L Tutorial

Vu Phan

March 15, 2016

This L tutorial is based on the L specification by Evgenii Balai. The tutorial is easier-to-read but less detailed.

Contents

1 Syntax and Semantics

1.1 Symbols

L characters include letters (A to Z and a to z) as well as digits (0 to 9). L also uses these special characters:
$> < = ! \setminus + - * / \% . , | \{ \} ( )$

Each L symbol is a sequence of one or more of the above characters. Some such symbols are *brandon* (an identifier) and `!=` (the not-equal operator).

1.2 Basic Terms

To express the English clause "the sky is blue", you can write $is\_blue(sky)$ in L. In this case, *sky* is a ***basic term***.

A basic term can be a ***constant***, which in turn could be a non-negative integer (such as 0 and 23) or a constant name (a string starting with a lowercase letter). You can assign an integer to a constant name with a *constant declaration* (discussed later).

A basic term can also be a ***variable***, such as X or $UnknownNum$. A variable name shall start with an uppercase letter.

A basic term may be a ***typed variable***. (*Type declarations* will be explained fully later. Basically, a type is non-empty finite set.) Assume you declared *type student* $= \{ann, bob\}$. Then a typed variable could be *student* S, where *student* is the type name and S is the variable belonging to type *student* (meaning S can stand for either *ann* or *bob*).

Also, a basic term might be an ***arithmetic term***, such as $1 + 5 - 2$, $3 * X$ (variable X), or $2 + numPeople$ (assuming constant name *numPeople* was assigned some numeric value, such as 8). The arithmetic operators are + (addition), − (subtraction), $*$ (multiplication), / (floor division), and % (modulo).

Lastly, a basic term can be a ***functional term***. For instance, the English phrase "the color of my car" can be written in L as $color(my\_car)$, which is a functional term where *color* is the function name. Function names must begin with a lowercase letter. Another example is the functional term
$book(godel, title\ T, year(1930 + 2))$, where *godel* is a constant name, T is a variable of type *title*, and $year(1930 + 2)$ is a "smaller" functional term.

1.3 Constant Declarations

To make your L program more meaningful and modifiable, you can assign integers to some constant names using ***constant declarations***. Some examples follow.

```
/* this is a comment */
const speedOfSound = 340. /* a constant declaration to the left of this comment */
const speedOfSound2 = speedOfSound + 3. /* another constant declaration */
const speedOfSoundCurrentCondition = speedOfSound2.
```

1.4 Set Expressions

In L, you can declare types (discussed in detail soon). Basically, a type is a non-empty finite set. To declare a type, you assign a ***set expression*** to a type name (a string starting with a lowercase letter).

A simple set expression is $\{honda, toyota, ford\}$.

Now assume you have declared *const height* = 180., then the set expression $\{height..height + 10\}$ contains integers from 180 to 190.

Assume you have type $t1$ is $\{1..3\}$ and type $t2$ is $\{2, 4\}$. The following are set expressions:

- $t1$
  The type name $t1$ now stands for its previously assigned set $\{1, 2, 3\}$.
- $coordinate(X, Y)$ *where* X *in* $t1$, Y *in* $t2$ (given *coordinate* is a function name)
  The set expression above is equivalent to
  $\{coordinate(1, 2), coordinate(1, 4), coordinate(2, 2), coordinate(2, 4), coordinate(3, 2), coordinate(3, 4)\}$.
- $t1 + t2$ (the set-theoretic union of $t1$ and $t2$ is $\{1, 2, 3, 4\}$)
  Similarly, the intersection $t1 * \{3..5\}$ is $\{3\}$. You can also use set complementation as in this more complicated set expression:
  $(f(X)$ *where* X *in* $t1) \setminus (f(Y)$ *where* Y *in* $t2)$
  (which evaluates to the set $\{f(1), f(3)\}$.

