Generating Quantifiers and Negation to Explain Homework Testing

Jason Perry and Chung-chieh Shan

Rutgers University

June 5, 2010

Slides available at
`http://paul.rutgers.edu/~jasperry/gqn_bea.pdf`

Automated Programming Assignment Checking

- Professor teaching Programming 101 types requirement:
  "Every source file compiles and 'Readme.txt' mentions every source file."

Automated Programming Assignment Checking

- Professor teaching Programming 101 types requirement:
  "Every source file compiles and 'Readme.txt' mentions every source file."
- Computer understands this requirement and automatically checks students' files.

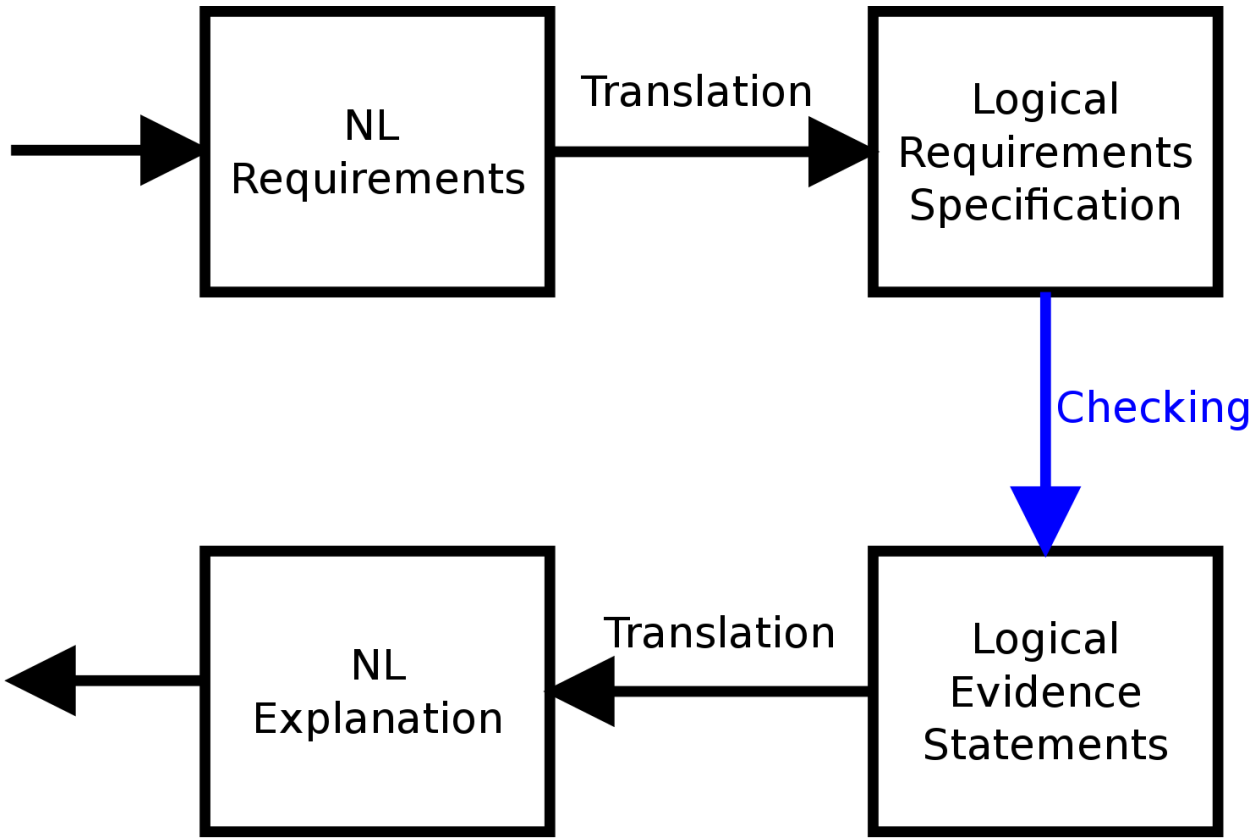
Automated Programming Assignment Checking

- Professor teaching Programming 101 types requirement:
  "Every source file compiles and 'Readme.txt' mentions every source file."
- Computer understands this requirement and automatically checks students' files.
- Student automatically receives feedback:
  "Credit was lost because 'foo.c' doesn't compile and 'Readme.txt' doesn't mention 'bar.c'."

Goals

- Workable automated checking of basic programming assignment requirements with a natural language interface.
- A study of quantifiers and negation in NL generation, within an end-to-end NLP framework.

Prograder NLP System Architecture Overview



Semantics of a Requirements Statement

- Each requirement specification is a sentence, whose truth value is determined by checking a single student's programming assignment.
- Use the types of Montague grammar, which are the base type of entities e, the base type of propositions t, and function types notated by $\rightarrow$.

