Definite Clause Grammars

COMP3411/COMP9814 - Artificial Intelligence

Logic Grammars

- A grammar rule is a formal device for defining sets of sequences of symbols.
- Sequence may represent a statement in a programming language.
- Sequence may be a sentence in a natural language such as English.

BNF Notation

- A BNF grammar specification consists of *production rules*.

  <s> ::= a b

  <s> ::= a <s> b

- First rule says that whenever s appears in a string, it can be rewritten with the sequence ab.

- Second rule says that s can be rewritten with a followed by s followed by b.

BNF Notation

- s is a non-terminal symbol.
- a and b are terminal symbols.
- A grammar rule can generate a string, e.g.

$$s \to a\ s\ b$$
$$a\ s\ b \to a\ a\ s\ b\ b$$
$$a\ a\ s\ b\ b \to a\ a\ a\ b\ b\ b$$

Grammar for a robot arm

- Two commands for a robot arm are: *up* and *down*, i.e. move one step up or down respectively.

<move> ::= <step>

<move> ::= <step> <move>

<step> ::= up

<step> ::= down

- The grammar is recursive and has a termination rule.

Definite Clause Grammars

- Prolog has a DCG grammar notation that parses sequences of symbols in a list.

```
s --> [a], [b].
s --> [a], s, [b].

?- s([a, a, b, b], X).
X = []

?- s([a, c, b], X).
false.
```

```
move --> step.
move --> step, move.

step --> [up].
step --> [down].
```

DCGs are translated into Prolog

```
s --> [a], [b].
s --> [a], s, [b].

move --> step.
move --> step, move.

step --> [up].
step --> [down].
```

```
[a, a, b, b]     (2)
[a, b, b]        (2)
[b]              (1)
[]
```

→

```
s([a, b|X], X).
s([a|X], Y) :-
    s(X, [b|Y]).

move(X, Y) :-
    step(X, Y).
move(X, Z) :-
    step(X, Y),
    move(Y, Z).

step([up|X], X).
step([down|X], X).
```

A simple subset of English

sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun_phrase.

determiner --> [a].
determiner --> [the].

noun --> [cat].
noun --> [mouse].

verb --> [scares].
verb --> [hates].

E.g.

the cat scares the mouse

the mouse hates the cat

the mouse scares the mouse

Context Dependence

- Programming languages usually use context free grammars.
  - Type of symbol is determined completely by its position in sentence.
- Natural language is often context dependent.
  - Correctness of one symbol depends on type of other symbols in sentence.
  - E.g. number in English.

Context Dependence

noun --> [cats].

verb --> [hate].

- Adding these rules to the grammar makes the following sentence legal:

the mouse hate the cat.

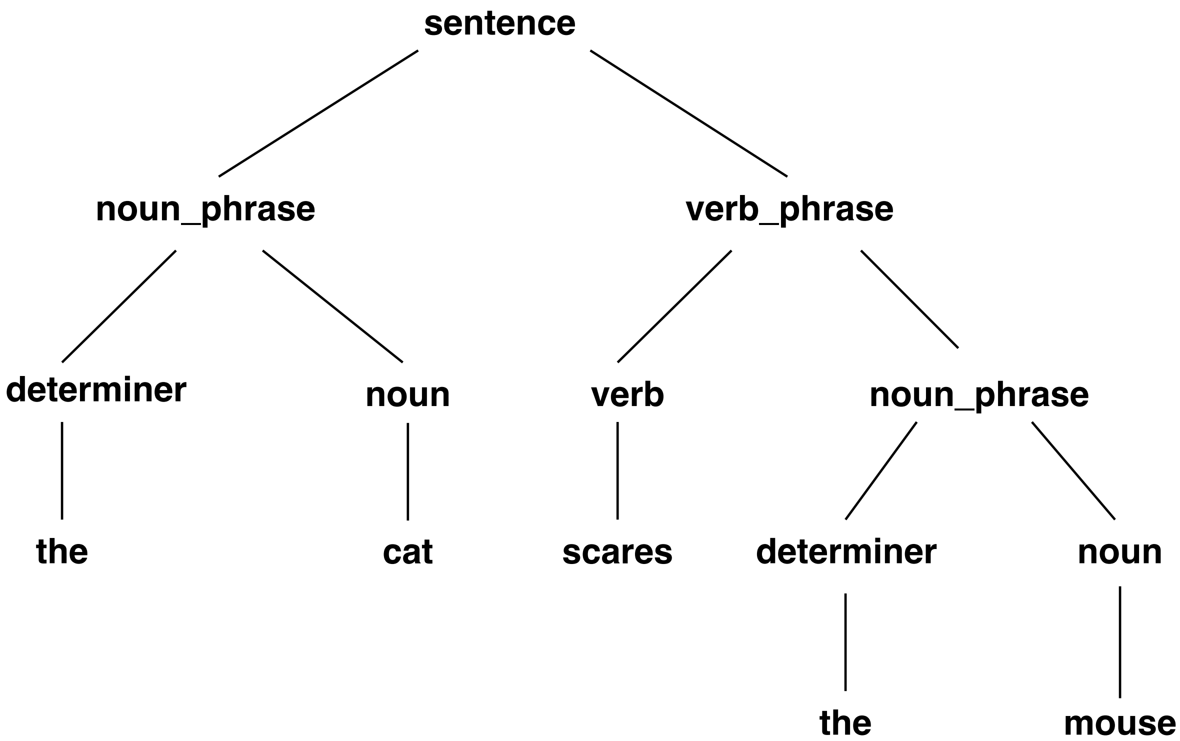
- Additional constraints must be added to the grammar to ensure that the number of all the parts of speech is consistent.

