Prolog

… grammar & parsing

Ref: Learn Prolog Now! On-line Prolog Documentation.

Blackburn, Bos, Striegnitz.

Review of Prolog 2

- Recursive programming techniques - Accumulator example

```
reverse(Xs, Ys)             :- reverse(Xs, [ ], Ys).
reverse([X|Xs], Acc, Ys)    :- reverse(Xs, [X|Acc], Ys).
reverse([ ], Ys, Ys).
```

NB: 2 reverse
reverse/2
reverse/3

- Recursive programming techniques – Insert example

```
insert([],       It, [It]).
insert([H|T], It, [It, H|T])        :- H @> It.
insert([H|T], It, [H|NewT])         :- H @< It, insert(T, It, NewT).
```

- Note
  - **Identify lists** - Empty list **[ ]** and non-empty list **[H|T]**
  - **Identify** "**in**" and "**out**" parameters
  - **Accumulator** – the **result** is constructed during **recursive descent**
  - **Insert** – the **result** is constructed during **recursive ascent**

Review of Prolog 2

```
reverse(Xs, Ys)              :- reverse(Xs, [ ], Ys).
reverse([X|Xs], Acc, Ys)     :- reverse(Xs, [X|Acc], Ys).
reverse([ ], Ys, Ys).
```

NB: 2 reverse
reverse/2
reverse/3

```
insert([ ],      It, [It]).
insert([H|T], It, [It, H|T])        :- H @> It.
insert([H|T], It, [H|NewT])         :- H @< It, insert(T, It, NewT).
```

```
append([ ], L, L).
append([H|T], L2, [H|L3])  :-  append(T, L2 , L3).
```

- Note
  - Where the **empty list** occurs and how the **result** is constructed
  - How the result is constructed in the **recursive case**
  - For **reverse**, **TWO PREDICATES** have been defined

Parsing using Prolog

- In this presentation we look at parsing using Prolog
- We have already discussed Context Free Grammars (CFGs)
- In Prolog, there is a special notation to express the rules from CFGs and for implementing parsers in Prolog
- **This notation is called a Definite Clause Grammar (DCG)**
- The notation allows the grammar production rules (P) to be written in Prolog almost verbatim
- **Left recursion must be removed and replaced by tail (right) recursion – this we already did in the C Parser**
- In this presentation we will also look at **the relationship between the DCG and the corresponding Prolog code**
- The DCG is in effect a “syntactic sugar” wrapper!

Parsing using Prolog

- Example grammar in DCG notation

/* rules with non-terminal symbols */

sentence → noun_phrase, verb_phrase.
noun_phrase → determiner, noun.
verb_phrase → verb.
verb_phrase → verb, noun_phrase.

/* rules with terminal symbols */

determiner → [the].
noun → [man].
noun → [apple].
verb → [sings].
verb → [eats].

Parsing DCG → Prolog

Prolog converts the above DCG form to:-

// non-terminal symbols (NT)

```
sentence(A, C)      :- noun_phrase(A, B),  verb_phrase(B, C).
noun_phrase(A, C)   :- determiner(A, B),   noun(B, C).
verb_phrase(A, B)   :- verb(A, B).
verb_phrase(A, C)   :- verb(A, B),  noun_phrase(B, C).
```

// terminal symbols (T)

```
determiner( [the | A],  A).
noun( [man   |A], A).
noun( [apple |A], A).
verb(  [eats   |A], A).
verb(  [sings  |A], A).
```

Note how results are "**passed**"
Note for the terminals that this corresponds to **match** in the C parser. I.e. the input is a **list** and the **Terminal** is matched against the **head** of the list.
Success ➔ **tail** is "**returned**".