Semantics of a Requirements Statement

- Each requirement specification is a sentence, whose truth value is determined by checking a single student's programming assignment.
- Use the types of Montague grammar, which are the base type of entities e, the base type of propositions t, and function types notated by $\rightarrow$.
- A domain-specific *first-order* language, executable in Python

Semantics of a Requirements Statement

- Each requirement specification is a sentence, whose truth value is determined by checking a single student's programming assignment.
- Use the types of Montague grammar, which are the base type of entities e, the base type of propositions t, and function types notated by $\rightarrow$.
- A domain-specific *first-order* language, executable in Python

```
and (everysourcefile (lambda x : compiles (x)) ,
     everysourcefile (lambda z : mentions (z, ("Readme.txt"))))
```

Semantics of a Requirements Statement

- Each requirement specification is a sentence, whose truth value is determined by checking a single student's programming assignment.
- Use the types of Montague grammar, which are the base type of entities e, the base type of propositions t, and function types notated by $\rightarrow$.
- A domain-specific *first-order* language, executable in Python

```
and (everysourcefile (lambda x : compiles (x)) ,
     everysourcefile (lambda z : mentions (z, ("Readme.txt"))))
```

- Checking of quantified statements is done through iteration over the domain (submitted files).

Explaining Truth Values

- The checking code should produce not only a truth value but also an explanation of that value.

Explaining Truth Values

- The checking code should produce not only a truth value but also an explanation of that value.
- An explanation of a truth value may be viewed as a conjunction of a sufficient number of evidence statements, one for each failed check, in the same logical language:

Explaining Truth Values

- The checking code should produce not only a truth value but also an explanation of that value.
- An explanation of a truth value may be viewed as a conjunction of a sufficient number of evidence statements, one for each failed check, in the same logical language:

```
not(compiles('foo.c'))
not(mentions('bar.c')('Readme.txt'))
```

Explaining Truth Values

- The checking code should produce not only a truth value but also an explanation of that value.
- An explanation of a truth value may be viewed as a conjunction of a sufficient number of evidence statements, one for each failed check, in the same logical language:

```
not(compiles('foo.c'))
not(mentions('bar.c')('Readme.txt'))
```

- Expand the type definition of truth value: instead of just a boolean, use a *(boolean, explanation)* pair.

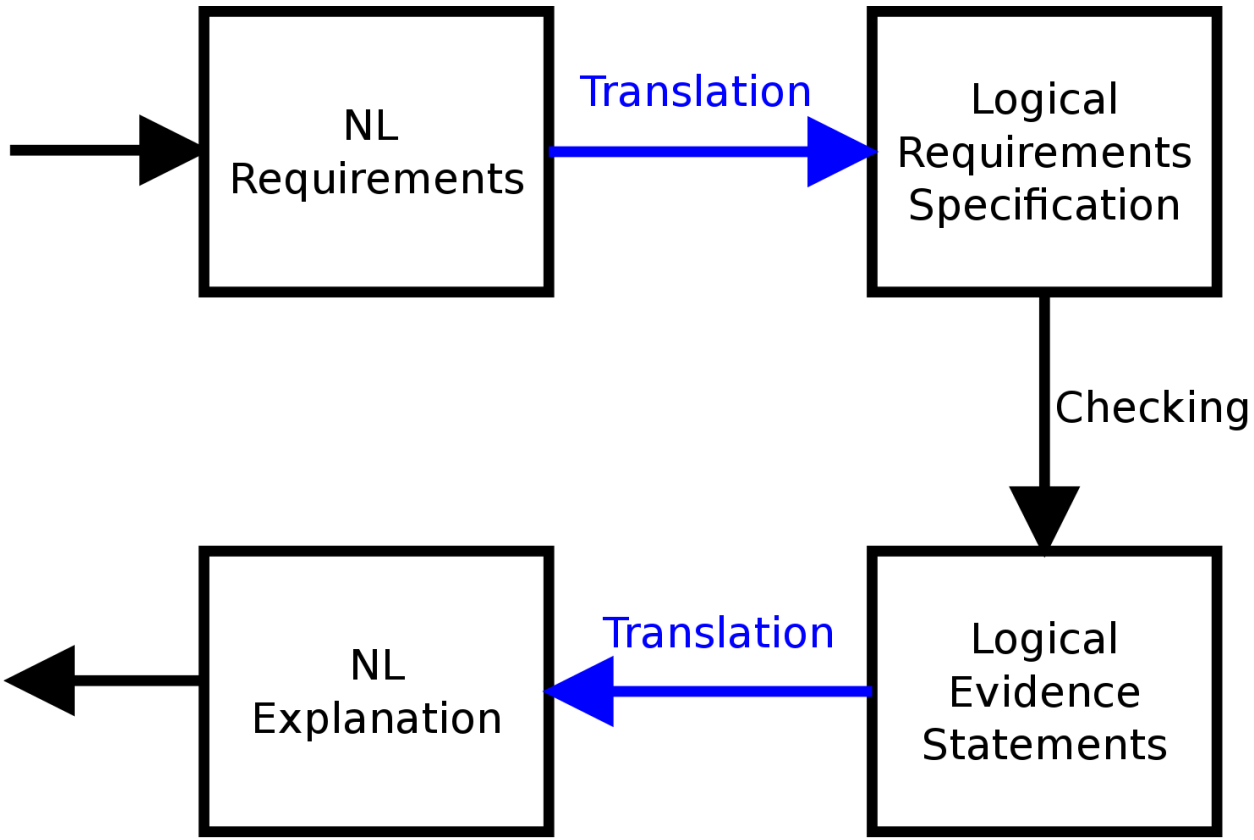
Explaining Truth Values

- The checking code should produce not only a truth value but also an explanation of that value.
- An explanation of a truth value may be viewed as a conjunction of a sufficient number of evidence statements, one for each failed check, in the same logical language:

