PROLOG

Definite Clause Grammars

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 10

28/10/04

Contents

- Definite Clause Grammars
- Grammar Rules
- Terminals and non-terminals
- Grammar rules in Prolog
- How Prolog uses grammar rules
- A very simple English grammar
- Adding arguments and Prolog goals
- Explanation of Difference Lists

Definite Clause Grammars

- A *grammar* is a precise definition of which sequences of words or symbols belong to some *language*.
- Grammars are particularly useful for natural language processing: the computational processing of human languages, like English.
- But they can be used to process any precisely defined 'language', such as the commands allowed in some human-computer interface.
- Prolog provides a notational extension called *DCG (definite Clause Grammar)* that allows the direct implementation of formal grammars.

Grammar rules

- In general, a grammar is defined as a collection of *grammar rules.* These are sometimes called *rewrite rules*, since they show how we can rewrite one thing as something else.
- In linguistics, a typical grammar rule for English might look like this:

  sentence → noun_phrase, verb_phrase

  e.g " The man ran."

- This would show that, in English, a *sentence* could be constructed as a *noun phrase*, followed by a *verb phrase*.
- Other rules would then define how a noun phrase, and a verb phrase, might be constructed. For example:

  noun_phrase → noun
  noun_phrase → determiner, noun
  verb_phrase → intransitive_verb
  verb_phrase → transitive_verb, noun_phrase

Terminals and non-terminals

- In these rules, symbols like *sentence*, *noun*, *verb*, etc., are used to show the structure of the language, but they don't go as far down as individual ‘words’ in the language.
- Such symbols are called *non-terminal symbols*, because we can't stop there.
- In defining grammar rules for *noun*, though, we can say:

  noun → [ball]

  noun → [dog]

  noun → [stick]

  noun → [‘Edinburgh’]

- Here, ‘ball’, ‘dog’, ‘stick’ and ‘Edinburgh’ are words in the language itself.
- These are called the *terminal symbols*, because we can't go any further. They can't be expanded any more.

Grammar rules in Prolog

- Prolog allows us to directly implement grammars of this form.
- In place of the → arrow, we have a special operator: -->.
- So, we can write the same rules as:

```
sentence      -->  noun_phrase, verb_phrase.
noun_phrase   -->  noun.
noun_phrase   -->  determiner, noun.
verb_phrase   -->  intransitive_verb.
verb_phrase   -->  transitive_verb, noun_phrase.
```

- Here, each non-terminal symbol is like a predicate with no arguments.
- Terminal symbols are represented as lists containing one atom

```
noun --> [ball].
noun --> [dog].
noun --> [stick].
noun --> ['Edinburgh'].
```

Proper nouns must be written as strings otherwise they are interpreted as variables.

How Prolog uses grammar rules

- Prolog converts DCG rules into an internal representation which makes them conventional Prolog clauses.
  - This can be seen by 'listing' the consulted code.
- Non-terminals are given two extra arguments, so:

```
sentence --> noun_phrase, verb_phrase.
```

becomes:

```
sentence(In, Out) :-
        noun_phrase(In, Temp),
        verb_phrase(Temp, Out).
```

- This means: some sequence of symbols In, can be recognised as a sentence, leaving Out as a remainder, if
  - a noun phrase can be found at the start of In, leaving Temp as a remainder,
  - and a verb phrase can be found at the start of Temp, leaving Out as a remainder.

How Prolog uses grammar rules (2)

- Terminal symbols are represented using the special predicate 'C', which has three arguments. So:

```
noun --> [ball].
```

becomes:

```
noun(In, Out) :-
        'C'(In, ball, Out).
```

- This means: some sequence of symbols In can be recognised as a noun, leaving Out as a remainder, if the atom *ball* can be found at the start of that sequence, leaving Out as a remainder.

- The built-in predicate 'C' is very simply defined:

```
'C'( [Term|List], Term, List ).
```

where it succeeds if its second argument is the head of its first argument, and the third argument is the remainder.

