Semantic Parser

6.S083 Fall 2015 Final Project

Jessica Kenney
December 18, 2015

Contents

1 Overview

Semantic Parser translates sentences of English into sentences of predicate logic, and then generates conclusions or checks given conclusions. For example, generating conclusions:
Input:

"John is both a man and a farmer.
Every man eats."

Translation:

(Mj $\wedge$ Fj)
($\forall$x)(Mx $\rightarrow$ Ex)

Generated Conclusions:

Ej "John eats"
($\exists$x)(Fx $\wedge$ Ex) "There is some entity x such that x is a farmer and x eats"

And validating conclusions:
Input:

"Every man eats.
Marc is a man.
Therefore, Marc drinks a coke."

Translation:

($\forall$x)(Mx $\rightarrow$ Ex)
Mm
$\therefore$ ($\exists$x)(Cx $\wedge$ Dmx)

Validity Testing:

INVALID
Counterexample: Universe(
    Domain: m, hypothetical_a
    Set C: {hypothetical_a}
    Set M: {m}
    Set E: {m}
    Set D: {}
)

In the above counterexample universe, Marc is a man and Marc eats, and there is a coke, but Marc doesn't drink it. This universe is possible given the two priors, and the conclusion is false in this universe, so the conclusion is invalid.

1.1 Running

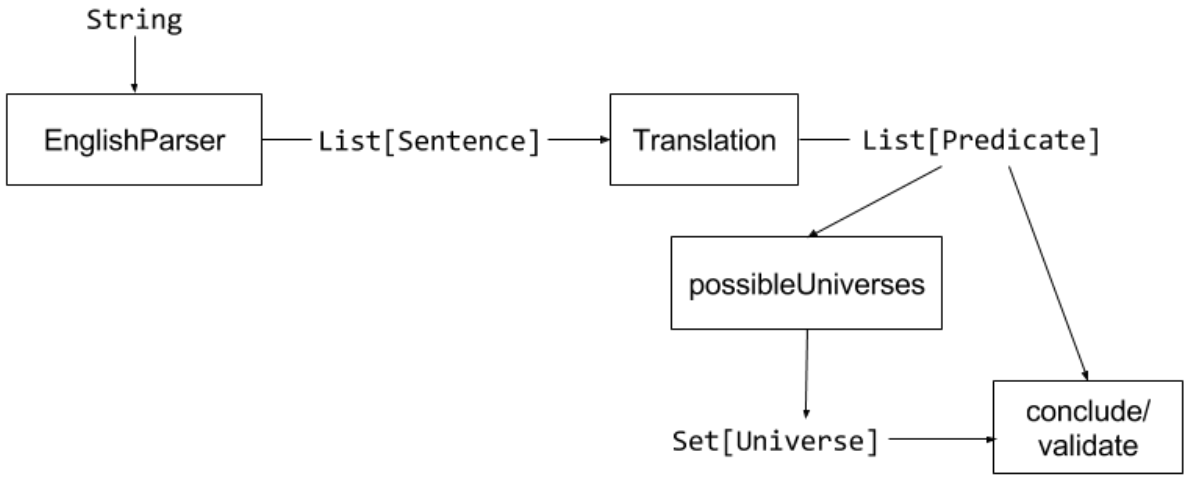
The system is written in Scala and takes a file name as input. To run, you need to first install sbt. The project code is on github at `https://github.com/seveneightn9ne/semantic-parser`. To download and run the code:

```
$ git clone https://github.com/seveneightn9ne/semantic-parser.git
$ cd semantic-parser
$ sbt "run input/multiple.txt"
```

This will run with the input coming from the file `input/multiple.txt`. There are many other example inputs there that you can try, too.

An input file should contain a sequence of sentences in the present tense. If a sentence starts with "Therefore," it will be validated as an attempted conclusion. If there are no sentences that start with "Therefore," it will try to generate some conclusions for you. There is no support (yet) for prepositional phrases or adjectives.

1.2 System Diagram



This diagram shows what happens in the easy case. The `EnglishParser` returns only one unambiguous list of `Sentence`s, and they are all able to be translated into `Predicate`s. The `Predicate`s are used to generate possible universes, which are then used to evaluate the conclusions (either made up conclusions, or ones provided in the input).

Often, there will be multiple possible parses for the `Sentence`s. In this case, we check if all interpretations are translatable (and that they translate to different `Predicate`s). If they do, we have an unrecoverable ambiguity, and it will print the ambiguous `Sentence` trees and `Predicate`s to the console, and give up on conclusions.