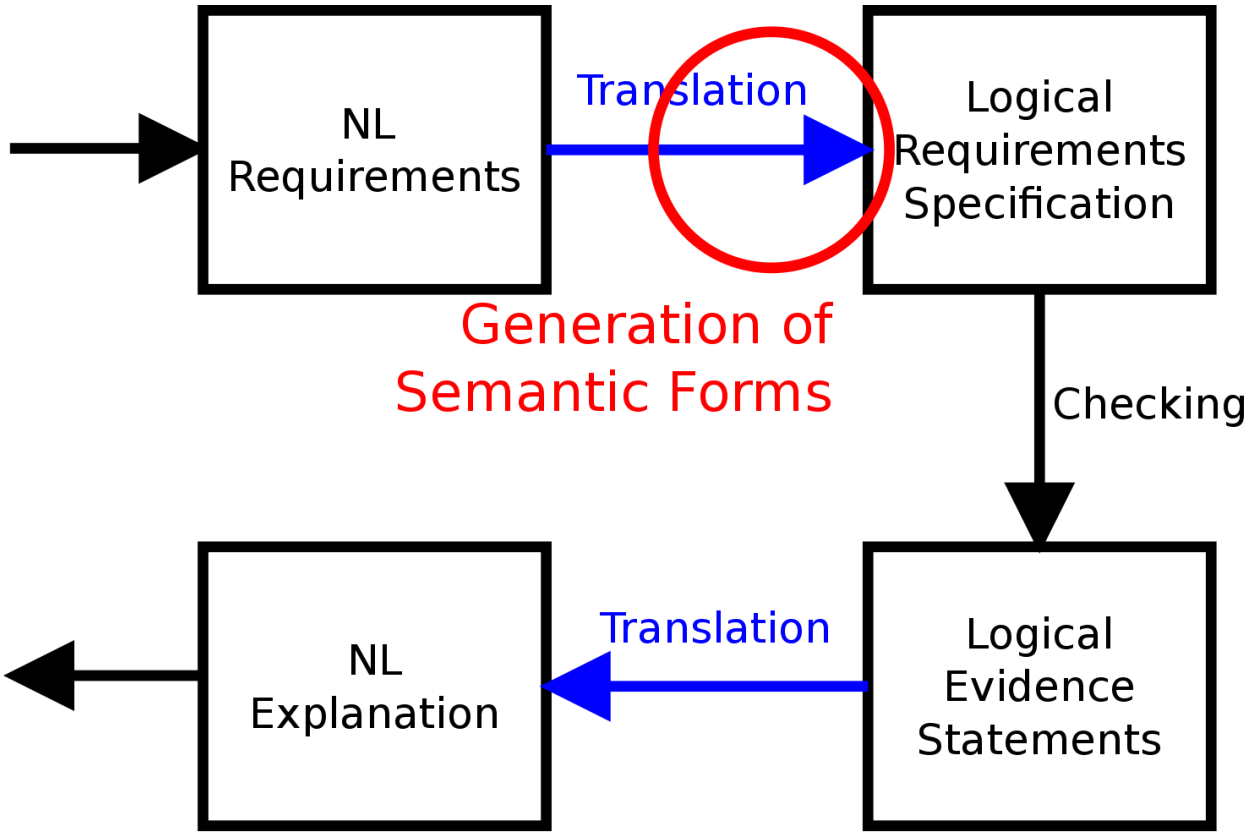
```
not(compiles('foo.c'))
not(mentions('bar.c')('Readme.txt'))
```

- Expand the type definition of truth value: instead of just a boolean, use a *(boolean, explanation)* pair.
- Summarization by grouping/quantifying over evidence statements

Prograder NLP System Architecture



Prograder Architecture - the Hard Part



Aarne Ranta's Grammatical Framework

`http://www.grammaticalframework.org/`

- A type-theoretical framework for symmetric parsing and linearization

Aarne Ranta's Grammatical Framework

`http://www.grammaticalframework.org/`

- A type-theoretical framework for symmetric parsing and linearization
- Supports translation through separation into an *abstract grammar* and *concrete grammar*

Aarne Ranta's Grammatical Framework

`http://www.grammaticalframework.org/`

- A type-theoretical framework for symmetric parsing and linearization
- Supports translation through separation into an *abstract grammar* and *concrete grammar*
  - Common abstract grammar, separate concrete grammar for each language
  - Parse using one concrete grammar, generate using the other, and vice-versa

