

The Penn Lambda Calculator: Pedagogical Software for
Natural Language Semantics

Lucas Champollion, Joshua Tauberer and Maribel Romero

University of Pennsylvania

Proceedings of the GEAF 2007 Workshop

Tracy Holloway King and Emily M. Bender (Editors)

CSLI Studies in Computational Linguistics ONLINE

Ann Copestake (Series Editor)

2007

CSLI Publications

<http://csli-publications.stanford.edu/>

Abstract

This paper describes a novel pedagogical software program that can be seen as an online companion to one of the standard textbooks of formal natural language semantics, Heim and Kratzer (1998). The *Penn Lambda Calculator* is a multifunctional application designed for use in standard graduate and undergraduate introductions to formal semantics: Teachers can use the application to demonstrate complex semantic derivations in the classroom and modify them interactively, and students can use it to work on problem sets provided by the teacher. The program supports demonstrations and exercises in two main areas: (1) performing beta reduction in the simply typed lambda calculus; (2) application of the bottom-up algorithm for computing the compositional semantics of natural language syntax trees. The program is able to represent the full range of phenomena covered in the Heim and Kratzer textbook by function application, predicate modification, and lambda abstraction. This includes phenomena such as intersective adjectives, relative clauses and quantifier raising. In the student use case, emphasis has been placed on providing “live” feedback for incorrect answers. Heuristics are used to detect the most frequent student errors and to return specific, interactive suggestions.

1 Introduction

1.1 Background

For almost ten years now, the textbook by Heim and Kratzer (1998) (henceforth HK) has enjoyed a remarkable success as the textbook of choice for many introductory courses in natural language formal semantics. The semantic framework it presents can be seen as a standardization of Montague-style semantics (Montague, 1974) when applied to Generative Grammar syntax, with lexical items corresponding to simply typed lambda calculus expressions and with a very small number of composition rules. This framework has become a de facto standard in which much formal semantic work has been expressed over the last decade.

Teaching formal semantics can be a challenging classroom experience both to instructors and to students. Anyone who has ever taught a course on formal semantics will be familiar with the problem of drawing ever larger derivations, and changing them on the fly as the class goes on. A sentence of just ten words can easily fill an entire blackboard and take half an hour to draw (see Figure 4 below for an example). As for students, once they have left the classroom, they are often on their

[†]We would like to thank Patrick Blackburn, Chris Potts and the audiences of several demonstration sessions during the 2007 Stanford Linguistic Institute of the Linguistic Society of America (LSA) for helpful comments and suggestions. We are grateful to Yuval Masory for programming advice. We also thank the participants of the Spring and Summer 2007 introductory semantics classes at the University of Pennsylvania and at LSA respectively for their valuable feedback, as well as the participants of our usability tests. This work has been supported by a University of Pennsylvania SAS technology grant.

own with their homework exercises. In our experience, however, early feedback is crucial for student performance on lambda calculus and HK derivations.

For both these problems, the use of educational software suggested itself to us. Experiences with linguistic learning environments such as the *Trees* program (Kroch and Crist, 2002) in syntax courses at the University of Pennsylvania have been positive throughout (Anthony Kroch, p.c.) However, we were surprised to find that there does not seem to exist any educational software suited for our task. While some semantics- or logic-oriented educational programs exist (see Section 4), they are geared towards different (though related) applications, and not specific to the HK framework. Looking beyond natural language semantics, there appears to exist (apart from software geared towards students with a programming background) no training software for the more general field of the lambda calculus.

The present work describes our attempt at filling this much needed gap.

1.2 What It Does

The *Penn Lambda Calculator* is a multifunctional tool, designed for supporting both the instructor and the student in a variety of scenarios. The application is available as a “teacher edition” and a “student edition”. The main difference is that the student edition is limited so that it does not provide automatic answers to the exercises the student is working on.

The following functionality is available in both editions of the application:

- **Interactive exercise solving.** Typically, the instructor will prepare exercises ahead of time in the form of a file, though the application also contains a graphical interface (“Scratch Pad”) that allows users to input problem statements of their own. In each case, the program reads in the problem statement, internally generates a solution as applicable, displays the exercise and waits for student input. As the student progresses through the exercise, his or her answers are checked for correctness and the program gives appropriate feedback. The application currently supports the following kinds of exercises: type checking, reduction of lambda terms and bottom-up semantic derivations. Section 2 discusses them in further detail.

In addition, the teacher edition provides the following functions:

- **Visual presentation of semantic derivations.** This mode is intended to be used with a digital projector in class. The instructor provides the program with the tree and the lexical entries of the terminal nodes. The application then computes the denotations of nonterminal nodes automatically in a bottom-up fashion. At any point in time, the instructor can interrupt or rewind the derivation and/or modify any of the lexical entries involved. See section 3.1 for details.

- **Automatic scoring and grade management.** Students submit their completed work to their instructors electronically. Section 3.3 describes the tools the program provides to instructors to inspect and grade submitted work.

1.3 What We Aimed For

In the development of this program, we have adopted a few specific goals that go beyond best practices in software development.

- We view the textual feedback as a central component of the functionality of the application. Accordingly, we have made extended efforts to keep this feedback informative without constraining the range of admissible inputs more than absolutely necessary.
- The program has been designed so that it can be used by students with minimal outside instruction beyond the semantics that is needed to complete the exercises. Unlike related software (Barwise and Etchemendy, 1999; Larson et al., 1997), we do not presuppose that students read program documentation. We have performed extensive usability testing to ensure that the student interface is easy and intuitive to use for students at the introductory level of formal semantics with little background in computer usage.