Parsing using Prolog

- Testing the grammar: is a sentence syntactically correct or not

```
?- sentence( [the, man, sings], []).
true.
?- sentence( [the, man, reads], []).
false.
?- sentence( [the, man, eats, the, apple], []).
true.
```

- **Note**: that the goal is to process all the elements of the sentence hence the expected result will be [] – the empty list

Parsing using Prolog

- **This also give us the possibility of <u>inspecting</u> the grammar**

```
?- verb(X, []).
X = [eats] ;
X = [sings].
?- noun(X, []).
X = [man] ;
X = [apple].
?- noun_phrase(X, []).
X = [the, man] ;
X = [the, apple].
```

```
?- verb_phrase(X, []).
X = [eats] ;
X = [sings];
X = [eats, the, man] ;
X = [eats, the, apple];
X = [sings, the, man] ;
X = [sings, the, apple].
```

➔ **12 possible sentences**

Parsing using Prolog

- Recall that **relations** work in **BOTH directions** hence
- To see all possible sentences in this grammar

```
language :- findall(X, sentence(X, []), L), display(L).
```

```
?- language.
[the, man, eats]
[the, man, sings]
[the, man, eats, the, apple]
[the, man, eats, the, man]
[the, man, sings, the, apple]
[the, man, sings, the, man]
[the, apple, eats]
[the, apple, sings]
[the, apple, eats, the, apple]
[the, apple, eats, the, man]
[the, apple, sings, the, apple]
[the, apple, sings, the, man]
```

Parsing using Prolog

This allows us to test the grammar STEPWISE during the development

```
?- noun([man], []).
true.
?- noun([book], []).
false.
?- verb([eats], []).
true.
?- verb([reads], []).
false.
?- verb_phrase([reads, the, book], []).
false.
?- verb_phrase([eats, the, apple], []).
true.
```

Parsing using Prolog

What does Prolog generate from this grammar?

```
/* non-terminal symbols – RULES */
sentence(A, C)          :- noun_phrase(A, B), verb_phrase(B, C).
noun_phrase(A, C)       :- determiner(A, B), noun(B, C).
verb_phrase(A, B)       :- verb(A, B).
verb_phrase(A, C)       :- verb(A, B), noun_phrase(B, C).
/* terminal symbols – FACTS */
determiner( [the | A], A ).
noun( [man   | A],     A ).
noun( [apple | A],     A ).
verb(  [eats   | A] ,  A ).
verb(  [sings  | A],   A ).
```

Compare this with the C parser using match – Prolog matches the head of the list (see the terminal definitions). So in sentence(A, C), C will be []. I.e. all "tokens" have been matched by the parser. The token stream is the list A.

Prolog – call/exit/fail/redo

- These can be viewed as

- **CALL → EXIT** means a predicate has succeeded
- **CALL → FAIL** means a predicate has failed
- **REDO:** repeat until all possibilities have been found
- if more rules exist try these in turn until the process **FAILs**

- CALL / EXIT is “similar” to procedural programming
- REDO / FAIL is unique to Prolog

Parsing using Prolog

How can we check this using Prolog? **Answer – use trace**

```
?- trace.
?- sentence([the, man, sings], []).
Call:  (6) sentence([the, man, sings], [])  ?
Call:  (7) noun_phrase([the, man, sings], _G430) ?
Call:  (8) determiner([the, man, sings], _G430) ?
Exit:  (8) determiner([the, man, sings], [man, sings])
Call:  (8) noun([man, sings], _G430) ?
Exit:  (8) noun([man, sings], [sings]) ?
Exit:  (7) noun_phrase([the, man, sings], [sings]) ?
Call:  (7) verb_phrase([sings], []) ?
Call:  (8) verb([sings], []) ?
Exit:  (8) verb([sings], []) ?
Exit:  (7) verb_phrase([sings], []) ?
Exit:  (6) sentence([the, man, sings], [])  ?
true .
?- notrace, nodebug.
```