Aarne Ranta's Grammatical Framework

`http://www.grammaticalframework.org/`

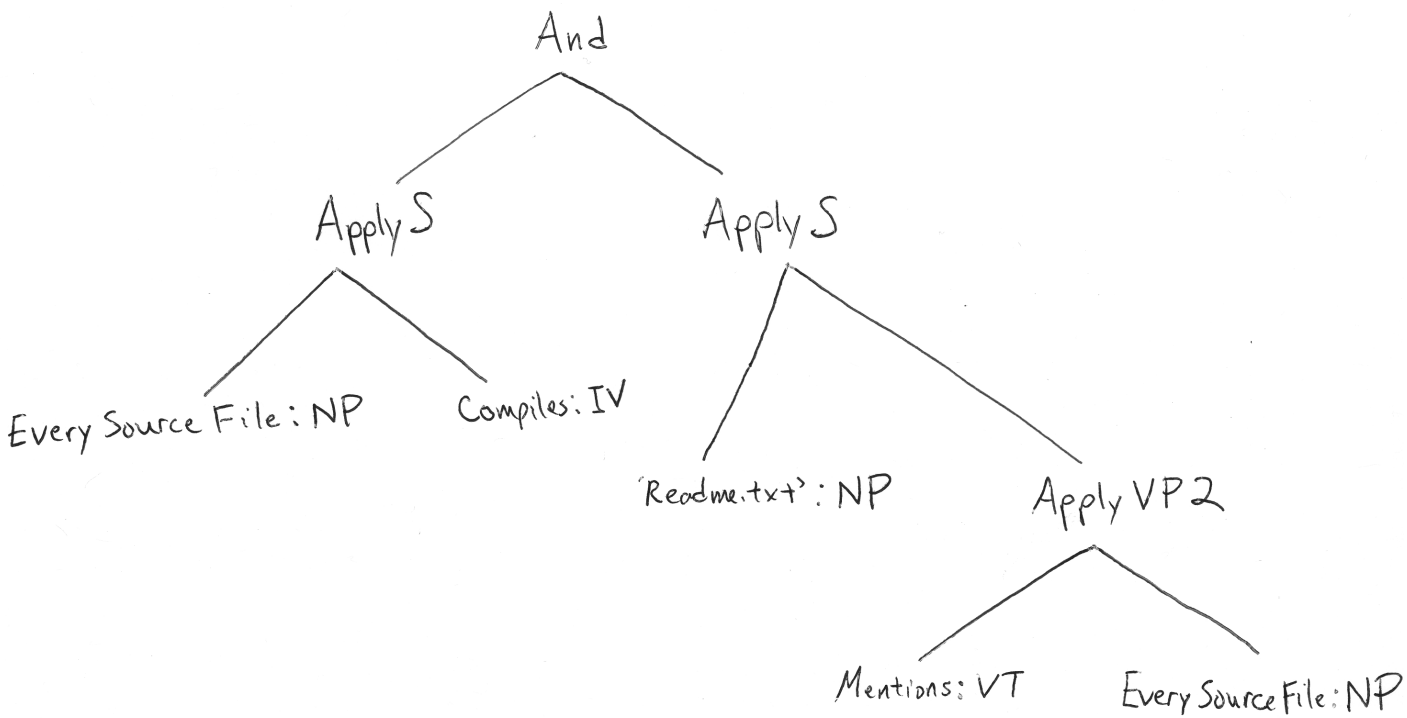
- A type-theoretical framework for symmetric parsing and linearization
- Supports translation through separation into an *abstract grammar* and *concrete grammar*
  - Common abstract grammar, separate concrete grammar for each language
  - Parse using one concrete grammar, generate using the other, and vice-versa
- Abstract grammar is functional, concrete grammar uses string concatenation with record structures for efficiency (context-free+)

Aarne Ranta's Grammatical Framework

`http://www.grammaticalframework.org/`

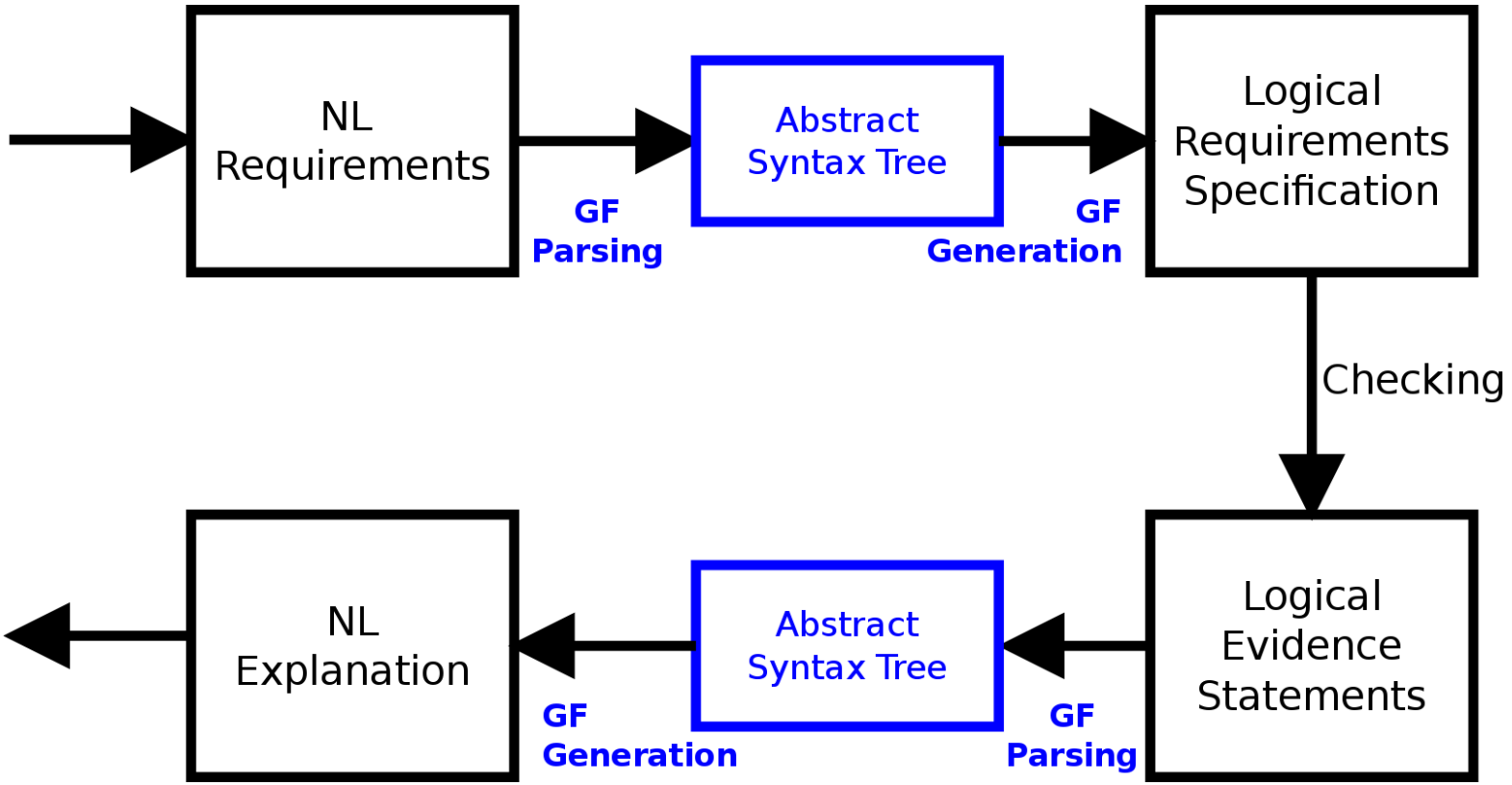
- A type-theoretical framework for symmetric parsing and linearization
- Supports translation through separation into an *abstract grammar* and *concrete grammar*
  - Common abstract grammar, separate concrete grammar for each language
  - Parse using one concrete grammar, generate using the other, and vice-versa
- Abstract grammar is functional, concrete grammar uses string concatenation with record structures for efficiency (context-free+)
- **Prograder uses one concrete grammar for parsing/generating English, another for the Python logical form.**