1.4 How to Get It

The *Penn Lambda Calculator* is a stand-alone application available as a platform-independent Java Jar file, which is directly executable on Mac OS X and on most Unix systems. It is also available as a Microsoft Windows executable. All files are downloadable from the project's website. The student edition of the program is open source, licensed with the common GNU GPL license (Stallman, 2007), and the source code is linked from the website. In addition, the “engine” of the program, a fine-grained object-oriented model of simply typed lambda calculus expressions, is also downloadable as a separate library. The special edition for instructors is not provided on our website and is not open source, as this would make cheating very easy — see section 6.2. Instructors should contact the authors for a copy via the project website. The project website is <http://www.ling.upenn.edu/lambda>

2 Kinds of Exercises

As mentioned in the previous section, the application supports three kinds of exercises to be completed by the student. These three exercise kinds — type checking, reduction of lambda terms, and semantic derivations — are first presented by way of a walkthrough to the program. Later on we return to them in greater detail.

2.1 Walkthrough

This section is a detailed walkthrough that allows you to start working with the program and get an idea of its functionality as it presents itself to the student.

Here and in the following, we refer to version 1.0.5, the current version at the time of writing. We assume that you have the student edition available. If not, you can download the appropriate version from <http://www.ling.upenn.edu/lambda>, together with the sample exercise files on this website (right-click and save in most browsers). Even if you do not have access to the application, you can follow this section and refer to the figures to get an idea of how it works.

Double-click the program to start it and select “Interactive Exercise Solver”. Click on File in the menu, then Open. Select the file `example1.txt` and click Open.

The first exercise is displayed as in Figure 1.¹ It consists of the term $\lambda x.[P(x) \wedge Q(x)]$, of which you are asked to enter the type, based on the typing conventions displayed in the lower left hand corner of the window. Specifically, here P and Q are one-place predicate constants, and x is a variable of type e . The correct answer is therefore $\langle e, t \rangle$. Type this in and press Return to confirm, and again to move on to the next exercise.

The program now goes to a second type of exercise: reduction of lambda terms. It displays the term $\lambda x.[P(x) \wedge Q(x)](a)$, which you are asked to simplify by lambda conversion. Click Paste to copy the term into your answer box, then modify it, or start writing the reduced term from scratch if you prefer. To enter special characters like λ and \wedge , refer to the instructions in the middle left hand box. You can try various incorrect responses such as $[P(x) \wedge Q(x)](a)$ to observe the program’s responses.

When you are done (or bored) with the exercises in this file, open the next file `example2.txt` (see Figure 2). This third kind of exercise is very different. What you see is a syntax tree with some of the lexical entries already supplied. As explained in the instructions at the top of the main area of the window, your task consists in adding a lexical entry to the terminal α that is lacking one. As the text points out, the author of the exercise has used α to represent a reflexivizing morpheme.

To do this, click on that terminal α , then click into the text field below the tree and enter a lambda expression, conforming to the typing conventions displayed in the lower left hand corner. For example, to enter a variable of type $\langle e, \langle e, t \rangle \rangle$, use the letter R.

Confirm your choice of a lexical entry (the correct answer, in this case, is $\lambda R.\lambda x.R(x)(x)$) by hitting Return. It should now appear in the tree, under the terminal α . The tree is now ready to be semantically computed. Click on the VP node. You are now presented with a choice of three composition rules taken from

¹These screenshots have been taken within the Mac OS X operating system. The corresponding windows look slightly different on other operating systems.

Interactive Exercise Solver

Homework 1

- Part A. Semantic Types
 - 1. Type: $\lambda x.[P(x) \wedge Q(x)]$
- Part B. Lambda Conversion
 - 1. λ -Conv.: $\lambda x.[P(x) \wedge Q(x)] (a)$
 - 2. λ -Conv.: $\lambda x.\lambda y.[R(a,y) \wedge Q(x)] (e)$
 - 3. λ -Conv.: $\lambda x.\exists y.[R(y,x)] (y)$
 - 4. λ -Conv.: $\lambda x.[a] (b)$
 - 5. λ -Conv.: $\lambda x.[P(x) \rightarrow (\exists x)[R(x,b)]]$
 - 6. λ -Conv.: $\lambda x.\lambda x.[P(x) \rightarrow R(x,c)] (c)$

Part A. Semantic Types

Give the semantic type of the following lambda-expressions. You may want to simplify them in your mind if necessary before assigning a type.

1. Give the semantic type

Feedback

You have not yet started this exercise.

How to Enter Special Characters

When typing lambda expressions, use the following keyboard shortcuts:

- Type Ctrl-I for λ
- Type Ctrl-a, Ctrl-e, and Ctrl-i for $\forall, \exists,$ and ι
- Type & or ^ for \wedge and Ctrl-v for \vee
- Type the tilde (~) for \neg
- Type -> for \rightarrow and <-> for \leftrightarrow

Typing Conventions

Use the following typing conventions:

- a-c : constants of type e
- x-z : variables of type e
- P-Q : constants of type one-place predicate
- X-Y : variables of type one-place predicate
- R : constants of type two-place predicate

Figure 1: Type identification and lambda-conversion exercises.

Interactive Exercise Solver

Homework 2

- Part A. Formulating NatLg denotations as lambda-expressions
 - 1. Tree: Carlos α shaves
 - 2. Tree: Carlos β introduces Paco
 - 3. Tree: Carlos γ introduces Paco
 - 4. Tree: Carlos δ introduces Paco
- Part B. More NatLg denotations as lambda-expressions
 - 1. Tree: Ron Ag⁰ hug Pat⁰ Sue in
 - 2. Tree: far el Mukasa Pat⁰ khali
- Part C. Top-Down Semantic Computations
 - 1. Tree: the woman who₁ the topm
 - 2. Tree: the woman which₁ t₁ intr

Part A. Formulating NatLg denotations as lambda-expressions

In some languages, there is a morpheme (e.g., Middle Voice in Ancient Greek, reflexivizing affix in Kannada, Passive Voice in Finnish, etc.) that attaches to the verb stem and reduces its arity by one. Let us take the following imaginary morphemes α , β , γ , and δ . Assuming the syntactic structure given, give a denotation for each of these morphemes (in lambda-notation, for any arbitrary situation).