1.5 Type Declarations

A ***type declaration*** assigns a set expression to a type name (a string starting with a lowercase letter). The L program below illustrates the point.

```
const numSeats = 50. /* first, a constant declaration */

type seatNum = {1..numSeats}.
/* The above type declaration assigns the set {1, 2, 3,..., 50} to the type name ‘‘seatNum’’ */

type vipSeatNum = {1..numSeats/10}.
/* 10% of the total seats are VIP (seats 1 to 5) */

type regularSeatNum = seatNum \ vipSeatNum.
/* the remaining seats (from 6 to 50) are regular seats */

type vipAttendee = attendee(X) where X in vipSeatNum.
/* VIP attendees include attendee(1) through attendee(5) */

type regularAttendee = attendee(X) where X in regularSeatNum.

type attendee = vipAttendee + regularAttendee.
/* all attendees */
```

1.6 Quantified Terms

If you declared *type person* = $\{holly, maddie, lana\}$., then you can make an assertion that applies to all members of type *person* by using the ***quantified term*** *every person*. You can also refer to one unspecified member of the type by using another quantified term: *some person* (which can be either *holly*, *maddie*, or *lana*).

1.7 Terms

Assume `type storeNum = {1..3}`.

A ***term*** is either a basic term (such as `47` and `storeNum X`) or a quantified term (such as `every storeNum` and `some storeNum`).

1.8 Atoms

In English, the assertion "Today is Monday" is either true or false. Similarly, L supports the ***atom*** `today(monday)`, whose value is Boolean.

The atom above is specifically a ***predicate atom***, where the predicate name is `today`. Another example predicate atom is `squareOf(num X, X*X)` (this atom means the first argument squared equals the second argument, provided type `num` contains only non-negative integers).

An atom can also be a ***built-in atom***. Some instances are `0 < 1`, `X >= 3`, `2 * 1 = 2`, and `5 != 7`.

1.9 Literals

A ***literal*** is an atom that it possibly preceded by the keyword `not`. For instance, if the literal `colorOf(myCar, silver)` is true then the literal `not colorOf(myCar, silver)` is false, and vice versa.

1.10 Sentences

An L ***sentence*** can be:

- a single literal, such as `not isBlue(ballon1)`
- multiple literals with `and`/`or` in between, such as:
  - `moonIsClose and tideIsHigh`
  - `not isGoodAt(tim, soccer) or enjoys(some student, badminton) and needsPracticeWith(student X, volleyball)`

1.11 Maybe Constructs

Sometimes, we reason by cases. For instance, a student Vi hesitates on whether to do a homework assignment. The cases here are: (1) to do the assignment and (2) not to do the assignment. Nothing bad happens if Vi chooses case (1), so this case is acceptable. But Vi chooses case (2), then her grades will decrease, and she might lose some scholarships, which is bad. So case (2) should be discarded. In summary, Vi's reasoning process splits into two branches at the decision point on whether to do the homework. Tracing down each branch, Vi finds that only the first branch is acceptable. Therefore, there is exactly one solution in this story: to do the assignment.

To simulate such a decision point in L, we can use a ***maybe construct***, such as `maybe doHomework`. This construct is a shorthand for the predicate sentence `doHomework or not doHomework`. Although the sentence seems to be a tautology and consequently redudant, it has a "side effect": spliting the current logic flow into two paths, where the first path contains `doHomework` and the second path contains `not doHomework`. Other parts of the L program will determine which path corresponds to a solution (possibly both paths or even no paths have solutions).

Another illustrative maybe literal is `maybe hasA(student S, laptop)`.

1.12 Cardinality Constraints

Assume you have three friends: Ann, Bob, and Cher. You want to invite some of them to your apartment for your birthday party. But your apartment is small and can only accomodate at most two guests. Of course, you want to invite at least one friend to avoid celebrating your birthday alone. In short, the number of friends invited should be between 1 and 2. You want to generate all possible combinations of invites satisfying that limit. You need an L ***cardinality constraint***. See the L program below.

```
type friend = {ann, bob, cher}. /* just a type declaration */

maybe invite(friend X).
/* maybe literal (excluding the dot): any friend is possibly invited */

1 <= |{invite(friend X)}| <= 2.
/* cardinality constraint (excluding the dot): limit on the number of invites */
```

Solving this L program produces six ***models*** (acceptable cases):

1. invite(bob) invite(cher)
2. invite(cher)
3. invite(bob)
4. invite(ann) invite(bob)
5. invite(ann) invite(cher)
6. invite(ann)

1.13 Rules

A ***rule*** can be an unconditional assertion such as `skyIsBlue.` (this rule is simply an atom followed by a dot).