Parsing Requirements to Abstract Syntax



- Non-phrase-structure aspects such such as agreement are handled in the (English) concrete grammar.
- Quantified NPs are not distinguished from syntactic NPs in the abstract syntax.

Prograder NLP Architecture Revisited



Produce a logical semantic representation from the syntax tree *using GF's generation capability*.

Quantifier Scoping Overview

- *"A file mentions every source file"*: How to implement semantics of quantifier scoping?

Quantifier Scoping Overview

- *"A file mentions every source file"*: How to implement semantics of quantifier scoping?
  - Semantic attachments to the syntax tree, in the form of lambda expressions representing the denotation of categories
  - Expressions are combined with a composition rule and beta-reduced
  - Composition rules can be specified by a functional/categorial grammar

Quantifier Scoping Overview

- *"A file mentions every source file"*: How to implement semantics of quantifier scoping?
  - Semantic attachments to the syntax tree, in the form of lambda expressions representing the denotation of categories
  - Expressions are combined with a composition rule and beta-reduced
  - Composition rules can be specified by a functional/categorial grammar

What kind of expressions/application rules implement scoping?

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.
- NP's have this structure in Montague Grammar: type $((e \to t) \to t)$ instead of e.

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.
- NP's have this structure in Montague Grammar: type $((e \to t) \to t)$ instead of e.
- Continuation grammars [Barker & Shan] generalize the use of higher-order functions in grammar rules to provide access to continuation for all constituents.

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.
- NP's have this structure in Montague Grammar: type $((e \to t) \to t)$ instead of e.
- Continuation grammars [Barker & Shan] generalize the use of higher-order functions in grammar rules to provide access to continuation for all constituents.
- Treat each constituent, quantified or non-quantified, as having access to its own continuation.

Surface Scope with Continuation Grammar Rules

Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.
- NP's have this structure in Montague Grammar: type $((e \to t) \to t)$ instead of e.
- Continuation grammars [Barker & Shan] generalize the use of higher-order functions in grammar rules to provide access to continuation for all constituents.
- Treat each constituent, quantified or non-quantified, as having access to its own continuation.

```
fun ApplyS NP VP
    = NP(lambda n: VP(lambda v: v(n)))
```