Note the typing conventions of P, R and S.

For the sentence in (1), make the structure below yield the meaning in (2) by supplying the denotation of α . (1)
 Carlos-SU α -shaves
 (2) "Carlos shaves Carlos" (i.e., Carlos shaves himself)

```

      graph TD
      IP --- Carlos
      IP --- VP
      VP --- alpha
      VP --- shaves
      style alpha stroke:#0000FF,stroke-width:2px
      
```

How to Enter Special Characters

When typing lambda expressions, use the following keyboard shortcuts:

- Type Ctrl-I for λ
- Type Ctrl-a, Ctrl-e, and Ctrl-i for \forall , \exists , and ι
- Type & or ^ for \wedge and Ctrl-v for \vee
- Type the tilde (~) for \neg
- Type -> for \rightarrow and <-> for \leftrightarrow

Typing Conventions

Use the following typing conventions:

- c j m p r - s : constants of type e
- x-z : variables of type e
- e : variables of type v
- P-Q X-Z : variables of type $\langle e, t \rangle$
- R : variables of type $\langle e, \langle e, t \rangle \rangle$
- S : variables of type $\langle e, \langle e, \langle e, t \rangle \rangle \rangle$
- T : variables of type $\langle v, t \rangle$

Enter an expression

Assign Denotation

Enter the denotation for the selected terminal node, or select a denotation from the list.

- $\lambda x. [\text{sleep}(x)]$
- $\lambda x. [\text{happy}(x)]$
- $\lambda x. [\text{guy}(x)]$
- $\lambda y. \lambda x. [\text{love}(x, y)]$
- $\lambda X. [\exists X(X)]$
- $\lambda X. [\forall X(X)]$

Figure 2: A natural language derivation exercise.

HK: function application, predicate modification, and lambda abstraction. Select the correct rule. The VP denotation will now change to the unreduced term

$$\lambda R.\lambda x.[R(x)(x)] (\lambda x.\lambda y.[shaves(y, x)]). \quad (1)$$

At this point, you are asked to reduce this lambda term. This corresponds to the second kind of exercise described above, and the program reacts to your input by feedback and error messages in exactly the same way as before. After three steps, this expression reduces to $\lambda x.shaves(x)(x)$, and you are asked to click on another node to continue. Click on the IP node and repeat the operation. You should end up with the formula $shaves(c, c)$ at the root. You are now free to go back and reassign lexical entries to terminal nodes or to select another exercise.

This completes our first overview of the *Penn Lambda Calculator* as it presents itself to the student. This walkthrough has not touched at all on several important features of the program, in particular the teacher-oriented functions. All of these will be described below. We begin by turning to a more systematic discussion of the kinds of exercises that the program supports.

2.2 Type Checking

The first kind of exercise, expected only to be used for a short time at the start of introductory semantics courses, asks the user to identify the semantic type (e , $\langle e, t \rangle$, $\langle\langle e, t \rangle, \langle e, t \rangle\rangle$, etc.) of expressions. The instructor provides a list of expressions for the student. The instructor does not need to provide the program with the answers, i.e. the type of each expression — this is computed by the program automatically based on typing conventions for constants and variables (either default conventions or ones provided by the instructor).

Some example problems are:

Problem	Answer
$P(x) \wedge \forall y[Q(y)]$	t
$\lambda x.P(x) \wedge \forall y[R(x, y)]$	$\langle e, t \rangle$
$\lambda x.\lambda y.\lambda z.P(x) (c)$	$\langle e, \langle e, t \rangle \rangle$
$\lambda x.\lambda w.sleeps(x, w)$	$\langle e, \langle s, t \rangle \rangle$

When the user provides an answer, the program first checks that the answer is a syntactically well-formed description of a type. For instance, $\langle ett \rangle$ is not well-formed. While the program does accept two common shortcuts (both et and $\langle et \rangle$ are acceptable), it is otherwise fairly strict with respect to how to enter semantic types. User answers that could not be understood as types are returned with a hopefully helpful diagnosis as to the problem. In the case of $\langle ett \rangle$, for which the user probably meant $\langle e, \langle tt \rangle \rangle$ or $\langle\langle et \rangle, t \rangle$, the program suggests that the user add brackets.

2.3 Reduction of Lambda Terms

Reduction of lambda terms (or lambda conversion as called in the program, i.e. β -reduction together with α -conversion) is one of the primary kinds of exercises in the program. For these exercises, the user is presented with a lambda expression and is asked to simplify it by performing lambda conversions one at a time. The centerpiece of the program is its informative feedback provided to students when incorrect answers are provided, and this is explained below. Special keyboard shortcuts are available to enter logical symbols. As with the type checking exercises, the instructor provides the program ahead of time with the problem, a lambda expression, but the program will compute the answer and any intermediate steps automatically. Intermediate steps may be necessary both because of the presence of multiple lambdas in the expression and because of the need to create an alphabetical variant:

Problem	Expected Answer
$\lambda x.[P(x) \wedge \forall x[Q(x)]] (a)$	$P(a) \wedge \forall x[Q(x)]$
$\lambda x.\lambda y.R(x, y) (a) (b)$	Step 1: $\lambda y.R(a, y) (b)$ Step 2: $R(a, b)$
$\lambda x.\forall y[R(x, y)] (y)$	Step 1 (e.g.): $\lambda x.\forall y'[R(x, y')] (y)$ Step 2: $\forall y'[R(y, y')]$

Student inputs are first checked for whether they are syntactically well-formed lambda expressions. If they are not, feedback is provided as to the nature of the problem. For instance, the expression $\lambda.P(x)$ is returned with feedback indicating that a lambda must be followed by a variable. The expression $P(a) \wedge Q(a) \vee P(b)$ is returned indicating that the expression is ambiguous and requires parentheses. (Issues that arose in parsing and providing feedback for lambda expressions are described in section 6.1.)

