PROLOG

Parsing and Semantics in DCGs

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 11

01/11/04

Definite Clause Grammars Recap

- We can use the --> DCG operator in Prolog to define grammars for any language.
- The grammar rules consist of *non-terminal symbols* (e.g. NP, VP) which define the structure of the language and *terminal symbols* (e.g. Noun, Verb) which are the words in our language.
- The Prolog interpreter converts the DCG notation into conventional Prolog code using *difference lists*.
- We can add *arguments* to non-terminal symbols in our grammar for any reason (e.g. number agreement).
- We can also add pure Prolog code to the right-hand side of a DCG rule by enclosing it in { }.

DCG Recognisers

- We can write simple grammars using the DCG notation that *recognise* if a string of words (represented as a list of atoms) belongs to the language.

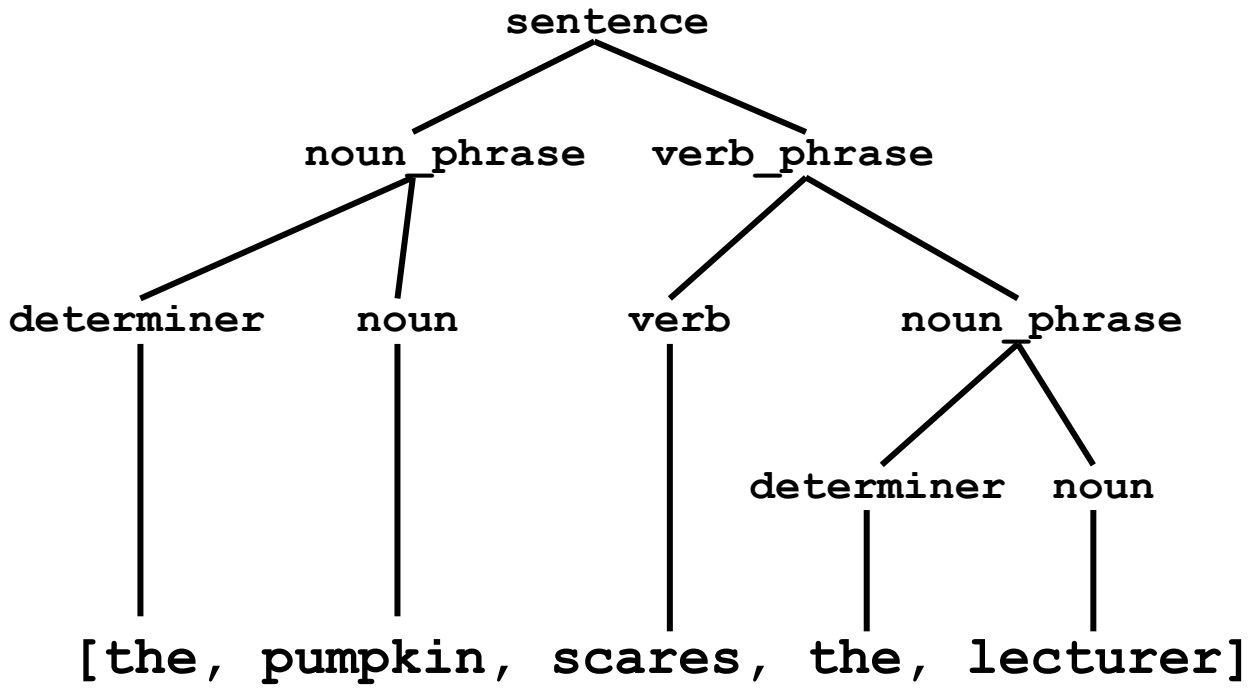
```
sentence --> noun, verb_phrase.
verb_phrase --> verb, noun.
noun --> [bob].
noun --> [david].
noun --> [annie].
verb --> [likes].
verb --> [hates].
verb --> [runs].

|?- sentence([annie, hates, david],[]).
yes
```

- However, this is of limited usefulness. Ideally we would like to *interpret* the input in some way: to understand it, parse it, or convert it into some other more useful form.

DCG: Parsers

- A parser *represents* a string as some kind of structure that can be used to understand the role of each of its elements.
- A common representation is a *parse tree* which shows how input breaks down into its grammatical constituents.



Two parsing techniques

- There are generally two ways of using DCGs to build a structural representation of the input.

1. Computing the structure once the constituents of the input have been identified.
   - Partial results can be passed via extra arguments in non-terminal symbols and computed to create a suitably representative result.
   - For example, we might want our DCG to represent a number expressed as a string as an integer.

```
number(N) --> digit(D), [hundred], {N is (D * 100)}.
digit(1) --> [one].

|?- number(X, [one, hundred], []).
X = 100?
yes
```

- This is only good for summary representations; it doesn't tell us anything about the internal structure of our input.

