Language Reference Manual for the Reinforcement Learning Language

Michael Groble (michael.groble@motorola.com)

October 17, 2006

1 Lexical conventions

The rll language has lexical elements for whitespace, identifiers, keywords, literals, operators and delimiters.

1.1 Whitespace

Similar to Python [1], there are some cases where whitespaces are significant in rll specifications. In particular, indentation is used to delineate blocks of statements. To handle this concept, the language uses the tokens `NEWLINE`, `INDENT` and `DEDENT`.

There is also a distinction between physical source lines and logical source lines. A backslash acts as a logical line continuation character. A physical line ending in a backslash will be joined with the next physical line to represent a single logical line. Logical line continuations also exist for content between the delimiter pairs `()`, `{}` and `[]`. The following code example is treated as a single logical line.

```
a = [1;
     2]
```

The `NEWLINE` token represents the end of a logical line. The example above therefore consists of the following tokens `a = [ 1 ; 2 ] NEWLINE`.

1.2 Comments

Comments start with a `#` character and continue to the end of the physical line.

1.3 Identifiers

Identifiers are sequences of letters (both upper and lower case), numbers and the underscore character `_`. The first character of an identifier must be a letter. Identifiers are case sensitive.

1.4 Keywords

The following are keywords in the rll language.

```
actions diagram else elsif environment episode for if in
mdp not random reward set states valid while
```

1.5 Literals

There are three types of literals in rll, integer numbers, floting point numbers and strings. Integers are sequences of digits. Floating point numbers are defined as in C. String literals are delimited by double quotes, for example "a string".

1.6 Operators

The following tokens are used as operators.

```
+ - * / = += -= *= /= < > == != <= >= && || !  '
```

1.7 Delimiters

The following tokens are used as delimiters.

```
( ) [ ] { } < > :  , ;
```

2 Types

integer an integer value

floating point a floating point value

string a string

array a fixed dimension array

set a set

function a function

state a named discrete scalar or single-dimension array

action a named discrete scalar or single-dimension array

reward a named floating point scalar

mdp a markov decision process

3 Expressions

3.1 Assignable Expressions

Assignable expressions are those expressions that may exist on the left hand side of an assignment (also known as "lvalues"). In rll, atomic assignable expressions consist of an identifier, optionally followed by the ' operator, optionally followed by an array index.

The ' operator is valid only for state types and is used to denote the updated value of the state.

An array index is an integer surrounded by square brackets. The following is an example using both.

```
s'[1] = s[1] + 2
```

Function calls may return multiple values. The individual return values are separated by commas and the entire group is enclosed in angle brackets, for example.

```
<s'[0],returns> = f()
```

3.2 Primary Expressions

Primary expressions consist of atomic assignable expressions, literals, parenthesized expressions, function calls, anonymous sets and anonymous arrays.

Function calls consist of an identifier followed by a possibly empty list of arguments surrounded by parentheses. The argument list can consist of both positional and named arguments, all separated by commas. Positional arguments may be any expression while a named argument is an identifier followed by = followed by an expression. For example.

```
sarsa(Racetrack, 0.6, episodes = 1000)
```

Anonymous sets are a comma delimited list of expressions surrounded by curly braces, for example.

```
{1,10,a}
```

Anonymous arrays consist of semicolon delimited rows surrounded by square brackets. Each row is a comma separated list of expressions. Each row must have the same number of elements.

```
[ a, 1; 0, b]
```

3.3 Unary Expressions

Unary expressions are primary expressions prefixed by one of the operators ! + - denoting logical negation, unary positive and unary negative respectively.

3.4 Multiplicative Expression

A multiplicative expression consists of a sole unary expression or two unary expressions separated by one of the binary operators `*` `/` denoting multiplication and division respectively.

3.5 Additive Expression

An additive expression consists of a sole multiplicative expression or two multiplicative expressions separated by one of the binary operators `+` `-` denoting addition and subtraction respectively.