If the student input has passed the test of syntactic well-formedness, it is then checked for well-typedness according to the typing conventions in place. For instance, assume that x is associated with type e and Q is associated with a type other than e . A user response of $\lambda x.P(x) (Q)$ will be returned to the user explaining that $\lambda x.P(x)$ denotes a function whose range is over expressions of type e , but it cannot be applied to Q because Q is of another type.

If the student input is well-typed but incorrect, the program checks it to see if the student fell into a number of common pitfalls. These pitfalls are captured by about a dozen abstract triggers applied to the answer roughly in order of decreasing specificity. They represent the most common student errors as observed in a decade of teaching introductory semantics courses.

If a known pitfall is encountered, appropriate feedback is provided. Whenever possible, we generate constructive hints which do not give away the answer but suggest to the student how to proceed with the reduction. If the student input is detected to be wrong but none of the triggers are activated, a generic error message informs the student about this limitation (“I’m afraid I can’t help you here.”).

The response that is displayed to the student is a collection of diagnoses and hints that may be produced by different triggers. Experience has shown that students confronted with a list of error messages tend to read only the first. For this reason, the diagnoses and hints are rendered as a single paragraph in an attempt to obscure their origin as separate entities. If more than one diagnosis is displayed, they are ordered with the most specific diagnosis first, and joined by discourse connectives such as “also”.

Some of the possible feedback responses are given below, preceded by an incorrect attempt at lambda conversion that would activate the corresponding trigger.

PROBLEM: $\lambda x.P(x) (a)$

EXPECTED ANSWER: $P(a)$

STUDENT INPUT: $P(a) (a)$

PROGRAM RESPONSE: “After each λ -conversion, exactly one argument should be gone on the right hand side. Also, note that your expression also has a problem with types: $P(a)$ cannot be applied as a function to what looks like an argument to its right (a) because $P(a)$ is of type t according to the typing conventions in effect and therefore is not a function.”

PROBLEM: $\lambda x.\lambda y.R(x, y) (a) (b)$

EXPECTED ANSWER: Step 1: $\lambda y.R(a, y) (b)$ Step 2: $R(a, b)$

STUDENT INPUT: $\lambda x.R(x, a) (b)$

PROGRAM RESPONSE: “When doing λ -conversion, start with the outermost λ . Remember, the leftmost λ -slot corresponds to the leftmost argument to be λ -converted.”

PROBLEM: $\lambda x.\forall x[P(x)] (a)$

EXPECTED ANSWER: $\forall x[P(x)]$

STUDENT INPUT: $\forall x[P(a)]$

PROGRAM RESPONSE: “Remember to substitute the argument for all instances of the λ variable that are free in the body of the λ expression, and for nothing else. That is, make sure that you substituted into all and only the variable slots that were bound by the λ .”

PROBLEM: $\lambda x.\forall y[R(x, y)] (y)$

EXPECTED ANSWER: Step 1: $\lambda x.\forall y'[R(x, y')] (y)$ Step 2: $\forall y'[R(y, y')]$

STUDENT INPUT: $\forall y[R(y, y)]$

PROGRAM RESPONSE: “Your answer changed the truth conditions of the expression because a free variable in the argument was accidentally bound during substitution. Go back and try to make an alphabetical variant.”

PROBLEM: as in the previous example

EXPECTED ANSWER: as in the previous example

STUDENT INPUT: $\lambda x.\forall y[R(x, y)] (y')$

PROGRAM RESPONSE: “This is an incorrect alphabetical variant. Only bound variables can be rewritten as other variables while preserving truth conditions. Try making another alphabetical variant.”

2.4 Semantic Derivations

Semantic derivations are another important part of our program. In this kind of exercise, a Logical Form syntax tree is presented to the user, who is expected to provide lexical entries for terminal nodes, choose the applicable composition rule at each nonterminal (function application, predicate modification, or lambda abstraction), and evaluate and simplify the nonterminal nodes in a bottom-up fashion. (Top-down evaluation is planned for future work.) The program displays the tree visually, with the user-provided denotations of each node displayed at each node in the tree. The user enters lexical entries and denotations at the bottom of the screen. A blue box shows which node is to be acted on next, and this box can be moved through the tree by clicking a node with the mouse (see Figure 2).

Lambda expressions are parsed and checked for well-typedness as described above for lambda conversion exercises. During the simplification of the denotation of a nonterminal node, the same lambda conversion pitfalls as in those exercises are detected and reported as feedback. Additionally, the choice of an incorrect composition rule, such as the choice of function application on two nodes typed $\langle e, t \rangle$ each, is reported.

Currently, instructors do not provide the correct solutions for lexical entry questions. The mistake of the student providing the wrong lexical entry for a terminal node is expected to be found by the user on his or her own once either 1) the user gets stuck at a nonterminal node that cannot be evaluated because, for instance, the types of the children do not allow for any composition rule, or 2) the tree is fully evaluated, but the student realizes the denotation arrived at for the root node is incorrect. In either case, the user can go back and revise the incorrect lexical entry, and then re-evaluate the affected part of the tree.