A very simple grammar

- Here's a very simple little grammar, which defines a very small subset of English:

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

- We can now use the grammar to test whether some sequence of symbols *belongs to* the language:

```
| ?- sentence([bob, likes, annie], []).
yes
| ?- sentence([bob, runs], []).
no
```

Need to write an extra rule for intransitive verbs.

A very simple grammar (2)

- By specifying that the remainder is an empty list we can use the grammar to generate all of the possible sentences in the language:

```
| ?- sentence(X, []).
X = [bob,likes,bob] ? ;
X = [bob,likes,david] ? ;
X = [bob,likes,annie] ? ;
X = [bob,hates,bob] ? ;
X = [bob,hates,david] ? ;
:
```

> This is a *recogniser*. It will tell us whether some sequence of symbols is in a language or not.
> This has limited usefulness.

- It would be much more useful if we could do something with the sequence of symbols, such as converting it into some internal form for processing, or translating it into another language.
- We can do this very powerfully with DCGs, by building a *parser*, rather than a recogniser. (next lecture)

PROLOG

Adding Arguments

- We can add our own arguments to the non-terminals in DCG rules, for whatever reasons we choose.
- As an example, we can very simply add *number* agreement (singular or plural) between the subject of an English sentence and the main verb.

```
sentence --> noun(Num), verb_phrase(Num).
verb_phrase(Num) --> verb(Num), noun(_).
noun(singular) --> [bob].
noun(plural) --> [students].
verb(singular) --> [likes].
verb(plural) --> [like].
```

- So now:

```
| ?- sentence([bob, likes, students], []).
yes
| ?- sentence([students, likes, bob], []).
no
```

Adding Prolog goals

- If we need to, we can add Prolog goals to any DCG rule.
- They need to be put inside { } brackets, so that Prolog knows they're to be processed as Prolog, and not as part of the DCG itself.
- Let's say that within some grammar, we wanted to be able to say that some symbol had to be an integer between 1 and 100 inclusive. We *could* write a separate rule for each number:

```
num1to100 --> [1].
num1to100 --> [2].
num1to100 --> [3].
num1to100 --> [4].
...
num1to100 --> [100].
```

- But using a Prolog goal, there's a much easier way:

```
num1to100 --> [X], {integer(X), X >= 1, X =< 100}.
```

Difference Lists

- We call our grammar with a list of *terminal symbols* and an *empty list* as we are checking that the first list conforms to the grammar with nothing left over.
  - sentence([the,man,ran],[]).
- We do this as the Prolog interpreter uses *difference lists* to convert the DCG rules into conventional code.
- The difference list representation is a way of expressing how two lists intersect.
- Any list can be represented as the difference between two lists:

[the,little,blue,man] can be represented as the difference between:

[the,little,blue,man]-[]

[the,little,blue,man,who,swam]-[who,swam]

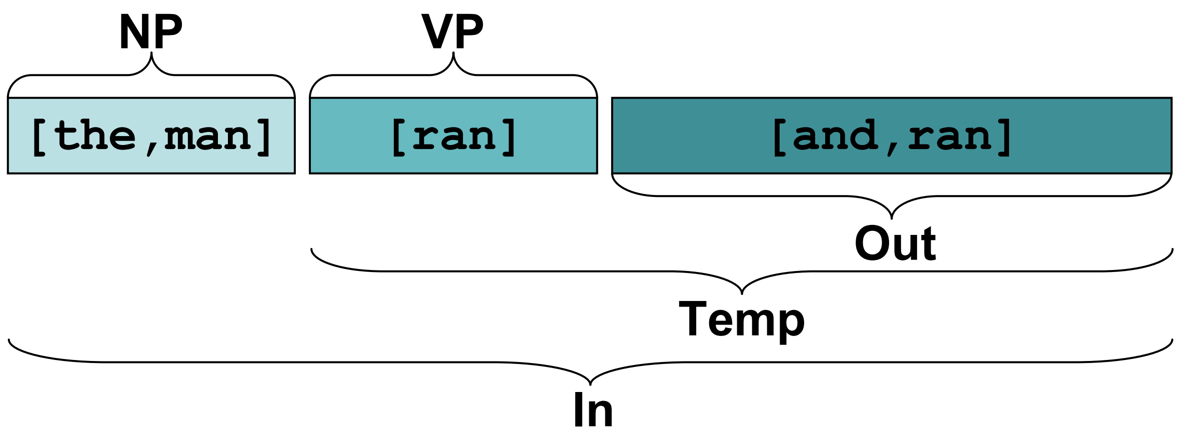
[the,little,blue,man,called,bob]-[called,bob]

