Lambek Calculus Theorem Prover

Edward Loper

April 23, 2002

1 Introduction

Applicative Categorial Grammar provides a compositional description of the syntactic structure of natural language. It assigns a denotation to each expression in the language. Each expression's *denotation* consists of a lambda calculus term and a category. A denotation's *term* specifies the "meaning" of the expression. A denotation's *category* (or *type*) describes how it can be combined with other denotations.

The Lambek Calculus extends Categorial Grammar by extending the meanings of categories, thereby providing new ways to combine denotations. In Categorial Grammar, a denotation D can combine in the manner specified by a category C if D has category C. But Lambek calculus adds the converse assertion: a denotation D has a category C if D can combine in the mannar specified by C.

In Lambek Calculus, theorems consist of statements that one expression's denotation can be reduced to another expression's denotation. This paper describes a system which will find all proofs[1] for any theorem in Lambek Calculus.

1.1 Lambek Calculus

This section gives a brief overview of the Lambek Calculus. I used Carpenter(1997)'s description of Lambek Calculus as the basis for my theorem prover. I tried to stay consistant with Carpenter's terminology, where possible. See Carpenter(1997) for more details on the Lambek Calculus.

1.1.1 Terms

Lambek Calculus terms give the "meanings" of expressions, in terms of the functional composition of a fixed finite set of basic terms, **BasTerm**. The internal structure of the terms in **BasTerm** are left unanalyzed.

Terms are written using lower case Greek letters, such as α and β. Term constants are bold faced words, such as **eat** and **cat**. Terms can also be written out in lambda calculus, such as $\lambda x.\mathbf{eat}(x)(\alpha)$

[1]More precisely, all cut-free proofs. But since a theorem is provable if and only if it has a cut-free proof, the system is still guaranteed to find a proof if and only if a proof exists.

1.1.2 Categories

Lambek Calculus categories specify how a denotation can combine with other denotations. They consist of a fixed finite set of base categories, **BasCat**, and the closure of those categories under the operations $\backslash$, $/$, and $\cdot$. These operations specify how denotations can be combined (where linear order of the denotations combined is signifigant):

- A denotation with category A can be combined with a denotation of category $A\backslash B$ to produce a denotation of category B.
- A denotation with category B/A can be combined with a denotation of category A to produce a denotation of category B.
- A denotation with category A can be combined with a denotation of category B to produce a denotation of category $A \cdot B$.

Categories are written using upper case italic letters, such as A and B.

1.2 Denotations

A denotation is a pair consisting of a term and a category. It therefore specifies both the meaning of an expression, and how that expression can combine with other expressions.

Denotations are written as terms and categories joined with a colon, such as $\alpha : A$ and $\lambda x.\alpha : B/A$. The variable Δ can also stand for a denotation.

1.2.1 Expressions

Expressions represent string of natural language words. Expressions consist of a fixed finite set of base expressions, **BasExp**, and their closure under concatination. A lexicon, **Lex**, maps each expression in **BasExp** to its denotation.

Expressions are written in Roman text, such as "John likes Mary."

1.2.2 Sequents

A *sequent* is a theorem in Lambek Calculus which asserts that one sequence of denotations can be reduced to another sequence of denotations.

Sequents are written as a comma separated list of denotations, followed by the symbol $\Rightarrow$, followed by another comma separated list of denotations, such as:

$$\Gamma_1, \alpha : A, \lambda x.\mathbf{like}(x) : A\backslash B \Rightarrow \Gamma_1, \mathbf{like}(\alpha) : B$$

The variables Γ_1, Γ_2, ... are be used to represent zero or more denotations.

1.2.3 Proofs

A proof states that one sequent (the *conclusion*) is true iff zero or more sequents (the *assumptions*) are true. Each step of a proof is written as a line, with the assumptions above, the conclusion below, and the name of he rule used to the right. Each step of a proof must match one of the Lambek calculus rule schemes. An example proof step is:

$$\frac{a : \mathrm{b} \Rightarrow a : \mathrm{b} \quad b(a) : \mathrm{a} \Rightarrow b(a) : \mathrm{a}}{b : \mathrm{a/b}, c : \mathrm{b} \Rightarrow b(a) : \mathrm{a}}/\mathrm{L}$$

1.2.4 Rule Schemes

The Lambek calculus uses the following rule schemes. See Carpenter(1997) for a detailed description of these schemes.