Sometimes, there are too many entities and relations (sets) introduced by the predicates, so that the number of possible universes becomes very large. The system only does conclusion/-validation by searching for counterexamples, so it generates all possible universes, and filters for ones in which the predicates are true. When the number of universes is more than 2^{22}, the system does not attempt to generate them, and fails with a message to the console.

1.3 Code Layout

All the Scala code is under `src/main/scala/`. The `XP` and `Xbar` datatypes representing English sentences are all in the `EnglishStructure/` folder. The `Predicate` datatype representing logical predicates is in `PredicateCalculus/Predicates.scala`.

2 X-bar Sentence Representation

`src/main/scala/EnglishStructure/*`

The system uses a simplified version of X-bar syntax. A `Sentence` is actually just a `VP`, so there is no `TP` or `CP`. There is also no notion of movement. This restricts the parseable sentences by a lot, but allows for a simpler and more readable grammar. It would not actually be difficult to expand the system to use tense, but it might not be very useful.

Another design decision was to make the noun the head of a determiner-noun phrase, instead of the determiner, because this seemed the most straightforward to me when I started. However, when it came to translation, it turns out that the logical translation of a determiner-noun phrase is basically entirely determined by the determiner (consider "every boy ..." which translates to "$(\forall x)(Bx \rightarrow ...)$", versus "some boy ..." "$(\exists x)(Bx \wedge ...)$". So it probably would have been better to switch to determiner-headed phrases here.

2.1 XP

`src/main/scala/EnglishStructure/XP.scala`

There is a type `XP`, which is parameterized on the types of the Specifier, Head, Complement, and Adjunct. So for example a `VP` is an `XP[Noun, Verb, Noun, Adverb]`, because its spec and complement are `NP`s, and its adjunct type is an `AdvP`. The implemented `XP` subtypes are:

- `VP` extends `XP[Noun, Verb, Noun, Adverb]`
- `NP` extends `XP[Determiner, Noun, Verb, Preposition]`
- `DP` extends `XP[Noun, Determiner, Word, Word]`
- `PP` extends `XP[Word, Preposition, Noun, Word]`
- `AdvP` extends `XP[Word, Adverb, Word, Adverb]`

In the parameterizations, the use of `Word` means that the phrase cannot have that thing. For example, PPs do not have specifiers or adjuncts. Each `XP` contains an optional specifier `XP`, and a head, which is either an `Xbar` or a conjunction of two `XP`s.

Although the definition for `PP`s was written, the system does not use them (the parser does not know how to parse prepositional phrases).

2.2 Xbar

`src/main/scala/EnglishStructure/XP.scala`

The `Xbar` types are analogous to the `XP` types. They contain a head which is either an `Xbar` or a `Word`. They have an optional complement `XP` and an optional adjunct `XP`. While a single `Xbar` cannot have both a complement and an adjunct, this is enforced at runtime as opposed to being enforced by the type system.

2.3 Words

`src/main/scala/EnglishStructure/*`

There are two types of words: open class, and closed class words. Closed class words, like determiners, are implemented as an `Enumeration` of possible values. Open class words such as nouns can take any `String` as their value, and they have some simple morphology attached, such as a plural marker. Verbs have a marker for plural agreement with the subject.

The advantage of the open class/ closed class approach is that the system can use any names or made up words in sentences that are still understandable as English. A disadvantage is that it cannot handle any special case rules, such as "men" being the plural for "man". If you want it to recognize that those words represent the same set of entities, then you have to write "men" as "mans" instead. While it would be possible to include special cases such as these in the grammar, I decided for the simplicity and readability of the grammar to not handle these cases.

As mentioned above, there is no tense on verbs, they are always interpreted as present tense.

2.4 Conjunctions

`src/main/scala/EnglishStructure/XP.scala`

Conjunctions seem to be the unfortunate outliers of X-bar theory. I implemented them as a `ConjP` phrase parameterized on the type of `XP` phrase it contains (and is contained by). `ConjP`s have an optional `Preconj` (such as "either," "both"), two `XP` subphrases, and a conjunction word ("and" or "or"). So an `XP` is headed by either an `Xbar` or a `ConjP[XP]`. There is a runtime restriction that an `XP` headed by a `ConjP` cannot have a spec, but this isn't restricted by the type system.

Also, every `XP` type has to allow conjunctions (this is possible to get around, but would make messier code), so you could construct a `Sentence` such as "Either the or a boy eats cake" (Where the determiner is a conjunction).

3 Predicate Calculus Representation

`src/main/scala/PredicateCalculus/Predicates.scala`

3.1 Entities

There are two subtypes of `Entity`: `EntityConstant` and `EntityVariable`. `EntityConstant` can represent either an individual constant, such as "j" representing the entity John, or a hypothetical such as "hypothetical_a" which are generated when creating universes under which to evaluate predicates, so that a premise set with no constants can still be evaluated. `EntityVariable`s represent the variables bound by `Existential`s and `Universal`s.