A rule can also be conditional. An example is `hasScholarships(student S) and hasInternships(S) if isOutstanding(S) or isWayTooLucky(S).`, notice that the variable `S` is known to be from the type `student` from its first occurence, whereas reoccurences of the same variable should not be typed.

1.14 Program

Basically, an L ***program*** is a sequence of constant declarations, type declarations, and rules in a logical order:

1. A constant/type name can only be used after the corresponding constant/type declaration.
2. The scope of each variable is within one rule. Consider the following L program.

```
type person = {katie, mandie}.
type car = {nissan, toyota}.

canDrive(person X) if owns(X, some car).
/*
Two occurences of the same variable ‘‘X’’ of type ‘‘person’’.
This rule is a shorthand for the rules:
canDrive(katie) if owns(katie, some car).
canDrive(mandie) if owns(mandie, some car).
*/

isCostly(car X) if hasPriceOver(X, 15000).
/*
The variable ‘‘X’’ in this rule is different from ‘‘X’’ in the first rule.
Now, ‘‘X’’ is in type ‘‘car’’.
*/
```

3. The first occurence of a variable in a rule must be be a typed variable (so that we know which type the variable belongs to). Latter occurences of the same variable in the rule should not be typed (to avoid redundancy and the possibility of mistakenly assigning a different type to the variable).

2 Example: L Program to Rank Students

2.1 How to Write from Sratch

Let us write the complete L program `ranking.l` from sratch. Assume your are a teacher of a class. You want to rank your students according to their performance and according to your evaluation policy.

The first step is to declare an L type whose each member is a student.

```
type student = {ann, bree, cher, dale, flo, gray}.
```

Next, you specify that there will be two quizzes: $q(1)$ and $q(2)$.

```
const numQuizzes = 2.
type quiz_num = {1..numQuizzes}.
type quiz = q(Num) where Num in quiz_num.
```

Now, you say there will be more exams than quizzes (the exams are numberred $e(1), e(2), \ldots$).

```
const numExams = numQuizzes + 1.
type exam_num = {1..numExams}.
type exam = e(Num) where Num in exam_num.
```

Then an "assessment" is either a quiz or and exam.

```
type assessment = quiz + exam.
```

You use a common grading scale.

```
type quiz_grade = {pass, fail}.
type exam_grade = {a, b, c, d, f}.
type grade = quiz_grade + exam_grade.
type acceptable_grade = grade \ {fail, f}.
```

Below are the grades of your students on assessments.

```
assessment_score(ann, q(1), pass). /* Ann scored Pass on quiz 1 */
assessment_score(ann, q(2), pass).
assessment_score(ann, e(1), a). /* Ann had an A on exam 1 */
assessment_score(ann, e(2), a).
assessment_score(ann, e(3), a).

assessment_score(bree, q(1), pass).
assessment_score(bree, q(2), pass).
assessment_score(bree, e(1), a).
assessment_score(bree, e(2), a).
assessment_score(bree, e(3), b).

assessment_score(cher, q(1), pass).
assessment_score(cher, q(2), pass).
assessment_score(cher, e(1), a).
assessment_score(cher, e(2), b).
assessment_score(cher, e(3), b).

assessment_score(dale, q(1), pass).
assessment_score(dale, q(2), pass).
assessment_score(dale, e(1), b).
assessment_score(dale, e(2), b).
assessment_score(dale, e(3), b).

assessment_score(flo, q(1), pass).
assessment_score(flo, q(2), fail).
assessment_score(flo, e(1), d).
assessment_score(flo, e(2), d).
assessment_score(flo, e(3), f).

assessment_score(gray, q(1), fail).
assessment_score(gray, e(1), f).
/* Gray skipped some classes! */
```

An assessment (either a quiz or an exam) of a student is acceptable if its grade is either Pass or A/B/C/D.

```
acceptable_assessment(student S, assessment A) if
    assessment_score(S, A, some acceptable_grade).
```

Standard ranks:
```
type rank = {outstanding, above_average, average, below_average, inferior}.
```