$$\frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \alpha : A/B, \Delta, \Gamma_2 \Rightarrow \gamma : C}/L \qquad \frac{\Delta \Rightarrow \beta : B \quad \Gamma_1, \alpha(\beta) : A, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \Delta, \alpha : B\backslash A, \Gamma_2 \Rightarrow \gamma : C}\backslash L$$

$$\frac{\Gamma, x : A \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : B/A}/R \qquad \frac{x : A, \Gamma \Rightarrow \alpha : B}{\Gamma \Rightarrow \lambda x.\alpha : A\backslash B}\backslash R$$

$$\frac{\Gamma_1, \pi_1(\alpha) : A, \pi_2(\alpha) : B, \Gamma_2 \Rightarrow \gamma : C}{\Gamma_1, \alpha : A \cdot B, \Gamma_2 \Rightarrow \gamma : C} \cdot L \qquad \frac{\Gamma_1, \Rightarrow \alpha : A \quad \Gamma_2 \Rightarrow \beta : B}{\Gamma_1, \Gamma_2 \Rightarrow < \alpha, \beta >: A \cdot B} \cdot R$$

1.2.5 Summary of Terminology

The following table sumarizes the terminology:

Term	= **BasTerm** | $\lambda x.$*Term* | *Term(Term)*
BasTerm	= **eat** | **sleep** | ...
Cat	= **BasCat** | *(Cat \ Cat)* | *(Cat / Cat)* | *(Cat · Cat)*
BasCat	= np | n | ...
Exp	= **BasExp** | *Exp Exp*
BasExp	= eat | sleep | ...
Denotation	= *Term: Cat*
Sequent	= *Denotation** ⇒ *Denotation**

2 Usage

I designed and implemented a theorem prover for Lambek calculus. This program interactively asks for sequents, and attempts to prove them. It will find all proofs for any derivable sequent, and will report that there are no proofs for any non-derivable sequent. In addition, it attempts to unify free variables in the sequent's terms. For example, given the sequent:

$$\mathbf{John} : np, \mathbf{sings} : np\backslash s \Rightarrow \alpha : s$$

The program will derive that:

$$\alpha = \mathbf{sings}(\mathbf{John})$$

Figure 1 shows an example session with the theorem prover. Note that the theorem prover can generate LaTeX output, and can operate in either normal mode, or short-circuit mode, where it returns the first proof it finds for each sequent.

```
[edloper@syse ling554]$ ./lambek.py lexicon.txt

>> [np/n] [n] => [np]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Proof(s) for a: np/n, b: n => c: np

------------ I   -------------------- I
c: n => c: n     a(c): np => a(c): np
------------------------------------- /L
      a: np/n, b: n => a(c): np

>> dog sleeps => [(np/n)\s]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Proof(s) for dog: n, sleeps: np\s => a: (np/n)\s

                     ------------------------ I   -------------------------------------- I
                     a(dog): np => a(dog): np     sleeps(a(dog)): s => sleeps(a(dog)): s
---------------- I   ------------------------------------------------------------------- \L
dog: n => dog: n                a(dog): np, sleeps: np\s => sleeps(a(dog)): s
---------------------------------------------------------------------------------------- /L
                   a: np/n, dog: n, sleeps: np\s => sleeps(a(dog)): s
---------------------------------------------------------------------------------------- \R
                 dog: n, sleeps: np\s => (\x.sleeps(x(dog))): (np/n)\s

>> [np/n] [n] => [s]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Can't prove a: np/n, b: n => c: s

>> help

% Lambek Calculus Theorem Proover
%
% Type a sequent you would like prooved.  Examples are:
%   [np/n] [n] => [np]
...
%   the city tom likes => [np*(s/np)]
%
% Other commands:
%   help         -- show this information
%   latexmode    -- toggle latexmode (outputs in LaTeX)
%   shortcircuit -- toggle shortcircuit mode (return just one proof)
%   lexicon      -- display the lexicon contents
%   quit         -- quit

>> lex

% Lexicon:
%  gives:          gives: (np\s/np)/np
%  mary:           mary: np
%  john:           john: np
%  the:            the: np/n
...
```

Figure 1: Example session with the Lambek calculus theorem prover

2.1 Example Proofs

This section lists a number of example proofs generated by the theorem prover. These proofs demonstrate some of the system's capabilities.

Proof for "α: np, β: np\s $\Rightarrow$ γ: s"

$$\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{\beta(\alpha){:}\mathrm{s}\Rightarrow\beta(\alpha){:}\mathrm{s}}\mathrm{I}}{\alpha : \mathrm{np}, \beta : \mathrm{np}\backslash\mathrm{s} \Rightarrow \beta(\alpha) : \mathrm{s}}\backslash\mathrm{L}$$