Surface Scope with Continuation Grammar Rules

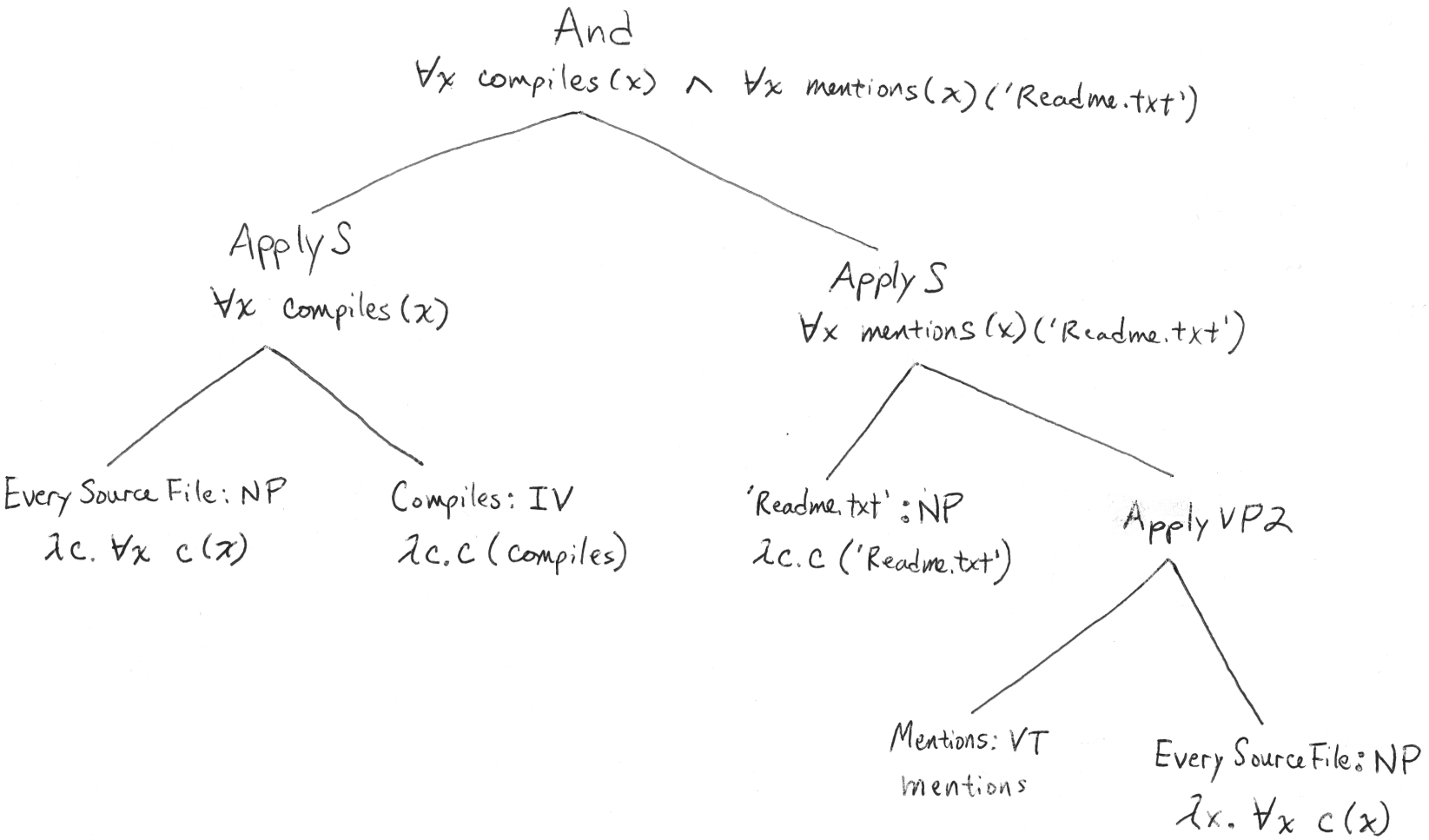
Express scoping preferences directly in the grammar by means of continuized denotations and combination rules.

- Model the ability of constituents to take scope over others.
- NP's have this structure in Montague Grammar: type $((e \to t) \to t)$ instead of e.
- Continuation grammars [Barker & Shan] generalize the use of higher-order functions in grammar rules to provide access to continuation for all constituents.
- Treat each constituent, quantified or non-quantified, as having access to its own continuation.

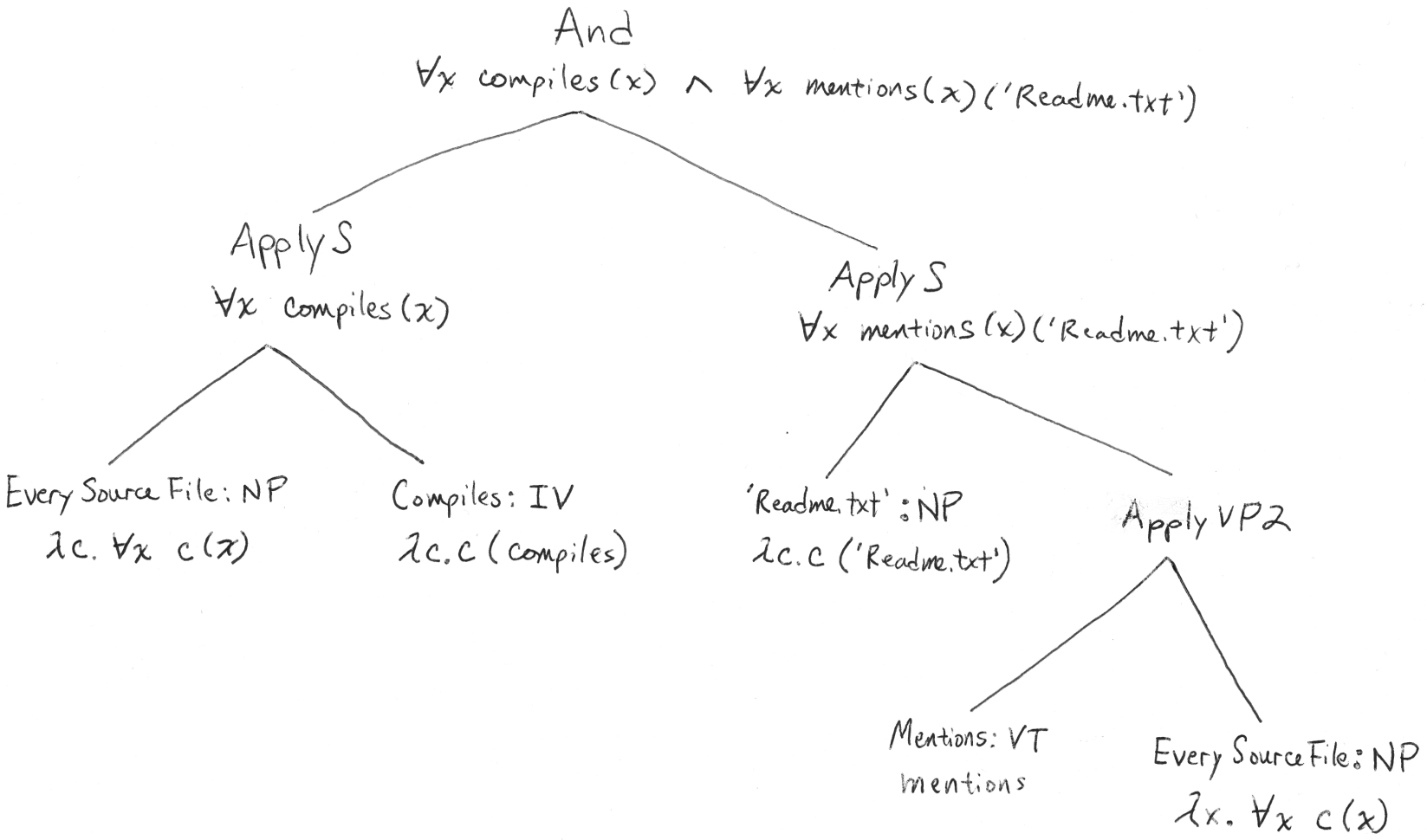
```
fun ApplyS NP VP
    = NP(lambda n: VP(lambda v: v(n)))
```