Two parsing techniques (2)

2. The more popular method is to use *unification* to identify the grammatical role of each word and show how they combine into larger grammatical structures.
   - This creates a representation similar to a parse tree.

```
sentence(s(NP,VP)) -->
    noun_phrase(NP), verb_phrase(VP).
```

- Which can be read as:

  The parsed structure of a **sentence** must be **s(NP,VP),**

  where **NP** is the parsed structure of the **noun phrase**, and

  VP is the parsed structure of the **verb phrase**.

- The rules for NPs and VPs would then need to be augmented so that they also represent a parse of their constituents in the head of the rule.

Example: parsing English

- So lets take a small grammar which defines a tiny fragment of the English language and add arguments so that it can produce a parse of the input.

Original grammar rules:

```
sentence --> noun_phrase(Num), verb_phrase(Num).

noun_phrase(Num) --> determiner(Num), noun_phrase2(Num).
noun_phrase(Num) --> noun_phrase2(Num).

noun_phrase2(Num) --> adjective, noun_phrase2(Num).
noun_phrase2(Num) --> noun(Num).

verb_phrase(Num) --> verb(Num).
verb_phrase(Num) --> verb(Num), noun_phrase(_).
```

- Note the use of an argument to enforce number agreement between noun phrases and verb phrases.

Example: parsing English (2)

- Now we can add a new argument to each non-terminal to represent its structure.

```
sentence(s(NP,VP)) -->
     noun_phrase(NP,Num), verb_phrase(VP,Num).

noun_phrase(np(DET, NP2), Num) -->
     determiner(DET, Num), noun_phrase2(NP2, Num).

noun_phrase(np(NP2), Num) -->
     noun_phrase2(NP2, Num).

noun_phrase2(np2(N), Num) --> noun(N, Num).

noun_phrase2(np2(ADJ, NP2), Num) -->
     adjective(ADJ), noun_phrase2(NP2, Num).

verb_phrase(vp(V), Num) --> verb(V, Num).

verb_phrase(vp(V, NP), Num) -->
     verb(V, Num), noun_phrase(NP, _).
```

Example: parsing English (3)

- We also need to add extra arguments to the terminal symbols i.e. the lexicon.

```
determiner(det(the), _) --> [the].
determiner(det(a), singular) --> [a].

noun(n(pumpkin), singular) --> [pumpkin].
noun(n(pumpkins), plural) --> [pumpkins].
noun(n(lecturer), singular) --> [lecturer].
noun(n(lecturers), plural) --> [lecturers].

adjective(adj(possessed)) --> [possessed].

verb(v(scares), singular) --> [scares].
verb(v(scare), plural) --> [scare].
```

- We represent the terminal symbols as the actual word from the language and its grammatical role. The rest of the grammatical structure is then built around these terminal symbols.

Using the parser.

- Now as a consequence of recognising the input, the grammar constructs a term representing the constituent structure of the sentence.
- This term is the 1st argument of **`sentence/3`** with the 2nd argument the input list and the 3rd the remainder list (usually []).

```
|?-sentence(Struct,[the, pumpkin, scares, the, lecturer],[]).
   Struct = s( np( det(the), np2(n(pumpkin)) ),
               vp( v(scares), np(det(the), np2(n(lecturer))) ) )?
   yes
```

- We can now generate all valid sentences and their structures by making the 2nd argument a variable.

```
|?-sentence(X,Y,[]).
   X = s(np(det(the),np2(adj(possessed),np2(n(lecturer)))),
         vp(v(scares),np(det(the),np2(n(pumpkin))))),
   Y= [the,possessed,lecturer,scares,the,pumpkin]?;
   ……etc
```

Extracting meaning using a DCG

Bratko, I "Prolog: Programming for Artificial intelligence" pg 568

- Representing the structure of a sentence allows us to see the beginnings of semantic relationships between words.
- Ideally we would like to take these relationships and represent them in a way that could be used computationally.
- A common use of meaning extraction is as a natural language interface for a database. The database can then be questioned directly and the question converted into the appropriate internal representation.
- One widely used representation is Logic as it can express subtle semantic distinctions:
  - e.g. "Every man loves a woman." vs. "A man loves every woman."
- Therefore, the logical structures of Prolog can also be used to represent the meaning of a natural language sentence.