One common student error in providing lexical entries is the confusion of the Predicate Logic two-place predicate R , as in $R(x, y)$, and the predicate R denoting a Schönfinkelized (or Curried) function from individuals to functions from individuals to truth values, as in $R(x)(y)$. (Only the latter term is of type $\langle e, \langle e, t \rangle \rangle$.) For instance, the student may be required to provide a function from a predicate of the latter type to a truth value and may incorrectly submit $\lambda R.R(x, x)$. In this case, the application recognizes the type mismatch and gives the feedback “ R is a function that takes (first) a single e -type argument alone, but you provided more than one argument. Rewrite your expression so that R is Schönfinkelized (i.e. each argument to R is surrounded by a separate pair of brackets).”

The following section describes the “teacher edition”, which can be used for performing semantic derivations in class.

3 Instructor Tools

3.1 Class Presentation Mode for Tree Derivations

The “teacher edition” of the *Penn Lambda Calculator* enhances the bottom-up derivation exercises (see previous section) with on-screen buttons to evaluate nodes in the tree automatically, rather than requiring the user to enter the denotation of each node and simplify it manually. Moreover, the type of each node is displayed in addition to its denotation. This mode is designed for in-class presentations as an alternative to the instructor writing out each step on a blackboard. It can also be used by the instructor to debug exercises he or she is writing for the students. The program can step forward and backward through simplification steps:

$$\llbracket \text{VP} \rrbracket^g (\llbracket \text{Carlos} \rrbracket^g) \leftrightarrow \lambda x. \text{shaves}(x, x) (c) \leftrightarrow \text{shaves}(c, c) \quad (2)$$

and can fill in entire subtrees with their denotations in one step to move quickly through the derivation.

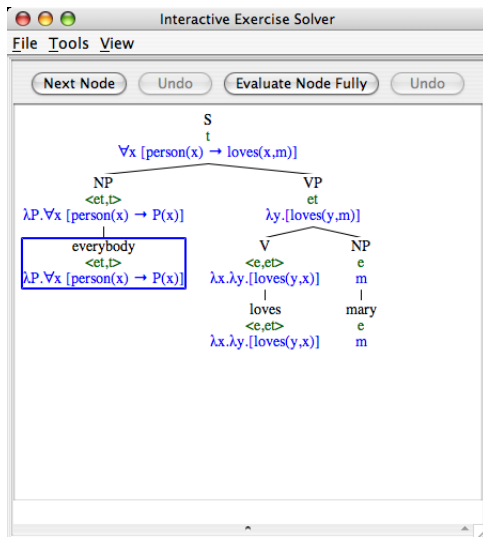
To prepare to use the presentation mode for tree derivations, the instructor creates a file containing the syntactic tree in labeled bracket notation, typing conventions for terms used in denotations, and any lexical entries that the instructor wants available ahead of time. (Additional lexical entries can be added while the program is running as well.) Because the program has the ability to simplify and combine lambda expressions, the instructor need not prepare the denotations of nonterminal nodes ahead of time. The appropriate composition rule at each step (e.g. function application versus predicate modification) is also chosen by the program based on the evaluated types of the daughter expressions, following the HK algorithm.

The program is able to represent the full range of phenomena covered in the HK textbook by function application, predicate modification, and lambda abstraction. This includes phenomena such as intersective adjectives, relative clauses and quantifier raising. As an example, derivations that illustrate different issues arising in connection with quantifiers are displayed in Figure 3. Figure 4 displays a complex noun phrase with two relative clauses of the kind that could easily take half an hour to draw on a blackboard. The simplification history of each node can be displayed in another box in the program (not shown in the figure).

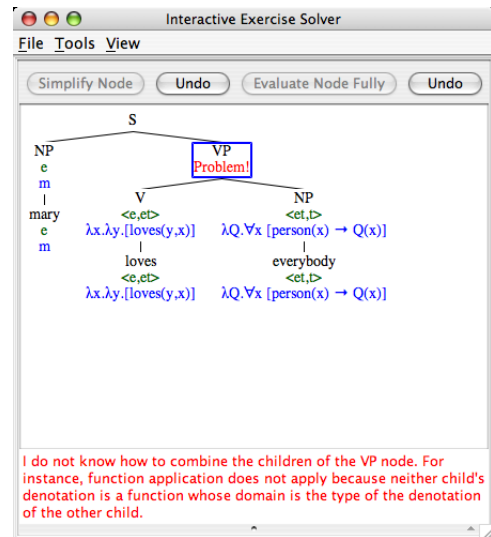
3.2 Creating Exercise Files

Exercises are provided by instructors to students in file form (e.g. via email or a webpage). Currently, exercise files are plain text files in which the instructor writes the title of the assignment, instructions, and each exercise one per line. Point values can be assigned to each problem in order to allow the program to compute grades automatically in the teacher review tool described in the next section. Plain text files can be created using any simple text editor (or word processor). The format of exercise files is documented on the website, which also provides samples.

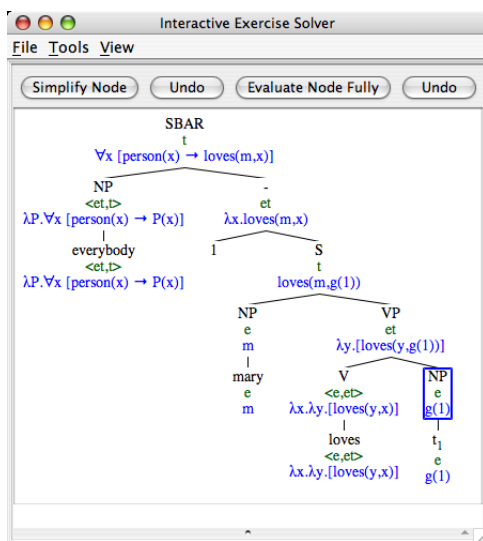
Although a plain text file format was chosen for exercise files for simplicity for the instructor, one drawback is that once sent to the students, the contents of these