Proof for "α: n, β: np\s $\Rightarrow$ γ: (np/n)\s"

$$\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{n}\Rightarrow\alpha{:}\mathrm{n}}\mathrm{I} \quad \dfrac{\dfrac{}{\beta(\alpha){:}\mathrm{np}\Rightarrow\beta(\alpha){:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{\gamma(\beta(\alpha)){:}\mathrm{s}\Rightarrow\gamma(\beta(\alpha)){:}\mathrm{s}}\mathrm{I}}{\beta(\alpha){:}\mathrm{np},\gamma{:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\gamma(\beta(\alpha)){:}\mathrm{s}}\backslash\mathrm{L}}{\beta{:}\mathrm{np}/\mathrm{n},\alpha{:}\mathrm{n},\gamma{:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\gamma(\beta(\alpha)){:}\mathrm{s}}/\mathrm{L}}{\alpha : \mathrm{n}, \gamma : \mathrm{np}\backslash\mathrm{s} \Rightarrow (\lambda\beta.\gamma(\beta(\alpha))) : (\mathrm{np}/\mathrm{n})\backslash\mathrm{s}}\backslash\mathrm{R}$$

Proof for "dog: n, sleeps: np\s $\Rightarrow$ α: (np/n)\s"

$$\dfrac{\dfrac{\dfrac{}{\mathbf{dog}{:}\mathrm{n}\Rightarrow\mathbf{dog}{:}\mathrm{n}}\mathrm{I} \quad \dfrac{\dfrac{}{\alpha(\mathbf{dog}){:}\mathrm{np}\Rightarrow\alpha(\mathbf{dog}){:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{\mathbf{sleeps}(\alpha(\mathbf{dog})){:}\mathrm{s}\Rightarrow\mathbf{sleeps}(\alpha(\mathbf{dog})){:}\mathrm{s}}\mathrm{I}}{\alpha(\mathbf{dog}){:}\mathrm{np},\mathbf{sleeps}{:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{sleeps}(\alpha(\mathbf{dog})){:}\mathrm{s}}\backslash\mathrm{L}}{\alpha{:}\mathrm{np}/\mathrm{n},\mathbf{dog}{:}\mathrm{n},\mathbf{sleeps}{:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{sleeps}(\alpha(\mathbf{dog})){:}\mathrm{s}}/\mathrm{L}}{\mathbf{dog} : \mathrm{n}, \mathbf{sleeps} : \mathrm{np}\backslash\mathrm{s} \Rightarrow (\lambda\alpha.\mathbf{sleeps}(\alpha(\mathbf{dog}))) : (\mathrm{np}/\mathrm{n})\backslash\mathrm{s}}\backslash\mathrm{R}$$

Proof for "the: np/n, kid: n, runs: np\s $\Rightarrow$ α: s"

$$\dfrac{\dfrac{}{\mathbf{kid}{:}\mathrm{n}\Rightarrow\mathbf{kid}{:}\mathrm{n}}\mathrm{I} \quad \dfrac{\dfrac{}{\mathbf{the}(\mathbf{kid}){:}\mathrm{np}\Rightarrow\mathbf{the}(\mathbf{kid}){:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{\mathbf{runs}(\mathbf{the}(\mathbf{kid})){:}\mathrm{s}\Rightarrow\mathbf{runs}(\mathbf{the}(\mathbf{kid})){:}\mathrm{s}}\mathrm{I}}{\mathbf{the}(\mathbf{kid}){:}\mathrm{np},\mathbf{runs}{:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{runs}(\mathbf{the}(\mathbf{kid})){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{the} : \mathrm{np}/\mathrm{n}, \mathbf{kid} : \mathrm{n}, \mathbf{runs} : \mathrm{np}\backslash\mathrm{s} \Rightarrow \mathbf{runs}(\mathbf{the}(\mathbf{kid})) : \mathrm{s}}/\mathrm{L}$$

Proof for "john: np, believes: np\s/s, tom: np, likes: np\s/np $\Rightarrow$ α: s/np"

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I} \quad \dfrac{\dfrac{}{\mathbf{t}{:}\mathrm{np}\Rightarrow\mathbf{t}{:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}\mathrm{I}}{\mathbf{t}{:}\mathrm{np},\mathbf{l}(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{t}{:}\mathrm{np},\mathbf{l}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}/\mathrm{L} \quad \dfrac{\dfrac{}{\mathbf{j}{:}\mathrm{np}\Rightarrow\mathbf{j}{:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{(\mathbf{b}((\mathbf{l}(\alpha))(\mathbf{t})))(\mathbf{j}){:}\mathrm{s}\Rightarrow(\mathbf{b}((\mathbf{l}(\alpha))(\mathbf{t})))(\mathbf{j}){:}\mathrm{s}}\mathrm{I}}{\mathbf{j}{:}\mathrm{np},\mathbf{b}((\mathbf{l}(\alpha))(\mathbf{t})){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\mathbf{b}((\mathbf{l}(\alpha))(\mathbf{t})))(\mathbf{j}){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{j}{:}\mathrm{np},\mathbf{b}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{s},\mathbf{t}{:}\mathrm{np},\mathbf{l}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow(\mathbf{b}((\mathbf{l}(\alpha))(\mathbf{t})))(\mathbf{j}){:}\mathrm{s}}/\mathrm{L}}{\mathbf{john} : \mathrm{np}, \mathbf{believes} : \mathrm{np}\backslash\mathrm{s}/\mathrm{s}, \mathbf{tom} : \mathrm{np}, \mathbf{likes} : \mathrm{np}\backslash\mathrm{s}/\mathrm{np} \Rightarrow (\lambda\alpha.\,(\mathbf{believes}((\mathbf{likes}(\alpha))\,(\mathbf{tom})))\,(\mathbf{john})) : \mathrm{s}/\mathrm{np}}/\mathrm{R}$$