Logical Relationships

- Whenever we are programming in Prolog we are representing meaning as logical relationships:
  - e.g. “John paints.” = paints(john).
  - e.g. “John likes Annie” = likes(john,annie).
- It is usually our job to make the conversion between natural language and Prolog but it would be very useful if a DCG could do it for us.
- To do this we need to add Prolog representations of meaning (e.g. paints(john)) to the non-terminal heads of our grammar.
  - Just as we added parse structures to our previous grammar,

    e.g.
    ```
    sentence(s(NP,VP)) --> noun_phrase(NP,Num),
                           verb_phrase(VP,Num).
    ```
  - We can construct predicates that represent the relationship between the terminal symbols of our language:

    e.g. `intrans_verb(Actor,paints(Actor)) --> [paints]`

Adding meaning to a simple grammar

- Here is a simple DCG to recognise these sentences:

```
sentence --> noun_phrase, verb_phrase
noun_phrase --> proper_noun.
verb_phrase --> intrans_verb.
verb_phrase --> trans_verb, noun_phrase.
intrans_verb --> [paints].
trans_verb --> [likes].
proper_noun --> [john].
proper_noun --> [annie].
```

```
| ?- sentence([john,likes,annie],[]).
yes
```

- To encode meaning we first need to represent nouns as atoms. Prolog atoms are existential statements

  e.g. `john` = "There exists an entity 'john' ".

```
proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
```

Adding meaning to a simple grammar (2)

- Now we need to represent the meaning of verbs.
- This is more difficult as their meaning is defined by their context i.e. a noun phrase.
- We can represent this in Prolog as a property with a variable entity. For example, the intransitive verb 'paints' needs an NP as its actor: "Somebody paints" = `paints(Somebody).`
- We now need to ensure that this variable '`Somebody`' is matched with the NP that precedes the VP.
- To do this we need to make the argument of the Prolog term ('`Somebody`') *visible* from outside of the term.
- We do this by adding another argument to the head of the rule.

e.g `intrans_verb(Somebody,paints(Somebody)) --> [paints].`

Adding meaning to a simple grammar (3)

- Now we need to ensure that this variable gets matched to the NP at the sentence level.
- First the variable needs to be passed to the parent VP:

```
verb_phrase(Actor,VP) --> intrans_verb(Actor,VP).
```

- The Actor variable must then be linked to the NP at the sentence level:

```
sentence(VP) --> noun_phrase(Actor), verb_phrase(Actor,VP).
```

- It now relates directly to the meaning derived from the NP.
- The logical structure of the VP is then passed back to the user as an extra argument in `sentence`.
- If the grammar is more complex then the structure returned to the user might be the product of more than just the VP. For example, determiners might be used as existential quantifiers (‘every’, ‘all’, ‘a’) to structure the output (see Bratko).

Adding meaning to a simple grammar (4)

- Lastly, we need to define the transitive verb.
- This needs two arguments, a Subject and an Object.

```
trans_verb(Subject,Object,likes(Subject,Object)) --> [likes].
```

- The `Subject` needs to be bound to the initial NP and the `Object` to the NP that is part of the VP.

```
verb_phrase(Subject,VP) --> trans_verb(Subject,Object,VP),
                            noun_phrase(Object).
```

- This binds the `Subject` to the initial NP at the sentence level as it appears in the right position the `verb_phrase` head.

A Grammar for Extracting Meaning.

- Now we have a grammar that can extract the meaning of a sentence.

```
sentence(VP) --> noun_phrase(Actor), verb_phrase(Actor,VP).

noun_phrase(NP) --> proper_noun(NP).

verb_phrase(Actor,VP) --> intrans_verb(Actor,VP).
verb_phrase(Actor,VP) --> trans_verb(Actor,Y,VP),
                          noun_phrase(Y).

intrans_verb(Actor, paints(Actor)) --> [paints].

trans_verb(X,Y, likes(X,Y)) --> [likes].

proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
```

A Grammar for Extracting Meaning.

- The meaning of specific sentences can be extracted:

```
| ?- sentence(X,[john,likes,annie],[]).
   X = likes(john,annie) ? ;
   no
```

- Or, all possible meanings can be generated:

```
| ?- sentence(X,Y,[]).

X = paints(john),
Y = [john,paints] ? ;

X = likes(john,john),
Y = [john,likes,john] ? ;

X = likes(john,annie),
Y = [john,likes,annie] ? ;

X = paints(annie),
Y = [annie,paints] ? ;

X = likes(annie,john),
Y = [annie,likes,john] ? ;

X = likes(annie,annie),
Y = [annie,likes,annie] ? ;

no
```

Extending the meaning

- By writing grammars that accept
  - conjunctions (e.g. ‘and’),
  - relative clauses (e.g. ‘that snores’ in ‘The man that snores’),
  - and conditionals (e.g. ‘I am blue *if I am a dolphin*’)

  all forms of logical relationship in Prolog can be extracted from strings of natural language.

Next few lectures…..

- I will show you how you can interface directly with the Prolog interpreter using natural language;
- turn these strings into lists of terms,
- manipulate these terms,
- use them to perform computations, and
- create new strings as output.