3.6 Boolean Relation Expression

A boolean relation expression consists of a sole additive expression or two additive expressions separated by one of the binary operators `>` `<` `>=` `<=` or the keywords `in` or `not in`. These represent the logical relations greater than, less than, greater than or equal, less than or equal, in (set membership) and not in (set exclusion) respectively.

3.7 Boolean Equality Expression

A boolean equality expression consists of a sole boolean relation expression or two boolean relation expressions separated by one of the binary operators `==` `!=` representing equals to or not equals to respectively.

3.8 Boolean And Expression

A boolean and expression consists of a sole boolean equality expression or two boolean equality expressions separated by the binary operator `&&` representing logical and.

3.9 Boolean Or Expression

A boolean or expression consists of a sole boolean and expression or two boolean and expressions separated by the binary operator `||` representing logical or.

3.10 Expression

An expression consists of a sole boolean or expression or a ternary expression of the form

booleanOrExpression `if` *booleanOrExpression* `else` *booleanOrExpression*

The middle expression is the test condition. The first expression is the value of the expression if the test is true, the last expression is the value if the test is false.

4 Statements

At the top level, there are two types of statements in rll, one type for defining markov decision processes and one type for defining procedural statements.

4.1 MDP Statements

4.1.1 MDP Definition

An MDP definition consists of a line of the form

`mdp` *identifier* `(` *parameter list* `)`

followed by an indented block of statements which consist of state declaration, action declaration, reward declaration, diagram declaration, constraint declaration, episode declaration, environment declaration or random declaration.

Parameter list is a comma separated list of named parameters with default values, for example.

```
mdp Gamblers(maxCapital = 100, pSuccess = 0.5)
```

4.1.2 State Declaration

A state declaration consists of the keyword `states` followed by an identifier followed by an optional dimension (an integer surrounded by square brackets). An MDP definition may have multiple state declarations in which case the MDP consists of all states. For example, the following creates an MDP with a total of four states by naming two different two-dimensional vectors.

```
states position[2]
states velocity[2]
```

4.1.3 Action Declaration

An action declaration consists of the keyword `actions` followed by an identifier followed by an optional dimension. As with states, multiple action declarations can exist in an MDP definition.

4.1.4 Reward Declaration

A reward declaration consists of the keyword `reward` followed by an identifier. Only a single reward declaration can exist in an MDP definition.

4.1.5 Diagram Declaration

A diagram declaration is used to create a graphical representation of a 2 dimensional gridworld. A diagram declaration consists of the keyword diagram on a line followed by an indented block of string literals. For example

```
diagram
    " A     B "
    " ....... "
    "     .   "
    "     .   "
    "    S    "
    "     .   "
    "     .   "
    "    ...X "
```

The non-space characters in a diagram are interpreted as special identfiers. Taken in a set literal, the set {"."} consists of those 14 positions which are marked with that character. A diagram declaration is an optional element of an MDP definition.

4.1.6 Constraint Declaration

A constraint declaration is used to define the set of valid states and actions. A constraint declaration consists of the keyword `valid` followed by either `states` or `actions` and then an anonymous set expression. An MDP definition must have both a states constraint and an actions constraint.

4.1.7 Episode Declaration

An episode declaration is used to define episodic nature of the MDP. It consists of the keyword `episode` on a line followed by an indented block of statements consisting of set declarations or random declarations. An episode declaration is an optional element of an MDP definition.

4.1.8 Environment Declaration

An environment declaration is used to define the state and reward update functions of the MDP. It consists of the keyword `environment` on a line followed by an indented block of statements consisting of set declarations, random declarations and assignment statements. An MDP definition must have one environment declaration. The assignment statements in the environment declaration must assign values to the reward identifier and must assign values to the next state value for each of the state variables.

4.1.9 Set Declaration

A set declaration is of the form `set` *identifier* `=` *expression* `NEWLINE`.