3.2 Relations

Within `Relation`s, there are unary and binary relations. Within unary relations, there are what I called is-a-relations, that represent nouns, and does-relations, which represent verbs. The `Relation`s themselves do not hold any entities in them; `Universe`s assert relationships between `Relation`s and `Entity`s.

There is no identity relation, so for example you couldn't represent the sentence "Marc is the only man who eats cake." That would be an infinite relation, which has a true/false assertion for every possible pair of entities. All the relations in this system are finite, so this limits the system from being "full" predicate calculus and therefore avoids the problems associated with Godel's Incompleteness Theorem and allows me to be able to deterministically validate proofs.

3.3 Predicates

`Predicate` is a recursive datatype with the following subtypes:

- `Atom`, which takes an `Entity` and `UnaryRelation`, as in the predicate "Wa" which is the translation of the sentence "Adam walks."
- `BinaryAtom`, which takes an `Entity` and `BinaryRelation`, as in the predicate "Lab" which is the translation of the sentence "Adam loves Bob."
- `Negation`, which takes a `Predicate`, as in "¬Wa" which is the translation of the sentence "Adam doesn't walk."
- `Conjunction`, which takes two `Predicate`s, as in "(Mj ∧ Fj)" which is the translation of the sentence "John is both a man and a farmer."

- `Disjunction`, which takes two `Predicate`s, as in "(Mj $\vee$ Fj)" which is the translation of the sentence "John is either a man or a farmer."
- `Existential`, which takes an `EntityVariable` and a `Predicate`. It asserts that there is some entity which satisfies the predicate, as in "($\exists$x)Mx" which is the translation of the sentence "There is a man."
- `Universal`, which takes an `EntityVariable` and a `Predicate`. It asserts that all entities satisfy the predicate, as in "($\forall$x)(Mx $\vee$ Wx)" which is the translation of the sentence "Everyone is a man or a woman."
- `Conditional`, which takes a `Predicate` antecedent and a `Predicate` consequent, as in "($\forall$x)(Bx $\rightarrow$ $\neg$Gx)" which is the translation of the sentence "No boys are girls."
- `Biconditional`, which takes two `Predicate`s, as in "($\forall$x)(Vx $\leftrightarrow$ Fx)" which would be the translation of the sentence "All vegetarians are feminists, and vice versa," though there are no translation rules that create biconditionals.
- `Conclusion`, which takes a `Predicate` ϕ and represents "$\therefore \phi$". Its only purpose is to represent that the predicate is an attempted conclusion, and it is the translation of sentences that start with "therefore."

`Predicate`s can be evalutated given a `Universe`, returning `true` or `false`. They can also be translated back to English, but this is done in a simple recursive manner directly to text, instead of using translation rules.

3.4 Naming

`src/main/scala/PredicateCalculus/UniqueDesignations.scala`

When a new relation or entity is introduced, it needs an appropriate name generated. `UniqueDesignations` provides a global namespace, so you can request a name to be made from a string and it will make the best name that hasn't been used yet. For the variable names in existentials and universals, you pass in a `Context` which will include any variables that are already bound. That way, different sentences can reuse the same variable (there's no global context for variables).

4 Universes

`src/main/scala/PredicateCalculus/Universe.scala`

A `Universe` (also known as an interpretation) represents a complete description of the truth value of atomic predicates with a given set of entities and relations. So, for example if the only relation is R and the only entity is e, then the universe would say that "Re" is either true or false.

5 CFG Parser

`src/main/scala/EnglishParser.scala`

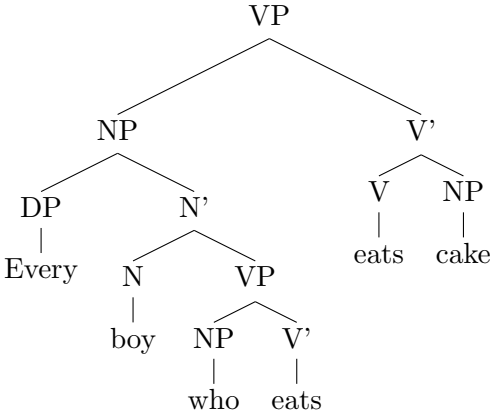
Scala provides a native parser-combinator library, but I instead used an implementation of the GLL[1] algorithm. It can handle even left-recursive grammars in worst-case O(n^3) time. More importantly, it parses nondeterministically, and so allows multiple parses as results. This is essential for being able to parse a natural language, and allows the grammar itself to be written in a very natural way, as opposed to having to manually resolve left-recursion in the grammar.