Proof for "john: np, likes: np\s/np, mary: np $\Rightarrow$ α: s"

$$\dfrac{\dfrac{}{\mathbf{mary}{:}\mathrm{np}\Rightarrow\mathbf{mary}{:}\mathrm{np}}\mathrm{I} \quad \dfrac{\dfrac{}{\mathbf{john}{:}\mathrm{np}\Rightarrow\mathbf{john}{:}\mathrm{np}}\mathrm{I} \quad \dfrac{}{(\mathbf{likes}(\mathbf{mary}))(\mathbf{john}){:}\mathrm{s}\Rightarrow(\mathbf{likes}(\mathbf{mary}))(\mathbf{john}){:}\mathrm{s}}\mathrm{I}}{\mathbf{john}{:}\mathrm{np},\mathbf{likes}(\mathbf{mary}){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\mathbf{likes}(\mathbf{mary}))(\mathbf{john}){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{john} : \mathrm{np}, \mathbf{likes} : \mathrm{np}\backslash\mathrm{s}/\mathrm{np}, \mathbf{mary} : \mathrm{np} \Rightarrow (\mathbf{likes}(\mathbf{mary}))\,(\mathbf{john}) : \mathrm{s}}/\mathrm{L}$$

Proof for "α: (np/n*n) $\Rightarrow$ β: np"

$$\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{n}\Rightarrow\alpha{:}\mathrm{n}}\mathrm{I} \quad \dfrac{}{\beta(\alpha){:}\mathrm{np}\Rightarrow\beta(\alpha){:}\mathrm{np}}\mathrm{I}}{\beta{:}\mathrm{np}/\mathrm{n},\alpha{:}\mathrm{n}\Rightarrow\beta(\alpha){:}\mathrm{np}}/\mathrm{L}}{\langle\beta,\alpha\rangle : (\mathrm{np}/\mathrm{n}{\cdot}\mathrm{n}) \Rightarrow \beta(\alpha) : \mathrm{np}}\cdot\mathrm{L}$$

Proof for “α: np\s/np ⇒ β: np\(s/np)”

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{}{\beta{:}\mathrm{np}\Rightarrow\beta{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{(\gamma(\alpha))(\beta){:}\mathrm{s}\Rightarrow(\delta(\beta))(\alpha){:}\mathrm{s}}\mathrm{I}}{\beta{:}\mathrm{np},\gamma(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\delta(\beta))(\alpha){:}\mathrm{s}}\backslash\mathrm{L}}{\beta{:}\mathrm{np},\gamma{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow(\delta(\beta))(\alpha){:}\mathrm{s}}/\mathrm{L}}{\beta{:}\mathrm{np},\gamma{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\delta(\beta){:}\mathrm{s}/\mathrm{np}}/\mathrm{R}}{\gamma:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\delta:\mathrm{np}\backslash(\mathrm{s}/\mathrm{np})}\backslash\mathrm{R}$$

Proof for “gives2: np\s/(np*np), tom: np, mary: np ⇒ α: np\s”

$$\dfrac{\dfrac{\dfrac{}{\mathbf{tom}{:}\mathrm{np}\Rightarrow\mathbf{tom}{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{\mathbf{mary}{:}\mathrm{np}\Rightarrow\mathbf{mary}{:}\mathrm{np}}\mathrm{I}}{\mathbf{tom}{:}\mathrm{np},\mathbf{mary}{:}\mathrm{np}\Rightarrow\langle\mathbf{tom},\mathbf{mary}\rangle{:}(\mathrm{np}{\cdot}\mathrm{np})}{\cdot}\mathrm{R}\quad\dfrac{}{\mathbf{gvs}_2(\langle\mathbf{tom},\mathbf{mary}\rangle){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{gvs}_2(\langle\mathbf{tom},\mathbf{mary}\rangle){:}\mathrm{np}\backslash\mathrm{s}}\mathrm{I}}{\mathbf{gvs}_2:\mathrm{np}\backslash\mathrm{s}/(\mathrm{np}{\cdot}\mathrm{np}),\mathbf{tom}:\mathrm{np},\mathbf{mary}:\mathrm{np}\Rightarrow\mathbf{gives2}(\langle\mathbf{tom},\mathbf{mary}\rangle):\mathrm{np}\backslash\mathrm{s}}/\mathrm{L}$$