Context Dependence

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

noun_phrase(Number) -->
     determiner(Number),
     noun(Number).

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

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

noun(singular) --> [cat].
noun(singular) --> [mouse].
noun(plural) --> [mice].

verb(singular) --> [hates].
verb(plural) --> [hate].
```

Parse Trees


sentence
noun_phrase
verb_phrase
determiner
noun
verb
noun_phrase
the
cat
scares
determiner
noun
the
mouse

Parse Trees

- Leaves are labelled by the terminal symbols of the grammar
- Internal nodes are labelled by non-terminals
- The parent-child relation is specified by the rules of the grammar.

Parse Trees

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

noun_phrase(Number, noun_phrase(Det, Noun)) -->
        determiner(Number, Det),
        noun(Number, Noun).

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

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

noun(singular, noun(cat)) --> [cat].
noun(plural, noun(cats)) --> [cats].
noun(singular, noun(mouse)) --> [mouse].
noun(plural, noun(mice)) --> [mice].

verb(singular, verb(scares)) --> [scares].
verb(singular, verb(hates)) --> [hates].
verb(plural, verb(hate)) --> [hate].
```

DCG Translation

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

Input list **Remainder**

```
sentence(sentence(A, D), B, F) :-
   noun_phrase(C, A, B, E),
   verb_phrase(C, D, E, F).
```

From parse tree to meaning

```
?- sentence(X, [the, mouse, hates, the, cat], Y).

X = sentence(
        noun_phrase(determiner(the), noun(mouse)),
        verb_phrase(verb(hates),
                    noun_phrase(determiner(the), noun(cat))))

Y = []
```

- In a two step understanding system, the parse tree returned from the grammar rules could be passed to a semantic analyser.

Defining 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(Subject, VP) -->
    trans_verb(Subject, Object, VP),
    noun_phrase(Object).

intrans_verb(Actor, paints(Actor)) --> [paints].
trans_verb(Subject, Object, likes(Subject, Object)) --> [likes].

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

Defining the meaning of a sentence

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

C|>sentence(_0)
C||>noun_phrase(_1)
C|||>proper_noun(_1)
E|||<proper_noun(john)
E||<noun_phrase(john)
C||>verb_phrase(john, _0)
C|||>intrans_verb(john, _0)
E|||<intrans_verb(john, paints(john))
E||<verb_phrase(john, paints(john))
E|<sentence(paints(john))

X = paints(john)
Y = []
```

Defining the meaning of a sentence

```
?- sentence(X, [john, likes, annie], _).

C|>sentence(_0)
C||>noun_phrase(_1)
C|||>proper_noun(_1)
E|||<proper_noun(john)
E||<noun_phrase(john)
C||>verb_phrase(john, _0)
C|||>intrans_verb(john, _0)
R||>verb_phrase(john, _0)
C|||>trans_verb(john, _9, _0)
E|||<trans_verb(john, _9, likes(john, _9))
C|||>noun_phrase(_9)
C||||>proper_noun(john)
R||||>proper_noun(annie)
E||||<proper_noun(annie)
E|||<noun_phrase(annie)
E||<verb_phrase(john, likes(john, annie))
E|<sentence(likes(john, annie))

X = likes(john, annie)
```

The Determiner 'a'

- 'A person paints' does *not* mean *paints(person)*.
- In this sentence 'person' is not a specific person. The correct meaning should be:

  exists(X, person(X) & paints(X))

- The general form for dealing with 'a'

  exists(X, person(X) & Assertion)

The determiner 'every'

E.g.

Every student studies

all(X, student(X) -> studies(X))

'every' indicates the presence of a *universally* quantified variable.

Relative Clauses

E.g.

Every person that paints admires Monet

Can be expressed in a logical form as:

For all X, if X is a person and X paints then X admires Monet.

in Prolog:

```
all(X, person(X) & paints(X) -> admires(X, monet)
```

in general:

```
all(X, Property1 & Property2 -> Assertion)
```

The Complete Grammar

```
?- op(700, xfy, &).
?- op(800, xfy, ->).

determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].
determiner(X, Property, Assertion, exists(X, (Property & Assertion))) --> [a].

noun(X, man(X)) --> [man].
noun(X, woman(X)) --> [woman].
noun(X, person(X)) --> [person].