The grammar itself has explicit rules matching the closed class words, such as determiners, existential "there", and "therefore". The open-class words like nouns are restricted to not be any of the closed class words, which helps in restricting the valid parses by a lot.

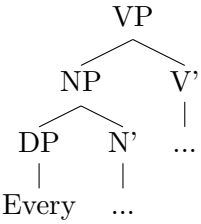
6 Sentence Translation

`src/main/scala/Translation.scala`

The `translate` function pattern matches on various `XP` and `Xbar` structures that it recognizes, and turns them into `Predicate`s recursively. For example, the sentence "Every boy who eats, eats cake.":



This sentence would match the rule that recognizes:



So it would translate the sentence as "(∀x)(`translate(nbar,x)` → `translate(vbar,x)`)", recursively finding the translations for the N' and V', and passing down the variable `x` as a `SubjectContext` object.

7 Generating Conclusions

`src/main/scala/Main.scala`
`src/main/scala/PredicateCalculus/Conclusions.scala`

To generate conclusions, there are five steps.

1. **Generate all possible universes.** To do this we first get all of the `EntityConstant`s, `UnaryRelation`s, and `BinaryRelation`s in the predicates, plus another two hypothetical `EntityConstant`s. The two hypotheticals are so that any universal conclusions are actually universal, and not just things that happened to be true for the given entities. For example, if the only given predicate is "(∃x)Wx", then there would be no entities, so you couldn't find any universe in which the premise is true. But if you added only one hypothetical entity, then you restricted to universes where the prior is true, then you would conclude that "(∀x)Wx", which is not true. Therefore, we add two hypothetical entities, which avoids this problem.

   Unfortunately, generating all possible universes gets really huge really fast. The number of universes is $2^{eu} * 2^{e^2b} = 2^{e^3ub}$, where e is the number of entities (including the 2

hypotheticals), u is the number of unary relations, and b is the number of binary relations. Whenever this number is larger than 2^{22} the program terminates immediately instead of trying to compute, because on my computer the JVM runs out of memory and crashes with 2^{22} universes. I tried to improve this by incrementally generating universes, so a partial universe could answer "I don't know" to an assertion, but it was very complicated and didn't seem to improve the runtime by enough to be worth it.

2. **Filter universes for ones in which all the predicates are true.** This is just straightforward filtering.

3. **Generate a lot of conclusions.** I generate all possible atoms and their negations, universal atoms, existential atoms, universal conditionals on atoms (of the form "$(\forall x)(\phi x \rightarrow \rho x)$"), and existential conjunctions ("$(\exists x)(\phi x \wedge \rho x)$"). Any time I wish the system concluded something, I add the form of that conclusion to this set of generated conclusions.

4. **Filter for conclusions which are true under all possible universes.** Once again, straightforward filtering.

5. **Filter for conclusions that are "interesting."** This means I take out conclusions that are true given only one of the premises, conclusions that are logically equivalent to each other, and conclusions that entail each other, keeping the most specific conclusion. For example, I would discard "$(\exists x)Ex$" (There is someone who eats) in favor of "Ej" (John eats).

It would also be possible to generate conclusions by performing the proof procedure rules given in 24.241 (Logic). Rules like universal specification, quantifier exchange ($\neg(\forall x)\phi = (\exists x)\neg\phi$), and tautological consequence. Perhaps this can be considered future work.

8 Validating Conclusions

`src/main/scala/Main.scala`
`src/main/scala/PredicateCalculus/Conclusions.scala`

Validating conclusions is very similar procedurally to the steps of generating conclusions.

1. **Generate all possible universes.** This is the same as steps 1-2 above.

2. **Assert that each conclusion is true in all possible universes.** If there exists a universe in which all the priors are true but a conclusion is false, then that universe will be presented as a counterexample, and the conclusion is invalid. Otherwise, the conclusion is valid.

9 System Evaluation

The system works well for small inputs in the forms that I have tested. The translation from `Sentence`s to `Predicate`s is quite specific and not super generalizable, so novel inputs have a good chance of tripping it up. Part of this I think is due to using noun headed phrases instead of determiner headed phrases, so you have to look deeper into the structure of a phrase to know what to do with it.

The amount of possible universes is the other main weakness of the system. I tried a lot of various ways of generating fewer universes, and some helped a lot (for example, you only need two hypothetical entities total, as opposed to two per universal or existential predicate). But overall the time it takes is still a hugely limiting factor.

References

[1] Elizabeth Scott and Adrian Johnstone. *GLL Parsing.* `http://dotat.at/tmp/gll.pdf`, 2010.