Proof for “gives: (np\s/np)/np ⇒ α: np\s/(np*np)”

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{}{\beta{:}\mathrm{np}\Rightarrow\beta{:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{}{\gamma{:}\mathrm{np}\Rightarrow\gamma{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}\Rightarrow((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}}\mathrm{I}}{\gamma{:}\mathrm{np},(\mathbf{gives}(\alpha))(\beta){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}}\backslash\mathrm{L}}{\gamma{:}\mathrm{np},\mathbf{gives}(\alpha){:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\beta{:}\mathrm{np}\Rightarrow((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}}/\mathrm{L}}{\gamma{:}\mathrm{np},\mathbf{gives}{:}(\mathrm{np}\backslash\mathrm{s}/\mathrm{np})/\mathrm{np},\alpha{:}\mathrm{np},\beta{:}\mathrm{np}\Rightarrow((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}}/\mathrm{L}}{\gamma{:}\mathrm{np},\mathbf{gives}{:}(\mathrm{np}\backslash\mathrm{s}/\mathrm{np})/\mathrm{np},\beta{:}(\mathrm{np}{\cdot}\mathrm{np})\Rightarrow((\mathbf{gives}(\alpha))(\beta))(\gamma){:}\mathrm{s}}{\cdot}\mathrm{L}}{\mathbf{gives}{:}(\mathrm{np}\backslash\mathrm{s}/\mathrm{np})/\mathrm{np},\beta{:}(\mathrm{np}{\cdot}\mathrm{np})\Rightarrow(\mathbf{gives}(\alpha))(\beta){:}\mathrm{np}\backslash\mathrm{s}}\backslash\mathrm{R}}{\mathbf{gives}:(\mathrm{np}\backslash\mathrm{s}/\mathrm{np})/\mathrm{np}\Rightarrow\mathbf{gives}(\alpha):\mathrm{np}\backslash\mathrm{s}/(\mathrm{np}{\cdot}\mathrm{np})}/\mathrm{R}$$

Proof for “the: np/n, city: n, tom: np, likes: np\s/np ⇒ α: (np*s/np)”

$$\dfrac{\dfrac{}{\mathbf{c}{:}\mathrm{n}\Rightarrow\mathbf{c}{:}\mathrm{n}}\mathrm{I}\quad\dfrac{\dfrac{}{\mathbf{the}(\mathbf{c}){:}\mathrm{np}\Rightarrow\mathbf{the}(\mathbf{c}){:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{}{\mathbf{t}{:}\mathrm{np}\Rightarrow\mathbf{t}{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}\mathrm{I}}{\mathbf{t}{:}\mathrm{np},\mathbf{l}(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{t}{:}\mathrm{np},\mathbf{l}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow(\mathbf{l}(\alpha))(\mathbf{t}){:}\mathrm{s}}/\mathrm{L}}{\mathbf{t}{:}\mathrm{np},\mathbf{l}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow(\lambda x.(\mathbf{l}(\alpha))(\mathbf{t})){:}\mathrm{s}/\mathrm{np}}/\mathrm{R}}{\mathbf{the}(\mathbf{c}){:}\mathrm{np},\mathbf{t}{:}\mathrm{np},\mathbf{l}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\langle\mathbf{the}(\mathbf{c}),(\lambda x.(\mathbf{l}(\alpha))(\mathbf{t}))\rangle{:}(\mathrm{np}{\cdot}\mathrm{s}/\mathrm{np})}{\cdot}\mathrm{R}}{\mathbf{the}:\mathrm{np}/\mathrm{n},\mathbf{city}:\mathrm{n},\mathbf{tom}:\mathrm{np},\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\langle\mathbf{the}(\mathbf{city}),(\lambda\alpha.\,(\mathbf{likes}(\alpha))\,(\mathbf{tom}))\rangle:(\mathrm{np}{\cdot}\mathrm{s}/\mathrm{np})}/\mathrm{L}$$