Parsing using Prolog

The parse may be thought of as

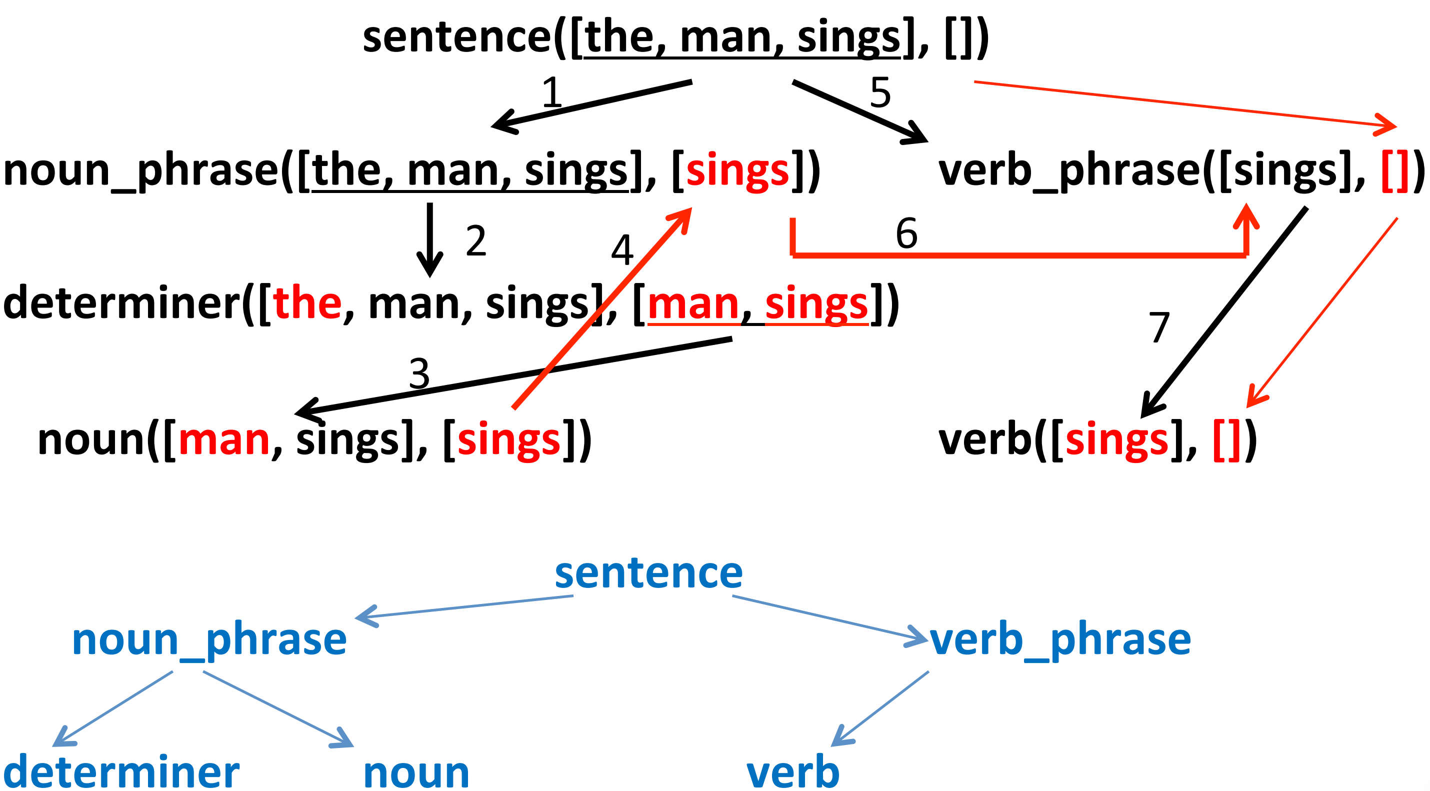
sentence([the, man, sings], []).