Basic order among ranks:

```
higher_rank(outstanding, above_average).
higher_rank(above_average, average).
higher_rank(average, below_average).
higher_rank(below_average, inferior).
```

To introduce transitivity, we need the following two rules:

```
higher_rank_in_between(rank R1, rank R2, rank R3) if
    higher_rank(R1, R2) and higher_rank(R2, R3).
/* for instance, this atom is true: higher_rank_in_between(outstanding, above_average, average) */

higher_rank(rank R1, rank R3) if
    higher_rank_in_between(R1, some rank, R3).
/*
For instance, we can now deduce higher_rank(outstanding, average),
because we already know higher_rank_in_between(outstanding, above_average, average).

Using higher_rank(outstanding, average) with the first of these two rules,
we can derive higher_rank_in_between(outstanding, average, below_average).

Then from the latter rule, higher_rank_in_between(outstanding, average, below_average) implies
higher_rank(outstanding, below_average).

Continuing in this manner, we can deduce all true atoms of the form higher_rank(X, Y) .
*/
```

Now, your evaluation policy specifies the criterion for each rank. (Note that a student can meet the criteria of several ranks.)

```
student_meets_criterion(student S, outstanding) if
    acceptable_assessment(S, every assessment) and
    assessment_score(S, every exam, a).


student_meets_criterion(student S, above_average) if
    acceptable_assessment(S, every assessment) and
    student_scores_a_on_two_distinct_exams(S, some exam, some exam).

student_scores_a_on_two_distinct_exams(student S, exam E1, exam E2) if
    assessment_score(S, E1, a) and assessment_score(S, E2, a) and
    E1 != E2.


student_meets_criterion(student S, average) if
    acceptable_assessment(S, every assessment) and
    assessment_score(S, some exam, a).


student_meets_criterion(student S, below_average) if
    acceptable_assessment(S, every assessment).


/* default rank */
student_meets_criterion(every student, inferior).
```

Your evaluation policy continues: a student is **not** in a rank if:

1. she doesn't meet the criterion for this rank (of course), or
2. she already meets the criterion for a higher rank (better for her that way)

```
not student_rank(student S, rank R2) if
    not student_meets_criterion(S, R2) or
    student_meets_criterion_for_higher_rank(S, some rank, R2).

student_meets_criterion_for_higher_rank(student S, rank R1, rank R2) if
    student_meets_criterion(S, R1) and higher_rank(R1, R2).
```

Because you cannot please all people at once:

```
student_dropped(gray).

student_remaining(student S) if not student_dropped(S).
```

Here is the hard part. You solve the ranking problem with two steps:

1. First generating many potential solutions
2. Then among them, selecting only reasonable solutions

```
maybe student_rank(student S, rank R) if student_remaining(S).
/* The rule above means each remaining student might be placed in any rank. */

1 <= |{student_rank(student S, rank R)}| <= 1 if student_remaining(S).
/*
This rule states that each remaining student must be placed in exactly one rank.

Q: Then how do we know which rank she will have?
A: Our previous rules guarantee that she will have the best rank whose criterion is met by her grades.
*/
```

After clicking `Compute Models` in the `L Editor` Eclipe plugin, you will see that our L program has exactly one model. The model contains the expected atoms:

1. $student\_rank(ann, outstanding)$
2. $student\_rank(bree, above\_average)$
3. $student\_rank(cher, average)$
4. $student\_rank(dale, below\_average)$
5. $student\_rank(flo, inferior)$

(Gray does not have a rank because he dropped.)