Proofs for “likes: np\s/np ⇒ α: np\s/np”

$$\dfrac{}{\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}}\mathrm{I}$$

$$\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}}\mathrm{I}}{\mathbf{likes}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}}/\mathrm{L}}{\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}}/\mathrm{R}$$

$$\dfrac{\dfrac{\dfrac{}{\alpha{:}\mathrm{np}\Rightarrow\alpha{:}\mathrm{np}}\mathrm{I}\quad\dfrac{\dfrac{\dfrac{}{\beta{:}\mathrm{np}\Rightarrow\beta{:}\mathrm{np}}\mathrm{I}\quad\dfrac{}{(\mathbf{likes}(\alpha))(\beta){:}\mathrm{s}\Rightarrow(\mathbf{likes}(\alpha))(\beta){:}\mathrm{s}}\mathrm{I}}{\beta{:}\mathrm{np},\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow(\mathbf{likes}(\alpha))(\beta){:}\mathrm{s}}\backslash\mathrm{L}}{\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}\Rightarrow\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}}\backslash\mathrm{R}}{\mathbf{likes}{:}\mathrm{np}\backslash\mathrm{s}/\mathrm{np},\alpha{:}\mathrm{np}\Rightarrow\mathbf{likes}(\alpha){:}\mathrm{np}\backslash\mathrm{s}}/\mathrm{L}}{\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}\Rightarrow\mathbf{likes}:\mathrm{np}\backslash\mathrm{s}/\mathrm{np}}/\mathrm{R}$$

3 Design

This section gives a detailed description of the design of the Lambek calculus theorem prover. The theorem prover was written in Python. It consists of five basic data classes, a lexicon, a lambda calculus unifier, and a proof function.

3.1 Data Classes

The five basic data classes, and their subclasses, are:

- `Term`: A lambda calculus expression.
  - `Var`: A variable (e.g., x or α).
  - `Const`: A constant (e.g., **run**).
  - `Appl`: An application (e.g., $\alpha(\beta)$).
  - `Abstr`: An abstraction (e.g., $\lambda x.x$).
  - `Tuple`: A tuple (e.g., $\langle x, y\rangle$).
- `Type`: A category.
  - `LSlash`: A left-slash category (e.g., A\B).
  - `RSlash`: A right-slash category (e.g., A/B).
  - `Dot`: A product category (e.g., A·B).
  - `BaseType`: A base type (e.g., np).
- `TypedTerm`: A denotation (e.g., α:A).
- `Sequent`: A sequent (e.g., α:A $\Rightarrow$ α:A).
- `Proof`: A Lambek calculus proof.

3.1.1 `Term`

Lambda calculus expressions are represented by trees of objects of type `Term`. The constructors and accessors for each subtype of `Term` are:

Term	Constructor	Accessor$_1$	Accessor$_2$
$\alpha(\beta)$	`Appl`(α, β)	`Appl.func` = α	`Appl.arg` = β
$\lambda x.\alpha$	`Abstr`(x, α)	`Abstr.var` = x	`Abstr.arg` = α
$\langle\alpha, \beta\rangle$	`Tuple`(α, β)	`Tuple.left` = α	`Tuple.right` = β
name	`Const`(*name*)	`Const.name` = *name*	
x	`Var`()		

The following functions are also defined for lambda expressions:

- `freevars`(α): Return a list of all free variables in the given lambda expression.
- `boundvars`(α): Return a list of all bound variables in the given lambda expression.
- `vars`(α): Return a list of all variables used in the given lambda expression. vars(α) = `freevars`(α) + `boundvars`(α)
- `replace`(x, α, β): Replace all free occurances of the x in β with α.
- `reduce`(α): Return the reduced form of α (i.e., the form in which $\beta-$ and $\eta-$ reduction can not be applied to α or any of its subterms).
- `unify`(α, β, *varmap*): Attempt to bind the free variables of α and β such that $\alpha = \beta$, and return the resulting expression. *varmap* is a dictionary providing a partial mapping variables to values. The bindings in *varmap* must be obeyed by the unification. *varmap* is modified by this procedure. See Section 3.3 for more details.
- `parse_term`(s): Return the lambda expression corresponding to the string s. An example string for a lambda expression is "\?x.likes(x)".

3.1.2 `Type`

Categories are represented by trees of objects of type `Type`. The constructors and accessors for each subtype of `Type` are:

Term	Constructor	Accessor$_1$	Accessor$_2$
A \ B	`LSlash`(A, B)	`LSlash.arg` = A	`LSlash.result` = B
B / A	`RSlash`(B, A)	`LSlash.arg` = A	`RSlash.result` = B
A · B	`Dot`(A, B)	`Tuple.left` = A	`Tuple.right` = B
type	`BaseType`(*type*)	BaseType.name = *type*	

The following function is also defined for lambda expressions:

- `parse_type`(s): Return the category corresponding to the string s. An example string for a category is "np\(s/(np*np))". The order of operations are as defined in Carpenter(1997).