→ noun_phrase([the, man, sings], _Gxxx).	call
→ determiner([the, man, sings], _Gxxx).	call
← determiner([the, man, sings], [man, sings]).	return [man, sings]
→ noun([man, sings], _Gxxx).	call
← noun([man, sings], [sings]).	return [sings]
← noun_phrase([the, man, sings], [sings]).	return [sings]
→ verb_phrase([sings], []).	call
→ verb([sings], []).	call
← verb([sings], []).	return
← verb_phrase([sings], []).	return

sentence([the, man, sings], []).

Parsing using Prolog

Compare this with the parse tree



Prolog – call/exit/fail/redo

- See the grammar1.pl example – here is the trace of

?- sentence([the, man, eats, the, apple], []).

Call: (6) sentence([the, man, eats, the, apple], [])

Call: (7) noun_phrase([the, man, eats, the, apple], _G439)

Call: (8) determiner([the, man, eats, the, apple], _G439)

Exit: (8) determiner([the, man, eats, the, apple], [man, eats, the, apple])

Call: (8) noun ([man, eats, the, apple], _G439)

Exit: (8) noun ([man, eats, the, apple], [eats, the, apple])

Exit: (7) noun_phrase ([the, man, eats, the, apple], [eats, the, apple])

Call: (7) verb_phrase ([eats, the, apple], [])

Call: (8) verb ([eats, the, apple], [])

Fail: (8) verb ([eats, the, apple], [])

Redo: (7) verb_phrase ([eats, the, apple], [])

Call: (8) verb ([eats, the, apple], [])

Prolog – call/exit/fail/redo

...
Call: (7) verb_phrase ([eats, the, apple], []) 🡐 vp ::= v
Call: (8) verb ([eats, the, apple], [])
Fail: (8) verb ([eats, the, apple], [])
Redo: (7) verb_phrase ([eats, the, apple], []) 🡐 vp ::= v, np
Call: (8) verb ([eats, the, apple], _G439)
Exit: (8) verb ([eats, the, apple], [the, apple])
Call: (8) noun_phrase([the, apple], [])
Call: (9) determiner([the, apple], _G439)
Exit: (9) determiner([the, apple], [apple])
Call: (9) noun([apple], [])
Exit: (9) noun([apple], [])
Exit: (8) noun_phrase([the, apple], [])
Exit: (7) verb_phrase([eats, the, apple], [])
Exit: (6) sentence([the, man, eats, the, apple], [])
true.

Prolog – call/exit/fail/redo

- See the grammar1.pl example – here is the trace of

?- sentence([the, man, eats, the, apple], []).

…

Call: (7) verb_phrase ([eats, the, apple], [])

Call: (8) verb ([eats, the, apple], [])

Fail: (8) verb ([eats, the, apple], [])

Redo: (7) verb_phrase ([eats, the, apple], [])

Call: (8) verb ([eats, the, apple], [])

What has happened? Look at the grammar definitions

verb_phrase --> verb. ← **this definition fails!**

verb_phrase --> verb, noun_phrase. ← **the redo then tries this one!**

Parsing using Prolog

Definite Clause Grammars may also be used to generate parse trees

See http://en.wikipedia.org/wiki/Definite_clause_grammar

```
sentence(s(NP, VP))          → noun_phrase(NP), verb_phrase(VP).
noun_phrase(np(DET, N))      → determiner(DET), noun(N).
verb_phrase(vp(V))           → verb(V).
verb_phrase(vp(V, NP))       → verb(V), noun_phrase(NP).

determiner(d(the))           → [the].
noun(n(man))                 → [man].
noun(n(apple))               → [apple].
verb(v(eats))                →[eats].
verb(v(sings))               →[sings].
```