2.2 Source Code

```
/*
    AN ILLUSTRATIVE L PROGRAM
    Written 2015/12/22, updated 2016/03/15
    Vu Phan

    This L program outputs ranks of students in an imaginary course
*/

type student = {ann, bree, cher, dale, flo, gray}.
    /* Comment: Define a set by listing its members */

const numQuizzes = 2. /* Define a constant */
type quiz_num = {1..numQuizzes}. /* Range of integers */
type quiz = q(Num) where Num in quiz_num. /* Set-builder notation */

const numExams = numQuizzes + 1.
type exam_num = {1..numExams}.
type exam = e(Num) where Num in exam_num.

type assessment = quiz + exam. /* Union of sets */

type quiz_grade = {pass, fail}.
type exam_grade = {a, b, c, d, f}.
type grade = quiz_grade + exam_grade.
type acceptable_grade = grade \ {fail, f}. /* Difference of sets */


assessment_score(ann, q(1), pass). /* Student "ann" scores "pass" on assessment "q(1)" */
assessment_score(ann, q(2), pass).
assessment_score(ann, e(1), a).
assessment_score(ann, e(2), a).
assessment_score(ann, e(3), a).

assessment_score(bree, q(1), pass).
assessment_score(bree, q(2), pass).
assessment_score(bree, e(1), a).
assessment_score(bree, e(2), a).
assessment_score(bree, e(3), b).

assessment_score(cher, q(1), pass).
assessment_score(cher, q(2), pass).
assessment_score(cher, e(1), a).
assessment_score(cher, e(2), b).
assessment_score(cher, e(3), b).

assessment_score(dale, q(1), pass).
assessment_score(dale, q(2), pass).
assessment_score(dale, e(1), b).
assessment_score(dale, e(2), b).
assessment_score(dale, e(3), b).

assessment_score(flo, q(1), pass).
assessment_score(flo, q(2), fail).
assessment_score(flo, e(1), d).
assessment_score(flo, e(2), d).
assessment_score(flo, e(3), f).

assessment_score(gray, q(1), fail).
assessment_score(gray, e(1), f).
```

```
/* Assessment A of student S is acceptable if its grade is
    either: pass (for a quiz), or a/b/c/d (for an exam) */
acceptable_assessment(student S, assessment A) if
    assessment_score(S, A, some acceptable_grade).



type rank = {outstanding, above_average, average, below_average, inferior}.

higher_rank(outstanding, above_average).
higher_rank(above_average, average).
higher_rank(average, below_average).
higher_rank(below_average, inferior).


/* the two rules below introduce transitivity */

higher_rank_in_between(rank R1, rank R2, rank R3) if
    higher_rank(R1, R2) and higher_rank(R2, R3).

higher_rank(rank R1, rank R3) if
    higher_rank_in_between(R1, some rank, R3).


/* Note below: If a student meets the criterion for a rank,
                then she also meets the criterion for each lower rank */


student_meets_criterion(student S, outstanding) if
    acceptable_assessment(S, every assessment) and
    assessment_score(S, every exam, a).


student_meets_criterion(student S, above_average) if
    acceptable_assessment(S, every assessment) and
    student_scores_a_on_two_distinct_exams(S, some exam, some exam).

student_scores_a_on_two_distinct_exams(student S, exam E1, exam E2) if
    assessment_score(S, E1, a) and assessment_score(S, E2, a) and
    E1 != E2.


student_meets_criterion(student S, average) if
    acceptable_assessment(S, every assessment) and
    assessment_score(S, some exam, a).

student_meets_criterion(student S, below_average) if
    acceptable_assessment(S, every assessment).


/* Default rank's criterion is met by all students */
student_meets_criterion(every student, inferior).
```

```
/* The two rules below mean: a student isn't in a rank if either:
    she doesn't meet the criterion for that rank, or
    she already meets the criterion for a higher rank */

    not student_rank(student S, rank R2) if
        not student_meets_criterion(S, R2) or
        student_meets_criterion_for_higher_rank(S, some rank, R2).

    student_meets_criterion_for_higher_rank(student S, rank R1, rank R2) if
        student_meets_criterion(S, R1) and higher_rank(R1, R2).


student_dropped(gray).
student_remaining(student S) if not student_dropped(S).

/* For each rank, a remaining student is possibly in that rank */
maybe student_rank(student S, rank R) if student_remaining(S).


/* Each remaining student is in precisely one rank */
1 <= |{student_rank(student S, rank R)}| <= 1 if student_remaining(S).


/*
The program has exactly one model containing, among other atoms,
a rank for every remaining student:
    student_rank(dale,below_average)
    student_rank(cher,average)
    student_rank(ann,outstanding)
    student_rank(flo,inferior)
    student_rank(bree,above_average)
*/
```