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

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

intrans_verb(X, paints(X)) --> [paints].
```

The Complete Grammar

```
sentence(S) -->
      noun_phrase(X, Assertion, S),
      verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
      determiner(X, Property12, Assertion, S),
      noun(X, Property1),
      rel_clause(X, Property1, Property12).
noun_phrase(X, Assertion, Assertion) -->
      proper_noun(X).

verb_phrase(X, Assertion) -->
      trans_verb(X, Y, Assertion1),
      noun_phrase(Y, Assertion1, Assertion).
verb_phrase(X, Assertion) -->
      intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
      [that],
      verb_phrase(X, Property2).
rel_clause(_, Property, Property).
```

The Complete Grammar

```
determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].
```

```
sentence(S) -->
      noun_phrase(X, Assertion, S),
      verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
      determiner(X, Property12, Assertion, S),
      noun(X, Property1),
      rel_clause(X, Property1, Property12).
noun_phrase(X, Assertion, Assertion) -->
      proper_noun(X).

verb_phrase(X, Assertion) -->
      trans_verb(X, Y, Assertion1),
      noun_phrase(Y, Assertion1, Assertion).
verb_phrase(X, Assertion) -->
      intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
      [that],
      verb_phrase(X, Property2).
rel_clause(_, Property, Property).
```

The Complete Grammar

```
sentence(S) -->
      noun_phrase(X, Assertion, S),
      verb_phrase(X, Assertion).

noun_phrase(X, Assertion, S) -->
      determiner(X, Property12, Assertion, S),
      noun(X, Property1),
      rel_clause(X, Property1, Property12).
noun_phrase(X, Assertion, Assertion) -->
      proper_noun(X).

verb_phrase(X, Assertion) -->
      trans_verb(X, Y, Assertion1),
      noun_phrase(Y, Assertion1, Assertion).
verb_phrase(X, Assertion) -->
      intrans_verb(X, Assertion).

rel_clause(X, Property1, (Property1 & Property2)) -->
      [that],
      verb_phrase(X, Property2).
rel_clause(_, Property, Property).
```

Variable Bindings

```
noun_phrase(X, Assertion, S) -->
      determiner(X, Property12, Assertion, S),
      noun(X, Property1),
      rel_clause(X, Property1, Property12).


determiner(X, Property, Assertion, all(X, (Property -> Assertion))) --> [every].
```

Result

```
?- sentence(X, [every,person,that,paints,admires,monet],_).

X = all(_24512, person(_24512)& paints(_24512)->admires(_24512, monet))
```

Annotations

```
noun_phrase(NP) --> proper_noun(NP), {asserta(history(NP))}.
```

- Annotations allow you to write any Prolog code you like to support the processing of the grammar.

- Anything in between ‘{‘ and ‘}’
- `Asserta` stores a clause in Prolog’s data base.
  - The new clause becomes the *first* in the database

Assert and Asserta add clauses to the database

```
?- assert(f(a)).
?- assert(f(b)).

?- listing(f/1).

f(a).
f(b).
```

```
?- asserta(f(a)).
?- asserta(f(b)).

?- listing(f/1).

f(b).
f(a).
```

Retract deletes clauses

```
?- assert(f(a)).
?- assert(f(b)).

?- listing(f/1).

f(a).
f(b).
```

```
?- retract(f(X)).
X = a ;
X = b.

?- listing(f/1).
:- dynamic f/1.

true.
```

Frames

- Objects are represented by a list of properties and values

- E.g.
  - object(shirt, [colour(green)]).
  - event(1, [
    actor(john),
    action(buy),
    object(shirt, [colour(green)]),
    location(myers)
    ])

- “Understanding” a sentence means filling in the slots.

Resolving References

- Use Prolog annotations in grammar to build frame’s property list.
- *new_event* asserts events in reverse order.
- Database events can be used to resolve references

Resolving References

Suppose history is:

```
history(object(John,  [isa(person), gender(masculine), number(singular)])).
history(object(annie, [isa(person), gender(feminine),  number(singular)])).
```

Simple example of pronoun resolution:

```
pronoun(Resolvent) --> [he],
    {resolve([gender(masculine), number(singular)], Resolvent)}.
pronoun(Resolvent) --> [she],
    {resolve([gender(feminine), number(singular)], Resolvent)}.
```

```
resolve(Properties, Name) :-
    history(object(Name, Props)),
    subset(Properties, Props).
```