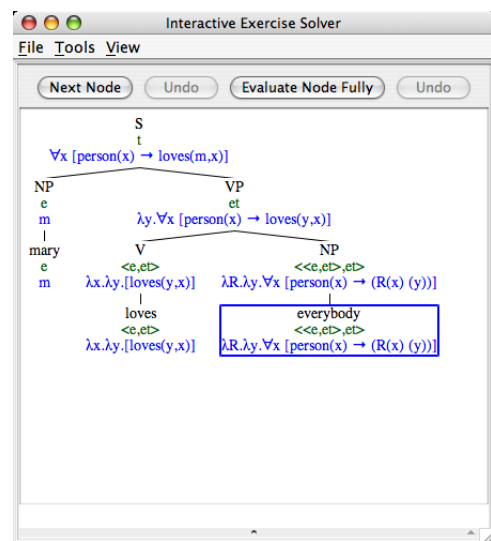
(a) A QNP that works in subject position ...



(b) ... does not work as an object unless ...



(c) ... you use quantifier raising or ...



(d) ... flexible types, i.e. another lexical entry.

Figure 3: Displaying various treatments of quantifiers using the teacher edition

files can be viewed by the students as well. For this reason, the instructor must be careful *not* to put the answers or any other such information in the file.

3.3 Homework and Teacher Review Tool

Students using the application for homework assignments can submit their work to their instructor by saving their progress to a file, which can then be e-mailed to the instructor. When saving, the program asks the student for his or her name, which is written into the saved-work file, along with the student's answers to the questions. This makes it easy for instructors to keep track of students' performance.

As with any submitted homework, there is, of course, no guarantee that a saved-work file actually represents any particular student's efforts. It was not a goal of the project to anticipate all of the many ways one might cheat using the program. The exercise files sent to students by instructors are plain-text files, as explained in the previous section. However, saved-work files are in binary format to make it at least non-trivial for students to modify a saved-work file once it has been created by the program, such as to put a different student's name in the file.

Saved-work files received by the instructor can be reviewed using the application. The review component of the program, called "Teacher Tool", displays detailed information on the student's answers to each individual exercise. A score is computed for those exercises whose answer can be automatically checked for correctness (all but the bottom-up derivations where the student needs to define a new lexical entry) and for which the exercise file has specified a score value. The application can also collect the scoring information of all the students and present it in a table along with mean and standard deviation for the final scores. The tool shows each student's final response to each of the questions in the homework, as well as the percentage of students who answered each problem correctly, and it allows the instructor to enter comments into the saved-work file for his or her own reference later. This table of student scores can be copied and imported into other programs such as spreadsheet applications for further processing.

4 Related Work

We were not able to find any software that would work as a companion to the HK textbook the same way as ours. However, some applications exist that do resemble ours, be it because they are also written for the linguistics classroom or because they support formal natural language semantics as well. In this section, we review and compare some of them to the *Penn Lambda Calculator*.²

²Space prevents us from doing justice to a number of additional related programs, such as CURT (Clever Use of Reasoning Tools), a collection of tools for first-order inference and translation from natural language that accompanies a textbook (Blackburn and Bos, 2005); and CLEARS (Konrad et al., 1996), an "interactive graphical environment for computational semantics" that supports various semantic formalisms such as Discourse Representation Theory (DRT) and situation semantics.

4.1 Semantica

Semantica (Larson et al., 1997) is perhaps the closest relative of the *Penn Lambda Calculator*. Like our program, it is an interactive, graphics-oriented application designed for assisting the student in learning to use a truth-conditional semantic derivation system. The original release of *Semantica* ran on the now defunct operating system NeXTstep, but its authors have since then re-released it for Windows.

The most important difference between the two programs is a difference in the underlying semantic theories. The HK framework, on which our program is based and which it faithfully reproduces, stands in the tradition of *type-driven translation*. This concept, introduced by Klein and Sag (1985) and Jacobson (1982) (see also (Dowty, 2006, p. 10)), denotes a semantic translation system in which the types of the expressions on the daughters of a syntactic tree node determine which semantic composition rule applies at that node. This allows one to decouple semantic rules from the syntax and to have only very few semantic rules. A pithier term for this idea, which Klein and Sag credit to Emmon Bach, is *shake'n'bake semantics*.

By contrast, many semantic translation systems have taken the grammar to include a set of rule pairs consisting of a phrase structure rule and a semantic composition rule. The best known example of this style of system is likely to be classical Montague semantics (Montague, 1974). Klein and Sag contrast this idea, termed *rule-to-rule hypothesis* by Bach (1976), to type-driven translation.

This dichotomy is also at the core of the main difference between the *Penn Lambda Calculator* and *Semantica*. Only the latter allows (and requires) the user to specify a different semantic composition rule for each syntactic phrase structure rule.³ In contrast, the *Penn Lambda Calculator* implements a system that is only equipped with a small collection of composition rules. Due to the type-driven nature of the HK computation system, these rules are sufficient to model a wide range of semantic phenomena in English.⁴

Both programs complement each other by offering important functions that the other one lacks:

- On the one hand, *Semantica* not only converts a syntactic tree to a logical formula, it also has the ability to evaluate that formula against a model, which consists of one or several worlds connected by modal and temporal relations. Each world is populated with individual objects of different kinds that stand in spatial relations to one another. The program contains an editor that allows the user to create and edit these models. This editor is quite easy to use. It is similar to and was modeled on the logic teaching program *Tarski's World* (see next section). *Semantica* can thus act as a simple theorem prover. The *Penn Lambda Calculator* is not able to do any of this.

³In practice, the *Semantica* user may load a file that contains a number of predefined rule-to-rule mappings of this kind.

⁴It is currently not possible for the user to add rules to this collection.