4.1.10 Random Declaration

A random declaration declares a random variable with a particular distribution and is of the form `random` *identifier* `=` *expression* `NEWLINE`.

4.1.11 Assignment Statement

An assignment statement is of the form *assignableExpression* op *expression* NEWLINE where op may be any of the assignment operators = += -= *= /=.

4.2 Procedural Statements

The procedural statements include the assignment statement described above and add the following.

4.2.1 Function Call Statement

A function call statement is simply a function call expression followed by a NEWLINE.

4.2.2 For Statement

A for statement consists of a line for (*iteratorExpression*) NEWLINE followed by an indented block of procedural statements. An iterator expression is an expression of the form *identifier* = *expression* : *expression*, or of the form *identifier* in *set*. In the first case, the iterator takes on the values in the range from the first expression to the second. In the second case, the iterator takes on each of the values in the set. The indented block of statements is executed once for each iterator value.

4.2.3 While Statement

A while statement consists of the line while (*expression*) NEWLINE followed by an indented block of procedural statements. The block is executed while the expression evaluates to true.

4.2.4 If Statement

An if statement consists of three different types of chunks, an if chunk followed by an arbitrary number of elsif chunks followed by an optional else chunk. An if chunk is of the form if (*expression*) NEWLINE followed by an indented block of procedural statements. An elsif chunk is of the form elsif (*expression*) NEWLINE followed by an indented block of procedural statements. An else chunk is of the form else NEWLINE followed by an indented block of procedural statements.

5 Built-in Functions

There are a number of pre-defined functions available in both MPD definitions and procedural statements. Mathematical functions are available in both and consist of

`max(a,b)` maximum value of `a` and `b`

`min(a,b)` minimum value of `a` and `b`

`abs(a)` absolute value of `a`

The mathematical operators work on both scalar and array types. On arrays, the operations are performed element-wise.

Discrete random variable distributions are available in MPD definitions and consist of

`uniform(n)` uniform distribution from `0` to `n-1`

`binomial(p)` binomial distribution with probability of success `p`

`poisson(n)` poisson distribution with expectation `n`

References

[1] Guido van Rossum. The Python Programming Language. http://www.python.org/.