- Represents surface scope: NP can take scope over VP

Abstract Syntax Tree with Semantic Attachments



Abstract Syntax Tree with Semantic Attachments



Can we generate/combine such representations using GF?

Simulating Bounded-depth Continuations with String Concatenation

- We want to generate the logical form in GF using a concrete grammar.

Simulating Bounded-depth Continuations with String Concatenation

- We want to generate the logical form in GF using a concrete grammar.
  ... then we get parsing for free

Simulating Bounded-depth Continuations with String Concatenation

- We want to generate the logical form in GF using a concrete grammar.
  ... then we get parsing for free
- But GF's linearization is limited to string concatenation - no higher-order functions allowed.

Simulating Bounded-depth Continuations with String Concatenation

- We want to generate the logical form in GF using a concrete grammar.
  ... then we get parsing for free
- But GF's linearization is limited to string concatenation - no higher-order functions allowed.

Solution: simulate higher-order functions with interleaved record fields.

Simulating Bounded-depth Continuations with String Concatenation

```
lin foosource =
    { "", "'foo.c'", "" }
lin everysourcefile =
    { "everysourcefile(lambda x:", "x", ")" }
lin compiles =
    { "", "compiles", "" }
```

Simulating Bounded-depth Continuations with String Concatenation

```
lin foosource =
    { "", "'foo.c'", "" }
lin everysourcefile =
    { "everysourcefile(lambda x:", "x", ")" }
lin compiles =
    { "", "compiles", "" }
```

- 'Apply' functions simply interleave and concatenate the record fields.

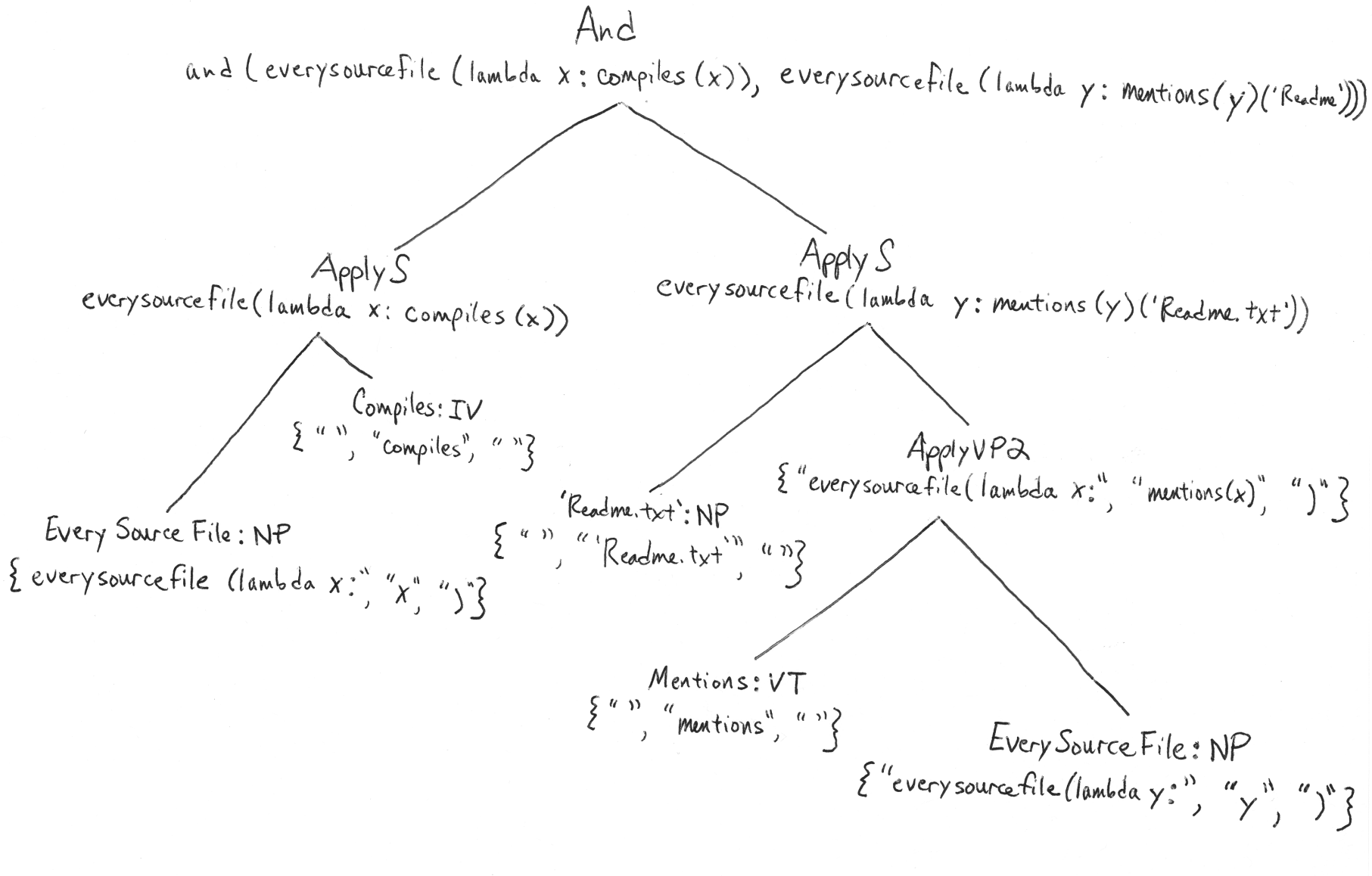
Simulating Bounded-depth Continuations with String Concatenation

```
lin foosource =
    { "", "'foo.c'", "" }
lin everysourcefile =
    { "everysourcefile(lambda x:", "x", ")" }
lin compiles =
    { "", "compiles", "" }
```