- On the other hand, *Semantica*'s emphasis on pedagogical issues and classroom management is not as strong. The program does not display the individual steps of the computation of a sentence's truth conditions, nor does it require the student to enter these steps. When the computation fails, only a generic error message is displayed that does not indicate the origin of the failure. ("Recheck rules and input tree.") Support for grading homework files in the style of our teacher tool is absent in *Semantica*. Perhaps for these reasons, using *Semantica* in the classroom has been reported to result in a "heavy initial burden" for the students and to require "considerably heavier time commitment than a traditional lecture-based course, both in terms of preparation and support" (Larson, 1997). Our experience with the *Penn Lambda Calculator* has been more encouraging (see Section 5).

Finally, a central difference is that *Semantica*'s underlying formalism does not make use of types nor of the lambda calculus, while the core functionality of the *Penn Lambda Calculator* consists in assisting students learning how to assign types to lambda terms and to reduce them.

4.2 Tarski's World

Tarski's World (released for Windows and Mac OS) is a pedagogical software program that helps students become fluent in first-order predicate logic. It displays logical formulae alongside graphical depictions of worlds (models) and asks the student to indicate whether any given formula is true in the world. Alternatively, the student could also be directed to build a world from scratch that makes a formula or collection of formulae true. Unlike *Semantica*, this program does not allow for models of modal or temporal logics, i.e. models in which several possible worlds are connected to each other by modal or temporal accessibility relations.

Tarski's World is similar to the *Penn Lambda Calculator* in that it focuses on providing helpful feedback to the student and on classroom management functions. It provides automatic grading via a central server, the *Grade Grinder*, to which students can electronically submit their files. However, this is where the similarities end: *Tarski's World* does not touch on natural language syntax or semantics.

4.3 Nessie

To conclude this section, we mention the *Nessie* project (Blackburn and Hinderer, 2007) as a recent example of an application created in the context of natural language formal semantics. Unlike the other programs presented here, *Nessie*'s approach is not pedagogical, and it is neither graphics-based nor interactive. The novelty of this project consists in its attempt at providing a generic framework for large-scale natural language semantic computation, based on the TY_n family of logics, which has been suggested as a uniform framework for virtually any kind of semantic analysis (Muskens, 1996). TY_n is based on the simply typed lambda

calculus and is therefore very similar to the logic underlying HK and our system. Furthermore, TY_n provides flexible support for any number of basic kinds of entities such as ordinary individuals, belief states, times, and situations. *Nessie*, a platform-independent application, fully implements TY_n and is developed with the aim of providing “a systematic way of combining the insights from many different approaches, ranging from DRT through situation semantics and classical possible world semantics, to event based semantics” (Blackburn and Hinderer, 2007, p. 5).

5 Field Experience

An early version of this program has been field-tested in the Spring 2007 graduate student introductory course to formal semantics at the University of Pennsylvania and has later undergone extensive usability testing in order to improve its user interface. In its current form (the result of about 400 man-hours of work), it has been deployed for the first time in the introductory course to semantics at the Linguistic Society of America Summer Institute 2007, at Stanford. Both courses have been taught by one of us (Romero). We have offered an internet forum in order to collect feedback from the students and to provide technical support. We expected to have to make changes to the program and to redeploy it several times as the course proceeded, but this turned out not to be necessary. Students used the same version of the application throughout the course. The forum was used primarily to clarify questions in the exercises rather than to ask questions about the program itself. Numerous minor improvements to the application were suggested and bugs were collected. As a result, we expect its basic design to remain stable in the near future.

The teacher edition’s ability to demonstrate a derivation on the screen turned out extremely helpful in the classroom. Even if one does it slowly enough so that the students have time to assimilate what is on the screen, it looks cleaner and saves time compared to writing the same derivation on the blackboard. A derivation that used to take us 30 minutes on the blackboard takes about 5-10 minutes using the application, depending on how much explanation is needed.

6 Issues in Program Development

6.1 Robust Parsing of a Formal Language

The syntax of the lambda calculus is usually given as a collection of CFG or BNF rules or as a recursive definition to that effect, together with the statement that when the formulae are presented to a human reader, parentheses can be dropped for convenience. To parse typed lambda calculus expressions entered by students and teachers, we needed to implement a “robust” syntax, able to handle these omitted parentheses and similar pitfalls. (We soon discovered that it was not advisable to force users to disambiguate every formula with parentheses, since this soon led to frustration, and it distracted users from the task at hand.)

Informally, parentheses may be dropped just in case the resulting expression appears unambiguous to the human reader. The exact conditions for this, as well as the rules that disambiguate these expressions, turned out surprisingly difficult to determine. Even our own experience with the typed lambda calculus did not allow us to define the rules we seemed to have unconsciously mastered, and so we had to discover them empirically.⁵ We discuss a few examples here.

The most striking phenomenon was the significance of spaces in expressions of function application. For instance, when the type of M was not specified, the expression $\lambda x.M(x) (a)$ was most likely to be interpreted as intending $\lambda x.M(x)$ to be applied to the argument a . However, the expression $\lambda x.M(x)(a)$ was understood as having a as the second argument of M itself (where M is now understood as a Schönfinkelized two-argument function). The difference is one of scope, with (a) in the first case having wide scope relative to the lambda, and in the second case narrow scope. Apparently, though, these structural preferences can be overridden: in our experience, most people would be reluctant to interpret the first x to be bound and the second x to be free in the expression $\lambda x.M(x) (x)$, regardless of the presence of space.

In some cases, we are able to use the fact that parentheses are regularly omitted because there is only one well-typed bracketing. $\lambda x.T(x)(a) \wedge U(b)$ is an example of this. Without knowing the types of T and U , the program will reject this expression on the grounds that it is ambiguous. However, if T is known to be of type $\langle e, t \rangle$, the program will give the user the benefit of the doubt and understand the expression as $[(\lambda x.T(x)) (a)] \wedge U(b)$. If T is instead entered as $\langle e, \langle e, t \rangle \rangle$, the expression will be treated as $\lambda x.(T(x)(a) \wedge U(b))$.