A ANTLR definition of parser and lexer

```
options {
        language="Cpp";
}
class RllParser extends Parser;
options {
    k=2;
    buildAST = true;
}
tokens {
        FILE;
        PARAMS;
        ARGS;
        MDP_BODY;
        SET;
        ARRAY;
        ROW;
        DIM;
        CALL;
        LHS_LIST;
        INDEX;
        DECL;
        UNARY_PLUS;
        UNARY_MINUS;
        PS_LIST;
```

```
        ITER;
        NARG;
}
fileInput
  : (NEWLINE | statement)* EOF!
    {#fileInput = #([FILE,"FILE"], fileInput); }
  ;
statement
  : mdpDefinition
  | proceduralStatement
  ;
proceduralStatementList
  : (proceduralStatement)+
    {#proceduralStatementList = #([PS_LIST,"PS_LIST"], proceduralStatementList);}
  ;

proceduralStatement
  : assignmentStatement
  | functionCallStatement
  | forStatement
  | whileStatement
  | ifStatement
  ;

mdpDefinition
  :  "mdp"^ ID OPEN_PAREN! (parameterList)? CLOSE_PAREN! NEWLINE!
     INDENT! mdpStatementList DEDENT!
  ;
parameterList
  : parameter (COMMA! parameter)*
    {#parameterList = #([PARAMS,"PARAMS"],parameterList);}
  ;

parameter
  : ID^ (ASSIGN^ expression)?
  ;
mdpStatementList
  : (mdpStatement)+
    {#mdpStatementList = #([MDP_BODY,"MDP_BODY"],mdpStatementList);}
  ;

mdpStatement
  : stateDeclaration
  | actionDeclaration
  | rewardDeclaration
  | diagramDeclaration
  | constraintDeclaration
  | episodeDeclaration
  | environmentDeclaration
  | randomDeclaration
```

```
  ;

stateDeclaration
  : "states"^ variableDeclaration NEWLINE!
  ;

actionDeclaration
  : "actions"^ variableDeclaration NEWLINE!
  ;

rewardDeclaration
  : "reward"^ ID NEWLINE!
  ;
constraintDeclaration
  : "valid"^ ("states" | "actions") set NEWLINE!
  ;
episodeDeclaration
  : "episode"^ NEWLINE!
    INDENT! (episodeStatement)+ DEDENT!
  ;
diagramDeclaration
  : "diagram"^ NEWLINE!
    INDENT! (STRING_LITERAL NEWLINE!)+ DEDENT!
  ;
environmentDeclaration
  : "environment"^ NEWLINE!
    INDENT! (environmentStatement)+ DEDENT!
  ;

episodeStatement
  : setDeclaration
  | randomDeclaration
  ;

setDeclaration
  : "set"^ ID ASSIGN! expression NEWLINE!
  ;
environmentStatement
  : setDeclaration
  | randomDeclaration
  | assignmentStatement
  ;
randomDeclaration
  : "random"^ variableDeclaration ASSIGN! expression NEWLINE!
  ;
forStatement
  : "for"^ OPEN_PAREN! iteratorExpression CLOSE_PAREN! NEWLINE!
    INDENT! proceduralStatementList DEDENT!
  ;
whileStatement
```

```
  : "while"^ OPEN_PAREN! expression CLOSE_PAREN! NEWLINE!
    INDENT! proceduralStatementList DEDENT!
  ;
ifStatement
  : "if"^ OPEN_PAREN! expression CLOSE_PAREN! NEWLINE!
    INDENT! proceduralStatementList DEDENT!
    ("elsif" OPEN_PAREN! expression CLOSE_PAREN! NEWLINE!
     INDENT! proceduralStatementList DEDENT!)*
    ("else"  NEWLINE!
     INDENT! proceduralStatementList DEDENT!)?
  ;

assignmentStatement
  : assignableExpression
    ( ASSIGN^
    | PLUS_ASSIGN^
    | MINUS_ASSIGN^
    | STAR_ASSIGN^
    | SLASH_ASSIGN^
    )
   expression NEWLINE!
  ;
expression
  : booleanOrExpression ("if"^ booleanOrExpression "else" booleanOrExpression)?
  ;

booleanOrExpression
  : booleanAndExpression (LOGICAL_OR^ booleanAndExpression)*
  ;
booleanAndExpression
  : booleanEqualityExpression (LOGICAL_AND^ booleanEqualityExpression)*
  ;
booleanEqualityExpression
  : booleanRelationExpression ((EQUAL^|NOT_EQUAL^) booleanRelationExpression)*
  ;
booleanRelationExpression
  : additiveExpression ((LESS_THAN^
                        |GREATER_THAN^
                        |LESS_OR_EQUAL^