Difference Lists (2)

- The Prolog interpreter converts

```
sentence --> noun_phrase, verb_phrase.
```

- into conventional Prolog code using difference lists that can be read as: The difference of lists **In** and **Out** is a sentence if the difference between **In** and **Temp** is a noun phrase and the difference between **Temp** and **Out** is a verb phrase.




```
sentence([the,man,ran,and,ran],[and,ran]):-
        noun_phrase([the,man],[ran,and,ran]),
        verb_phrase([ran],[and,ran]).
```

Diff. Lists: An Efficient Append

```
append([],L2,L2).
append([H|T],L2,[H|Out]):-
     append(T,L2,Out).
```

- append/2 is a highly inefficient way of combining two lists.

```
?- append([a,b,c],[d],X).
    append([b,c],[d],X1)        where X1 = [a|X2]
     append([c],[d],X2)          where X2 = [b|X3]
      append([],[d],X3)           where X3 = [c|X4]
       true.                        where X4 = [d]
```

- It must first recurse through the whole of the first list before adding its elements to the front of the second list.
- As the first list increases in length as does the number of recursions needed.
- If we represent the lists as *difference lists* we can append the second list directly to the end of the first list.

Diff. Lists: An Efficient Append (2)

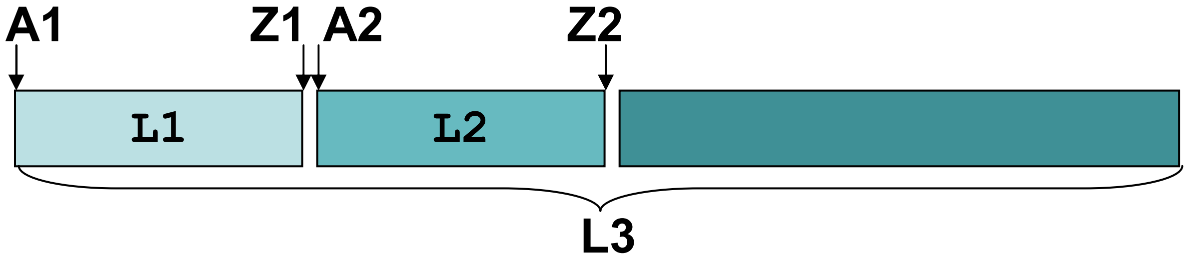
- We can represent any list as the difference between two lists:

  [a,b,c] can be represented as

  [a,b,c]-[] or [a,b,c,d,e]-[d,e] or [a,b,c|T]-T

  Where 'T' can be any list of symbols.
- As the second member of the pair refers to the end of the list it can be directly accessed.
- This allows us to define a version of append that just uses unification to append two lists L1 and L2 to make L3.

```
append(A1-Z1, Z1-Z2, A1-Z2).
```



- When L1 is represented by A1-Z1, and L2 by A2-Z2 the result L3 is A1-Z2 if Z1=A2.

Diff. Lists: An Efficient Append (3)

- If we replace our usual append definition by this one line we can append without recursion.

```
    append(A1-Z1, Z1-Z2, A1-Z2).
```

```
?- append([a,b,c|Z1]-Z1, [d,e|Z2]-Z2, L).
L = [a,b,c,d,e|Z2]-Z2,
Z1 = [d,e|Z2] ?
```

- A clean append can then be achieved by specifying that Z2 is an empty list.

```
| ?- append([a,b,c|Z1]-Z1, [d,e]-[], A1-[]).

1 Call: append([a,b,c|_506]-_506,[d,e]-[],_608-[]) ?
1 Exit: append([a,b,c,d,e]-[d,e],[d,e][],[a,b,c,d,e]-[]) ?

A1 = [a,b,c,d,e],
Z1 = [d,e] ?
yes
```