6.2 Issues in Distribution

In designing this application, we made the decision early on to make as much functionality available for free under a GPL-like license (Stallman, 2007), including the source code. At the same time, some of the functionality cannot be distributed to students. In this section, we discuss some issues that arise from this conflict. At the time of writing, these are open and serious problems for us, and we are grateful for any suggestions.

As mentioned above, we are currently only offering the student edition of the application on the project website. The reason for this is that the teacher edition has capabilities that would easily allow students to solve any exercise with almost no effort at all. Therefore we feel its distribution must be restricted. We are currently exploring different ways to manage this restriction:

- One option we are considering is making the teacher edition available as a restricted download. Only individuals who can document to us their affiliation as university faculty would be given access to the program. While this will create a certain delay in distribution, we anticipate that the added work for us

⁵The parallel to natural language syntax has not escaped our attention.

will be manageable and the delay short. However, this has the problem that a single instructor who, for whatever reason, makes the teacher edition available to some student might result in both versions being effectively freely available to all students.

- Another option, which avoids this problem, would consist in making the teacher edition available only as a password-protected web-based application. However, this would require that the instructor have Internet access during class sessions.
- A related issue concerns the extent to which we can release the source code. While we would like the full program to be available to be modified and reused by others, providing the full source code would allow others to compile and make the teacher edition available to students. We prefer to err on the side of caution and are currently making the source code available only partially. We will be happy to release the full source code to those that we would provide the teacher edition to.

7 Future Work

The *Penn Lambda Calculator* is usable in its current state; however, improvements are planned in several areas. Lambda expressions understood by the program will continue to be extended and refined to accommodate nonstandard ways of entering lambda expressions and to address pedagogical concerns. We will allow the program to accept expressions containing mathematical, set, and modal operators not yet considered, and situation variables as superscripts on interpretation functions. The set of semantic computation rules, which is hard-coded into the program at the moment, could be made user-extensible. We also plan to add support for top-down HK derivations.

A drawback of the rigid distinction between “teacher” and “student” editions of the application is that it is impossible for the instructor to allow the students to step through derivations at their own pace, unless he or she wants to give students access to the “teacher” edition. Currently, students can only watch the derivations as the instructor steps through them in class. If they try to replicate them in the “student” edition, they have to re-enter by hand all the lambda conversions involved in the derivation. This problem has emerged in the classroom and was not foreseen by us. We plan to address it by providing the instructor with a means to selectively unlock the student edition’s features for certain derivations only.

Finally, we intend to improve the integration of our program with related software. In particular, we plan to add the ability to exchange syntactic trees between the *Penn Lambda Calculator* and the *Trees* program, a learning environment for syntactic theory (Kroch and Crist, 2002), as well as the \LaTeX tree-drawing package *qtrees*. We may also link up the program with *Tarski’s World* and/or *Semantica*

(see section 4) in order to provide students with a way to check the truth of their sentences in a self-constructed model.

References

- Bach, Emmon. 1976. An extension of classical transformational grammar. In *Problems in Linguistic Metatheory, Proceedings of the 1976 Conference at Michigan State University*.
- Barwise, Jon and Etchemendy, John. 1999. *Language, Proof, and Logic*. CSLI Publications, Stanford.
- Blackburn, Patrick and Bos, Johan. 2005. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI Publications, Stanford.
- Blackburn, Patrick and Hinderer, Sébastien. 2007. From TY_n to DRT: an implementation. In *Proceedings of the 3rd International Language & Technology Conference (L&TC'07)*, pages 384–388.
- Dowty, David R. 2006. Compositionality as an empirical problem, unpublished manuscript, online 2007-10-14: <http://www.ling.ohio-state.edu/~dowty/context-free-semantics.pdf>.
- Heim, Irene and Kratzer, Angelika. 1998. *Semantics in Generative Grammar*. Oxford: Blackwell.
- Jacobson, Pauline. 1982. Visser Revisited. In Kevin Tuite, Robinson Schneider and Robert Chametsky (eds.), *Papers from the 18th Regional Meeting, Chicago Linguistic Society, April 15-16, 1982*, volume 18, pages 218–243.
- Klein, Ewan and Sag, Ivan A. 1985. Type-driven translation. *Linguistics and Philosophy* 8(2), 163–201.
- Konrad, Karsten, Maier, Holger and Pinkal, Manfred. 1996. CLEARs - an education and research tool for computational semantics. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING), Copenhagen*.
- Kroch, Anthony and Crist, Sean. 2002. Trees 3 for Windows. Pedagogical software. Online 2007-10-14: <http://www.ling.upenn.edu/~kroch/Trees.html>.
- Larson, Richard K. 1997. The Grammar as Science project. Day 2 (Semantica). Presentation at the Linguistic Society of America Summer Institute, Workshop on Linguistics and the Language Sciences: New Computer-based Methods and Materials for Undergraduate Education, Cornell University, Ithaca, NY. Online 2007-10-08: <http://semlab5.sbs.sunysb.edu/~rlarson/day2.pdf>.

Larson, Richard K., Warren, D.S., de Lima e Silva, J. Freire, Gomez, P. and Sagonas, K. 1997. *Semantica*. MIT Press, Cambridge.

Montague, Richard. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Edited and with an introduction by R. H. Thomason. Yale University Press, New Haven.

Muskens, Reinhard. 1996. *Meaning and Partiality*. Studies in Logic, Language and Information, CSLI Publications, Stanford.

Stallman, Richard. 2007. GNU General Public License.