                        |GREATER_OR_EQUAL^
                        |"in"^
                        |"not"^ "in") additiveExpression)*
  ;

additiveExpression
  : multiplicativeExpression ((PLUS^|MINUS^) multiplicativeExpression)*
  ;

multiplicativeExpression
  : unaryExpression ((STAR^|SLASH^) unaryExpression)*
```

```
  ;
unaryExpression
  : LOGICAL_NOT^ valueExpression
  | (PLUS^ {#PLUS->setType(UNARY_PLUS);}
       |MINUS^ {#MINUS->setType(UNARY_MINUS);}) valueExpression
  | valueExpression
  ;
valueExpression
  : atomicAssignableExpression
  | functionCall
  | set
  | array
  | INT
  | FLOAT
  | STRING_LITERAL
  | OPEN_PAREN^ expression CLOSE_PAREN!
  ;
assignableExpression
  : atomicAssignableExpression
  | assignableExpressionList
  ;
atomicAssignableExpression
  : atomicIndexableAssignableExpression
    (OPEN_BRACK! expression CLOSE_BRACK! {#atomicAssignableExpression = #([INDEX,"INDEX"], atomi
  ;
atomicIndexableAssignableExpression
  : ID (TICK^)?
  ;

assignableExpressionList
  : LESS_THAN! atomicAssignableExpression (COMMA! atomicAssignableExpression)* GREATER_THAN!
  {#assignableExpressionList = #([LHS_LIST,"LHS_LIST"], assignableExpressionList);}
  ;
iteratorExpression
  : ID ASSIGN! rangeExpression {#iteratorExpression = #([ITER,"ITER"], iteratorExpression);}
  | ID "in"! set {#iteratorExpression = #([ITER,"ITER"], iteratorExpression);}
  ;

rangeExpression
  : expression COLON^ expression
  ;
functionCallStatement
  : functionCall NEWLINE!
  ;

functionCall
  : ID OPEN_PAREN! (argumentList)? CLOSE_PAREN!
    {#functionCall = #([CALL,"CALL"],functionCall);}
  ;
```

```
argumentList
  : argument (COMMA! argument)*
  {#argumentList = #([ARGS,"ARGS"],argumentList);}
  ;
argument
  : positionalArgument
  | namedArgument
  ;

positionalArgument
  : expression
  ;

namedArgument
  : ID ASSIGN! expression {#namedArgument = #([NARG,"NARG"], #namedArgument);}
  ;
variableDeclaration
  : ID^ (variableDimension)?
  ;

variableDimension
  : OPEN_BRACK! INT CLOSE_BRACK!
    {#variableDimension = #([DIM,"DIM"],variableDimension);}
  ;

array
  : OPEN_BRACK! arrayRow (SEMI! arrayRow)* CLOSE_BRACK!
    {#array = #([ARRAY,"ARRAY"], array);}
  ;
set
  : OPEN_CURLY! expression (COMMA! expression)* CLOSE_CURLY!
    {#set = #([SET,"SET"], set);}
  ;
arrayRow
  : expression (COMMA! expression)*
    {#arrayRow = #([ROW,"ROW"], arrayRow);}
  ;

/*
  The following lexer was based on the Python lexer created by Terence Parr
  and Loring Craymer.  In particular the constructs used to correctly
  handle Python's way of using indentation to delimit compound statements

  I converted it to C++ and modified it to support the rll language.
  Mike Groble  Oct 1 2006
*/
/*
 [The "BSD licence"]
 Copyright (c) 2004 Terence Parr and Loring Craymer
 All rights reserved.
```

```
 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote products
    derived from this software without specific prior written permission.
 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
class RllLexer extends Lexer;
options {
    k=2;
    testLiterals=false; // have to test in identifier rule
}
{
/** Handles context-sensitive lexing of implicit line joining such as
 *  the case where newline is ignored in cases like this:
 *  a = [3,
 *       4]
 */
  private:
  int implicitLineJoiningLevel;

  public:
  void init() {
         implicitLineJoiningLevel = 0;
  }
}
OPEN_PAREN:  '(' {implicitLineJoiningLevel++;};
CLOSE_PAREN: ')' {implicitLineJoiningLevel--;};
OPEN_BRACK:  '[' {implicitLineJoiningLevel++;};
