params(params))) ^ "\n");*)
    List.map param_to_cnst params
;;
```

```

(* Convert a param list to a list of constraints *)
let sig_to_cnst signature =
  let param_to_cnst = function
      Tst.Lit(i) -> CEqlInt    (i)
    | Tst.Sym(s) -> CEqlSymbol (s)
    | Tst.Var(i) -> Any
    | Tst.Anon   -> Any
    | Tst.Str(s) -> CEqlStr    (s)
    | Tst.Arr(a) -> failwith "Arrays are not supported yet"
  in
    List.map param_to_cnst signature
;;


(* Evaluate a query *)
let rec run_eval db addDB name vars =
  let rec run_gen tail nextGen =
    let sols = (nextGen ()) in
      match sols with
          NoSolution -> eval_loop tail
        | Solution (cnst, gen) ->
            Solution(
              (list_first (List.length vars) cnst),
              (fun unit -> run_gen tail gen))
   and eval_loop e =
     match e with
         [] -> NoSolution
       | Fact (signature) :: tail
           when match_signature signature name vars ->
           Solution (cnstAndAll vars signature.params,
                     (fun unit -> eval_loop tail))
       | Rule (signature, exec) :: tail
           when match_signature signature name vars ->
           let matchedVars = cnstAndAll vars signature.params in
             run_gen tail (fun unit -> exec db addDB matchedVars)
       | head :: tail -> eval_loop tail
  in
    (*print_string ("In: " ^ (string_of_eval name vars));*)
    eval_loop (!addDB @ !db)
;;

(* Replace the i'th element with e in list
 * Horribly wasteful, but oh well
 *)
let rec list_replace i e list =
  match list with
      [] -> []
    | hd :: tl ->
        if i == 0
        then e :: tl
```

```
        else hd :: (list_replace (i - 1) e tl)
;;


(* The function that should be run when I type [i..j] *)
let rec range i j = if i >= j then [] else i :: (range (i+1) j)


let parseDB (prog) =
  (*
   *   All of the parse functions take the information regarding
   *   their statement and return a function of the type
   *   database -> database -> cnsts -> next
   *
   *   Which correspond to
   *   primary db -> add-on db -> the scope's constraints
   *
   *   And they return a lazy solution iterator.
   *
   *   These same functions which are returned by the query method
   *    are used internally to compose everything.  Makes it
   *    (relatively) easy to implement new functionality since nobody
   *    needs to know anything about their parents or children except
   *    that they conform to this interface.
   *)



  (* Our only compiler directive is print, for now.
     learn/forget have a special syntax *)
  let parseCompilerDirective name params =
    let nc = String.compare name in
      (* Print something... probably just "Hello World" *)
      if (nc "print") == 0
      then
        fun db addDB cnst ->
          let print_param param =
            match param with
                Tst.Lit(i) -> print_int i
              | Tst.Str(s) -> print_string s
              | Tst.Sym(s) -> print_string s
              | Tst.Var(i) -> print_string (string_of_cnst (List.nth cnst i))
              | _ -> ()
          in
            (List.iter print_param params;
             print_string "\n";
             NoSolution)
      else
        (print_string "Unknown compiler directive";
         fun db addDB cnst ->
           NoSolution)
```

```
  in


(* Compute AND blocks by cANDing all the solutions in each row
 *  of the cross product of all the possible solutions
 *)
 let rec parseAndBlock stmts =
   match stmts with
       [] -> (fun db addDB cnst -> Solution (cnst, fun unit -> NoSolution))
     | stmt :: tail ->
         let nextStatement = (parseAndBlock tail) in
         let thisStatement = (parseStatement stmt) in
           fun db addDB cnst ->
             let nextGenMain = (nextStatement db addDB) in
             let rec runThisGens thisGen =
               match (thisGen ()) with
                   NoSolution -> NoSolution
                 | Solution (thisCnsts, thisGenNxt) ->
                     let rec runNextGens nextGen =
                       match (nextGen ()) with
                           NoSolution ->
                             runThisGens thisGenNxt
                         | Solution(nextCnsts, nextGenNxt) ->
                             Solution(nextCnsts, fun unit -> runNextGens nextGenNxt)
                     in
                       runNextGens (fun unit -> nextGenMain thisCnsts)
             in
               runThisGens (fun unit -> thisStatement db addDB cnst)


 (* Return all the solutions from one, then go to the next *)
 and parseOrBlock stmts =
   match stmts with
       [] -> (fun db addDB cnst -> NoSolution)
     | stmt :: tail ->
         let nextStmt = (parseOrBlock tail) in
         let currStmt = (parseStatement stmt) in
           fun db addDB cnst ->
             let rec runOr nxt =
               match nxt with
                   NoSolution -> nextStmt db addDB cnst
                 | Solution(vars, nxt) -> Solution(vars,
                                                   (fun unit -> runOr (nxt ())))
             in
               runOr (currStmt db addDB cnst)