- 'Apply' functions simply interleave and concatenate the record fields.

```
"compiles('foo.c')"
"everysourcefile(lambda x: compiles(x))"
```

GF Generations as Semantic Attachments



Negatives as Quantifiers

- Negatives have scope too: *"Every source file doesn't compile"* versus *"Not Every source file compiles"*
- Important to handle negatives tastefully in the explanation to the student

Negatives as Quantifiers

- Negatives have scope too: *"Every source file doesn't compile"* versus *"Not Every source file compiles"*
- Important to handle negatives tastefully in the explanation to the student
  "It is not the case that not every source file doesn't compile"

Negatives as Quantifiers

- Negatives have scope too: *"Every source file doesn't compile"* versus *"Not Every source file compiles"*
- Important to handle negatives tastefully in the explanation to the student
  "It is not the case that not every source file doesn't compile"
- Claim: More natural-sounding sentences are generated when negation is pushed all the way in using De Morgan's rule.

Negatives as Quantifiers

- Negatives have scope too: *"Every source file doesn't compile"* versus *"Not Every source file compiles"*
- Important to handle negatives tastefully in the explanation to the student
  "It is not the case that not every source file doesn't compile"
- Claim: More natural-sounding sentences are generated when negation is pushed all the way in using De Morgan's rule.
  "No source file compiles"

"Generating" De Morgan's Rule in GF

- Every statement in the logical grammar should be parsed into an abstract tree with negation all the way in.

"Generating" De Morgan's Rule in GF

- Every statement in the logical grammar should be parsed into an abstract tree with negation all the way in.
- A grammar doesn't 'know' De Morgan's rule such that it can preserve semantics of negation (move negatives inside and flip quantifiers)

"Generating" De Morgan's Rule in GF

- Every statement in the logical grammar should be parsed into an abstract tree with negation all the way in.
- A grammar doesn't 'know' De Morgan's rule such that it can preserve semantics of negation (move negatives inside and flip quantifiers)
- But we can simulate it by storing two versions of each record, one for the original quantifiers and one with the dual, with a 'switched' flag.

"Generating" De Morgan's Rule in GF

- Every statement in the logical grammar should be parsed into an abstract tree with negation all the way in.
- A grammar doesn't 'know' De Morgan's rule such that it can preserve semantics of negation (move negatives inside and flip quantifiers)
- But we can simulate it by storing two versions of each record, one for the original quantifiers and one with the dual, with a 'switched' flag.

```
noteverysourcefile (not (not (compiles)))

"not every source file compiles"
```

Sample Output

```
$ ./runPrograder.py assn 'every source file compiles and
        "README" mentions every source file'
*****************************
RESULT: False, because:
a source file doesn't compile and "README" doesn't mention
        every source file
"nowork2.c" doesn't compile
"nowork.c" doesn't compile
"README" doesn't mention "nowork2.c"
"README" doesn't mention "hello.c"
"README" doesn't mention "work1.c"
```

Conclusion

- Simulating continuized grammar rules with records is a workable way to generate logical forms of quantified and negated statements for NLP applications, while keeping parsing and generation tractable.

Conclusion

- Simulating continuized grammar rules with records is a workable way to generate logical forms of quantified and negated statements for NLP applications, while keeping parsing and generation tractable.
- Grad students don't have to write as many scripts, professors don't have to learn a new system.

Continuing Work

- Testing with actual professors and students, expanding the vocabulary
- Finish the general formulation of the continuation-rule-to-record mapping, including higher-order quantifiers e.g. 'some'
- Further investigation of summarization within the type-theoretic framework

The End

Thank you!