Parsing using Prolog

Prolog generates

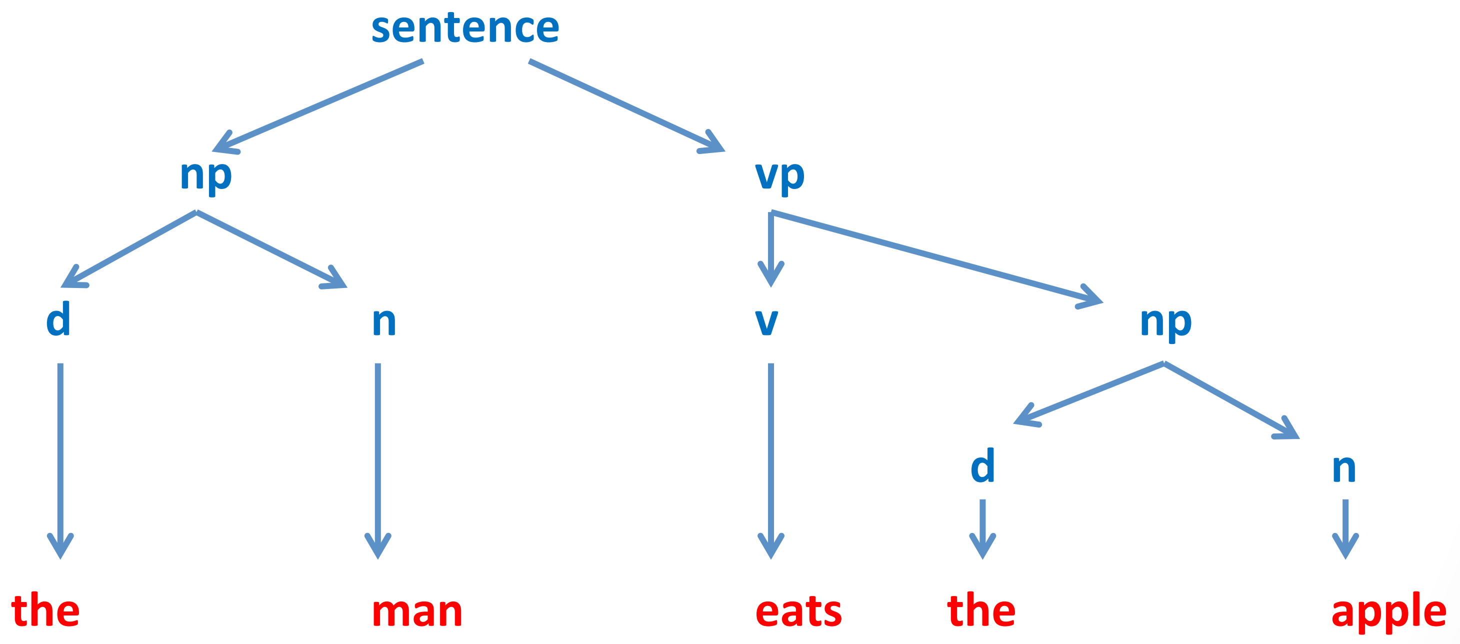
```
sentence(s(A, C), B, E)         :- noun_phrase(A, B, D), verb_phrase(C, D, E).
noun_phrase(np(A, C), B, E)  :- determiner(A, B, D), noun(C, D, E).
verb_phrase(vp(A), B, C)       :- verb(A, B, C),
verb_phrase(vp(A, C), B, E)    :- verb(A, B, D), noun_phrase(C, D, E).

determiner(d(the),     [ the   |  A ],   A).
noun(          n(man),    [ man   | A ],   A).
noun(          n(apple),  [ apple | A ],   A).
verb(          n(eats),     [ eats    | A ],   A).
verb(          n(sings),    [ sings | A ],   A).
```

Parsing using Prolog

The test is

?- sentence(PT, [the, man, eats, the, apple], []).