3.1.3 `TypedTerm`

Denotations are represented by objects of class `TypedTerm`. The following functions and accessors are defined for `TypedTerms`:

- `TypedTerm(`α, A`)`: Construct a `TypedTerm` representing the denotation a:A.
- `TypedTerm.type`: A `TypedTerm`'s type.
- `TypedTerm.term`: A `TypedTerm`'s term.
- `(`α`:A).unify(`β:B, *varmap*`)`: If A=B, return (`unify(`$\alpha, \beta,$ *varmap*`)`):A. Otherwise, return `None`. See Section 3.3 for more details about unification of lambda expressions.

3.1.4 `Sequent`

Sequents are represented by objects of cass `Sequent`. The following functions and accessors are defined for `Sequents`:

- `Sequent(`*left*, *right*: Construct a `Sequent` with the given left and right sides, where *left* and *right* are lists of `TypedTerms`.
- `Sequent.left`: A list containing the denotations on the left of the sequent.
- `Sequent.right`: A list containing the denotations on the right of the sequent.

3.1.5 `Proof`

Proofs are represented by trees of objects of cass `Proof`. The following functions and accessors are defined for `Proofs`:

- `Proof(`*rule, assumptions, conclusion, varmap*`)`: Construct a new sub-proof. *assumptions* is a list of `Proofs`, whose conclusions are the assumptions of the sub-proof. *conclusion* is a `Sequent` which gives the conclusion of the proof. *rule* is a string naming the rule used to produce this proof. And *varmap* is a dictionary mapping `Vars` to `Terms` which specifies values for (some of) the free variables of the terms in the proof.
- `Proof.rule`: A string naming the rule used to produce the proof.
- `Proof.assumptions`: A list of `Proofs`, whose conclusions are the assumptions of the sub-proof.
- `Proof.conclusion`: A `Sequent` giving the conclusion of the proof.
- `Proof.varmap`: A dictionary mapping `Vars` to `Terms` which specifies values for (some of) the free variables of the expressions in the proof.
- `Proof.simplify()`: Return a simplified version of this proof, where the proof's terms have had the values of their free variables filled in and have been reduced.
- `Proof.pp()`: Return a text representation of this proof tree (as seen in Figure 1).
- `Proof.to_latex()`: Return a latex representation of this proof tree (as seen in Section 2.1).

3.2 `Lexicon`

The `Lexicon` class maintains a mapping from words to `TypedTerm`s. It defines the following functions:

- `Lexicon`(): Construct a new (empty) lexicon.
- `Lexicon.load`(*file*): Add the lexicon entries in the given file to this lexicon. The file should have one lexicon entry per line, and lexicon entries should have the format "word:α:A". Any text to the right of a '#' character is considered a comment.
- `Lexicon`[*word*]: Return the `TypedTerm` corresponding to the given word. If none exists, return `None`.
- `Lexicon.parse`(*str*): Split the given string on whitespace, and return a list containing `Lexicon`[*word*] for each *word* in the string.
- `Lexicon.words`(): Return a list containing all words with values defined in the lexicon.

3.3 The Lambda Calculus Unifier

The lambda calculus unifier attempts to find a set of values for the free values of two terms that will make them equal. In addition, these values must be consistant with a given set of free-variable values.

The unifier is implemented by the function `unify`(α,β,*varmap*). This function attempts to unify α and β in a way consistant with the priors specified by *varmap*. It adds any new free-variable bindings to *varmap*, and returns a version of the term containing as few free variables as possible. As a special case, variables may be mapped by *varmap* to `None`, indicating that they cannot be unified with anything but themselves. This is used to ensure that bound variables do not get assigned values. If `unify` cannot unify α and β, it returns `None` and does not modify *varmap*. Note that the unifier is not currently guaranteed to find a unification of two terms if one exists. The unifier uses a case-by-case analysis to unify α and β, based on their classes. Figure 2 describes the different cases.

3.4 The Prover

The prover attempts to find all proofs for a given sequent. It works by matching the given sequent against the conclusion of each rule scheme. If a rule scheme matches the conclusion of a rule scheme, the prover recursively attempts to find proofs for each of the rule scheme's assumptions. This process is guaranteed to terminate because each sequent has only a finite number of cut-free proofs (Carpenter, 1997).

In addition to a sequent, the prover is given a set of free-variable values. These values are used to unify multiple occurances of the same term in the rule scheme (for example, in the /L rule, the term α appears both in the conclusion and in one of the assumptions). If the terms cannot be unified, then the proof fails. If the proof succeeds, then the set of free-variable values used is recorded in its `varmap` field.