 (* Return the results from a query *)
 and parseEval name params =
   let param_var_index var_idx =
```

```
      let rec pvi_iter plist idx =
        match plist with
            [] -> -1
          | Tst.Var(i) :: tl when i == var_idx -> idx
          | _ :: tl -> pvi_iter tl (idx + 1)
      in
        pvi_iter params 0
    in
    fun db addDB cnst ->
       let cnsts = cnst_of_params params cnst in
         (* Map the slots returned from the eval into our slot-space *)
       let revMap rCnsts =
         List.map2
           (fun cnst idx ->
              let pIdx = param_var_index idx in
                if pIdx == -1
                then cnst
                else cAnd cnst (List.nth rCnsts pIdx))
           cnst
           (range 0 (List.length cnst))
       in
         (* Run the eval, then send back the results, reverse mapping the slots as we go *
       let nxt = run_eval db addDB name cnsts in
       let rec doNxt nxt =
         match nxt with
             NoSolution -> NoSolution
           | Solution(rCnsts, nxt) ->
               (* print_string (string_of_eval name rCnsts); *)
               let rCnsts = revMap (list_first (List.length params) rCnsts) in
                 (* print_string (string_of_eval name rCnsts); *)
                 Solution(rCnsts, (fun unit -> doNxt (nxt ())))
       in
         doNxt nxt


(* ********    BEHOLD ----- The bane of my existence!!!!!!     *)

(* A dumber man could not have written this function...
 *   ... A smarter man would have known not to.
 *)
 and parseNotEval name params =
    let eval = parseEval name params in
      fun db addDB cnsts ->
        (* It probably will help to think of this function as a binary blob...
         *     I blacked out while I was writing it, but I remember it having
         *     something to do with lazily-generated cross products. ~John
         *)
        let rec iter_outs bigList =
          (* Printf.printf "%s\n" (string_of_eval "Level:" (List.hd bigList)); *)
          match bigList with
              [] -> failwith "Internal error 23"
```

```
          | myRow :: [] ->
              let rec linearGen myList =
                match myList with
                    [] -> NoSolution
                  | hd :: tl -> Solution([hd], fun unit -> linearGen tl)
              in
                linearGen myRow
          | myRow :: tl ->
              let tlGenMain = iter_outs tl in
              let rec twoGen myList nxtGen =
                match myList with
                    [] -> NoSolution
                  | myHd :: myTl ->
                      match nxtGen with
                          NoSolution ->
                            twoGen myTl tlGenMain
                        | Solution(sol, nxtGen) ->
                            Solution(myHd :: sol, fun unit -> twoGen myList (nxtGen()))
              in
                twoGen myRow tlGenMain
      in

      (* Iterate through all the solutions, subtracting all the new solutions
       * from the existing ones being stored in 'outs'
       *)
      let rec minus nxt outs =
        match nxt with
            NoSolution ->
              iter_outs outs
          | Solution(evCnsts, nxt) ->
              minus
                (nxt())
                (* (list_acc (fun out -> List.map2 cMinus out evCnsts) outs) *)
                (List.map2
                   (fun out evCnst ->
                      list_acc (fun o -> cMinus o evCnst) out)
                   outs
                   evCnsts)
      in
        (* Start with the input solution, and subtract all the results *)
        minus (eval db addDB cnsts) (List.map (fun c -> [c]) cnsts)


(* Run an eval in somebody else's database *)
and parseDot2 v pred params =
  let eval = parseEval pred params in
    (fun db addDB cnst ->
       match (List.nth cnst v) with
           CEqlAgent(adb) ->
             eval adb (ref []) cnst
         | a -> (Printf.printf
```

```
                    "Warning: attempted dot ('.') on a non-agent: %s\n"
                    (string_of_cnst a);
                  NoSolution))

and parseNDot2 v pred params =
  let eval = parseNotEval pred params in
    (fun db addDB cnst ->
       match (List.nth cnst v) with
           CEqlAgent(adb) ->
             eval adb (ref []) cnst
         | a -> (Printf.printf
                    "Warning: attempted dot ('.') on a non-agent: %s\n"
                    (string_of_cnst a);
                  NoSolution))


and doAnd myCnsts db addDB cnst =
  let sol = cnstAndAll myCnsts cnst in
    (*(print_string (string_of_eval "" myCnsts));
      (print_string (string_of_eval "" cnst));*)
    if List.for_all (fun a -> a != FalseSol) sol
    then Solution(sol, fun () -> NoSolution)
    else NoSolution

and parseCompOp op v e2 =
  let compOp i  =
    match op with
        Ast.Lt -> CLT(i)
      | Ast.Gt -> CGT(i)
      | Ast.Leq -> CLT(i + 1)
      | Ast.Geq -> CGT(i - 1)
      | Ast.Eq  -> CEqlInt(i)
      | _ -> failwith "Unsupported comparison operator"
  in
    match e2 with
        Tst.ELit(i) ->
          doAnd ((list_fill Any v) @ [(compOp i)])

and parseStrComp v s =
  doAnd ((list_fill Any v) @ [CEqlStr(s)])

and parseSymComp v s =
  doAnd ((list_fill Any v) @ [CEqlSymbol(s)])

and parseLearnForget name statements =
  let remove_facts db addDB cnsts =
    let remove_fact (name,params) =
      db := remove_fact_all !db name (cnst_of_params params cnsts)
    in
      List.iter remove_fact statements
  in
```