PT = s (np(d(the), n(man)), vp(v(eats), np(d(the), n(apple))))



Parsing using Prolog

This was a very elementary introduction to parsing sentences in English

There are in fact much more sophisticated techniques for parsing natural languages however we do not have the time in this course to continue

The techniques so far covered are enough to implement a parser for a programming language

The next exercise is to implement our Pascal-like language grammar

You should also be able to test each part of the grammar

This exercise should be doable in one lab pass

Parsing using Prolog

<program>	::=	<prog_header> <var_part> <stat_part>
<prog_header>	::=	program id (input , output) ;
<var_part>	::=	var <var_dec_list>
<var_dec-list>	::=	<var_dec> | <var_dedc_list> <var_dec>
<var_dec>	::=	<id_list> : <type> ;
<id_list>	::=	id | <id_list> , id
<type>	::=	integer | real | boolean
<stat_part>	::=	begin <stat_list> end .
<stat_list>	::=	<stat> | <stat_list> ; <stat>
<stat>	::=	<assign_stat>
<assign_stat>	::=	id := <expr>
<expr>	::=	<term> | <expr> + <term>
<term>	::=	<factor> | <term> * <factor>
<factor>	::=	(<expr>) | <operand>
<operand>	::=	id | number

Parsing using Prolog

Again we'll take a similar approach to that used for the C parser and implement the program header first. The start code is

```
program      → prog_head, var_part, stat_part.
prog_head    → [program], id, ['('], [input], [','], [output], [')'], [';'].
id           → [a]|[b]|[c].
var_part     → var_part_todo.
var_part_todo(_,_) :- write('var_part: To Be Done'), nl.
stat_part    → stat_part_todo.
stat_part_todo(_,_) :- write('stat_part: To Be Done'), nl.

testph :- prog_head([program, c, '(', input, ',', output, ')', ';'], []).
testpr :-    program([program, c, '(', input, ',', output, ')', ';'], []).
```

Note the mixture of → and :- definitions

Parsing using Prolog

```
program(A, D) :- prog_head(A, B), var_part(B, C), stat_part(C, D).
prog_head([program|A], H) :-
            id(A, B),     B=['('|C],        C=[input|D],
            D=[','|E],    E=[output|F],     F=[')'|G],        G=[;|H].
id(A, B)                    :- ( A=[a|B] ; A=[b|B] ; A=[c|B] ).
var_part(A, B)              :- var_part_todo(A, B).
var_part_todo(_, _)         :- write('var_part: To Be Done'), nl.
stat_part(A, B)             :- stat_part_todo(A, B).
stat_part_todo(_, _)        :- write('stat_part: To Be Done'), nl.

testph  :- prog_head([program, c, '(', input, ',', output, ')', ;], []).
testpr  :-    program([program, c, '(', input, ',', output, ')', ;], []).
```

Parsing using Prolog

The tests for this program outline are

```
?- testph.
true.
?- testpr.
var_part:   To Be Done
stat_part:  To Be Done
true.
```

See:-
http://www.cs.kau.se/cs/education/courses/dvgc01/PROLOGINFO/plcode/LabEx1.pl

Prolog generated code:-
http://www.cs.kau.se/cs/education/courses/dvgc01/PROLOGINFO/plcode/LabEx1.lis

Parsing using Prolog

This leaves lab 2 in Prolog: File Input + Lexer + Parser

See the specification for help material

http://www.cs.kau.se/cs/education/courses/dvgc01/lab_info/index.php?lab2=1

The File Input has been mostly written for you

Use the Clockson & Mellish reader with the 2 input files

cmlexer.txt & testok1.pas

Check that this corresponds to the given output in the specification

Skeletal ideas are given for the Lexer and the Parser

+ some extra help code

Your job is to put this all together to write the parser!

Again, test the parser stepwise.

Parsing using Prolog

Parsing in Prolog – Summary

Definite Clause Grammars allow the production rules P from the grammar to be expressed in a similar way in Prolog.

Note that left recursion must be changed to tail (right) recursion.

Prolog transforms the DCG syntax into normal Prolog code

DCG allows individual parts of the grammar to be easily tested as you write.

The trace predicate allows you to check the execution.

DCG allows you to write and test the parser first before adding the file input and the lexer, so that all the components may be tested.