The algorithms for matching sequents against the rule schemes are given in Figures 5, 6, 3, 4, 7, and 8.

```
If $\alpha \in$Var
    – If $varmap[\alpha]$ == None: fail
    – If $\alpha \in$ freevars($\beta$): fail
    – If $\alpha$ is in $varmap$, unify($varmap[\alpha]$, $\beta$
    – else: $varmap[\alpha] = \beta$
If $\alpha \in$Tuple, $\beta \in$Tuple
    – $\langle$unify($\pi_1(\alpha)$, $\pi_1(beta)$), unify($\pi_2(\alpha)$, $\pi_2(beta)$)$\rangle$
If $\alpha \in$Abstr, $\beta \in$Abstr
    – $\lambda x.$unify($\alpha$.body$[x/\alpha$.var$]$., $\beta$.body$[x/\alpha$.var$]$) Where $x$ is fresh.
If $\alpha \in$Appl, $\beta \in$Appl
    – unify($\alpha$.func, $beta$.func) (unify($\alpha$.var, $beta$.var))
    – If that fails, and $\alpha = x(y)$, where $y$ is free, then $varmap[x] = \lambda z.\alpha[z/y]$
If $\alpha \in$Const, $\beta \in$Const
    – if $\alpha = \beta$: return $\alpha$
    – else: fail
```

Figure 2: Pseudo-code for unify. This figure shows the case-by-case analysis used to unify two terms. Note that this algorithm is not guaranteed to find a unification if one exists.

```
• For each $\Gamma \Rightarrow \gamma$ : B/A that matches the given $seq$:
    – For each proof $p$ that $\Gamma, x$ : A $\Rightarrow \alpha_{assum}(x)$ : B ($x$ fresh)
        ∗ unify($\lambda x.\alpha, \lambda x.\alpha_{assum}$)
        ∗ return Proof("/R", [$p$], $seq$, $varmap$)
```

Figure 3: Pesudo-code for matching the /R rule.

```
• For each $\Gamma \Rightarrow \gamma$ : B\A that matches the given $seq$:
    – For each proof $p$ that $x$ : A$, \Gamma \Rightarrow \alpha_{assum}(x)$ : B ($x$ fresh)
        ∗ unify($\lambda x.\alpha, \lambda x.\alpha_{assum}$)
        ∗ return Proof("\R", [$p$], $seq$, $varmap$)
```

Figure 4: Pesudo-code for matching the \R rule.

```
• For each $\Gamma_1, \alpha : \mathrm{A/B}, \Delta, \Gamma_2 \Rightarrow \gamma : C$ that matches the given seq:
    – For each proof $p_{left}$ that $\Delta \Rightarrow \beta_{left}$ :B
        ∗ For each proof $p_{right}$ that $\Gamma_1, \alpha(\beta_{left}) : \mathrm{A}, \Gamma_2 \Rightarrow \gamma_{right} : C$
            · unify($\gamma$, $\gamma_{right}$)
            · return Proof("/L", [$p_{left}, p_{right}$], seq, varmap)
```

Figure 5: Pesudo-code for matching the /L rule.

```
• For each $\Gamma_1, \Delta, \alpha : \mathrm{A\backslash B}, \Gamma_2 \Rightarrow \gamma : C$ that matches the given seq:
    – For each proof $p_{left}$ that $\Delta \Rightarrow \beta_{left}$ :B
        ∗ For each proof $p_{right}$ that $\Gamma_1, \alpha(\beta_{left}) : \mathrm{A}, \Gamma_2 \Rightarrow \gamma_{right} : C$
            · unify($\gamma$, $\gamma_{right}$)
            · return Proof("\L", [$p_{left}, p_{right}$], seq, varmap)
```

Figure 6: Pesudo-code for matching the \L rule.

```
• For each $\Gamma_1, \alpha : \mathrm{A{\cdot}B}, \Gamma_2 \Rightarrow \gamma : C$ that matches the given seq:
    – For each proof $p$ that $\Gamma_1, \alpha_{left} : \mathrm{A}, \alpha_{right} : \mathrm{B}, \Gamma_2 \Rightarrow \gamma_{assum} : C$
        ∗ unify($\alpha$, $\langle \alpha_{left}, \alpha_{right} \rangle$))
        ∗ unify($\gamma, \gamma_{assum}$)
        ∗ return Proof("·L", [$p$], seq, varmap)
```

Figure 7: Pesudo-code for matching the ·L rule.

```
• For each $\Gamma_1, \Gamma_2 \Rightarrow \langle \alpha, \beta \rangle : \mathrm{A{\cdot}B}$ that matches the given seq:
    – For each proof $p_{left}$ that $\Gamma_1 \Rightarrow \alpha_{assum}$ :A
        ∗ For each proof $p_{right}$ that $\Gamma_1 \Rightarrow \beta_{assum}$ :B
            · unify($\langle \alpha, \beta \rangle$, $\langle \alpha_{assum}, \beta_{assum} \rangle$)
            · return Proof("·R", [$p_{left}, p_{right}$], seq, varmap)
```

Figure 8: Pesudo-code for matching the ·R rule.