```
    let remove_fact1 db addDB cnsts =
      let remove_fact (name,params) =
        db := remove_fact1 !db name (cnst_of_params params cnsts)
      in
        List.iter remove_fact statements
    in
    let add_facts db addDB cnsts =
      let add_fact (name,params) =
        db := Fact({name = name; params = (cnst_of_params params cnsts)}) :: !db
      in
        List.iter add_fact statements
    in
    let nm = String.compare name in
      if (nm "learn") == 0
      then (fun db addDB cnsts -> add_facts db addDB cnsts; NoSolution)
      else if (nm "forget") == 0
      then (fun db addDB cnsts -> remove_facts db addDB cnsts; NoSolution)
      else if (nm "forget1") == 0
      then (fun db addDB cnsts -> remove_fact1 db addDB cnsts; NoSolution)
      else failwith ("Invalid directive: " ^ name)

  and parseDot1 v dname statements =
    let study = parseLearnForget dname statements in
      (fun db addDB cnst ->
         match (List.nth cnst v) with
             CEqlAgent(adb) ->
               study adb (ref []) cnst
           | a -> (Printf.printf
                     "Warning: attempted @ dot ('.') on a non-agent: %s\n"
                     (string_of_cnst a);
                   NoSolution))

  and parseStatement statement =
    match statement with
        Tst.Block (redOp, statements)
          when 0 == (String.compare redOp "AND") ->
            parseAndBlock statements
      | Tst.Block (redOp, statements)
          when 0 == (String.compare redOp "OR") ->
            parseOrBlock statements
      | Tst.Block (redOp, statements) ->
          (Printf.printf "Invalid reduction operator %s\n" redOp;
           (fun db addDB cnst -> NoSolution))
      | Tst.Eval (name,   params) ->
          parseEval name params
      | Tst.NEval (name,   params) ->
          parseNotEval name params
      | Tst.Directive (name, params) ->
          parseCompilerDirective name params
      | Tst.Comp(e1, compOp, e2) ->
          parseCompOp compOp e1 e2
```

```
      | Tst.DirectiveStudy(name, statements) ->
          parseLearnForget name statements
      | Tst.StrComp(v, s) ->
          parseStrComp v s
      | Tst.SymComp(v, s) ->
          parseSymComp v s
      | Tst.Dot1(v, dname, statements) ->
          parseDot1 v dname statements
      | Tst.Dot2(v, pred, params) ->
          parseDot2 v pred params
      | Tst.NDot2(v, pred, params) ->
          parseNDot2 v pred params
  in

  let parseRule stmt slots actions =
    fun db addDB inCnsts ->
      let rec runPer sols nxt =
        match nxt with
            NoSolution -> NoSolution
          | Solution(outCnsts, nxt) ->
              (* Have we already given this solution? *)
              if (List.mem outCnsts sols)
              then runPer sols (nxt())
              else
                (List.iter
                   (fun action ->
                      (ignore (action db addDB outCnsts)))
                   actions;
                 Solution(outCnsts, fun () -> runPer (outCnsts :: sols)(nxt())))
      in
        (* print_string ("Num slots: " ^ (string_of_int slots) ^ "\n"); *)
        runPer [] (stmt db addDB (cnst_extend_to inCnsts slots))
  in

  let parseRF = function
      Tst.Rule (name, parms, numVars, statement, nseStmt) ->
        Rule ({ name = name; params = (sig_to_cnst parms)},
              (parseRule (parseStatement statement) numVars
                 (List.map parseStatement nseStmt)))
    | Tst.Fact (name, parms)             ->
        Fact ({ name = name; params = (sig_to_cnst parms)})
  in

  let tProg = Trans.translate(prog) in
    ref (List.map parseRF tProg)
;;

(* Primary entry point into the database.  Specify the
 * term to query and the number of variables to pass in.
 *
 * db is the database to query, and addDB is the "add on"
```

```
 * database so the caller can pass information into the program
 *)
let query db addDB pred numVars =
  run_eval db addDB pred (list_fill Any numVars)
;;


(* Print all the rules and facts in a DB- for debugging *)
let rec dump_db db =
  let print_sig s =
    Printf.printf "%s(%s)"
      s.name
      (String.concat "," (List.map string_of_cnst s.params))
  in
  let dump_rf rf =
    match rf with
        Fact(s) ->
          print_sig s;
          print_string ";\n"
      | Rule(s, f) ->
          print_sig s;
          print_string " {}\n"
  in
    List.iter dump_rf db
;;
```

Listing 56: CμLOG Interpreter