CLOSE_BRACK: ']' {implicitLineJoiningLevel--;};
OPEN_CURLY:  '{' {implicitLineJoiningLevel++;};
CLOSE_CURLY: '}' {implicitLineJoiningLevel--;};
COLON: ':';
COMMA: ',';
SEMI:  ';';
PLUS:   '+';
```

```
MINUS:  '-';
STAR:   '*';
SLASH : '/';
ASSIGN: '=';
PLUS_ASSIGN:  "+=";
MINUS_ASSIGN: "-=";
STAR_ASSIGN:  "*=";
SLASH_ASSIGN: "/=";
LESS_THAN:        '<';
GREATER_THAN:     '>';
EQUAL:            "==";
NOT_EQUAL:        "!=";
LESS_OR_EQUAL:    "<=";
GREATER_OR_EQUAL: ">=";
LOGICAL_AND: "&&";
LOGICAL_OR:  "||";
LOGICAL_NOT: '!';
TICK: '\'';
NUMBER
  : '.' Int (Exponent)? {$setType(FLOAT);}
  | Int ( '.' (Int)? (Exponent)? {$setType(FLOAT);}
        | Exponent {$setType(FLOAT);}
        | /*nothing*/ {$setType(INT);})
  ;
protected
Int
  : ('0'..'9')+
  ;

protected
Exponent
  : ('e' | 'E') ('+' | '-')? Int
  ;
ID
options {
        testLiterals = true;
}
  : ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
  ;
STRING_LITERAL
options {
        testLiterals = false;
}
  : '"'! (~('"'|'\n'|'\r'))* '"'!
  ;

/** Consume a newline and any whitespace at start of next line */
CONTINUED_LINE
  : '\\' ('\r')? '\n' (' '|'\t')* { newline(); $setType(antlr::Token::SKIP); }
  ;
```

```
/** Grab everything before a real symbol.  Then if newline, kill it
 *  as this is a blank line.  If whitespace followed by comment, kill it
 *  as it's a comment on a line by itself.
 *
 *  Ignore leading whitespace when nested in [..], (..), {..}.
 */
LEADING_WS
{
    int spaces = 0;
}
  : {getColumn()==1}?
     // match spaces or tabs, tracking indentation count
    ( ' '  { spaces++; }
    | '\t' { spaces += 8; spaces -= (spaces % 8); }
    | '\014' // formfeed is ok
    )+
    {
      if ( implicitLineJoiningLevel>0 ) {
          // ignore ws if nested
          $setType(antlr::Token::SKIP);
      }
      else {
          std::string s(spaces,' ');
          $setText(s);
      }
    }
    // kill trailing newline or comment
    ( {implicitLineJoiningLevel==0}? ('\r')? '\n' {newline();}
      {$setType(antlr::Token::SKIP);}
    |  // if comment, then only thing on a line; kill so we
       // ignore totally also wack any following newlines as
       // they cannot be terminating a statement
      '#' (~'\n')* ('\n' {newline();})+
      {$setType(antlr::Token::SKIP);}
    )?
  ;
/** Comments not on line by themselves are turned into newlines because
    sometimes they are newlines like
    b = a # end of line comment
    or
    a = [1, # weird
         2]
    This rule is invoked directly by nextToken when the comment is in
    first column or when comment is on end of nonwhitespace line.
    The problem is that then we have lots of newlines heading to
    the parser.  To fix that, column==1 implies we should kill whole line.
    Consume any newlines following this comment as they are not statement
    terminators.  Don't let NEWLINE token handle them.
 */
COMMENT
```

```
{
    int startCol = getColumn();
}
  : '#' (~'\n')* // let NEWLINE handle \n unless column = 1 for '#'
    { $setType(antlr::Token::SKIP); }
    ( {startCol==1}? ('\n' {newline();})+ )?
  ;
/** Treat a sequence of blank lines as a single blank line.  If
 *  nested within a (..), {..}, or [..], then ignore newlines.
 *  If the first newline starts in column one, they are to be ignored.
 */
NEWLINE
{
    int startCol = getColumn();
}
  : (options{greedy=true;}:('\r')? '\n' {newline();})+
    {if ( startCol==1 || implicitLineJoiningLevel>0 )
         $setType(antlr::Token::SKIP);
    }
  ;
WS
  : (' '|'\t')+ {$setType(antlr::Token::SKIP);}
  ;
```