
CSc 372

Comparative Programming Languages

27 : Prolog — Grammars

Christian Collberg

collberg+372@gmail.com

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg

Prolog Grammar Rules

- A DCG (**definite clause grammar**) is a phrase structure grammar annotated by Prolog variables.
- DCGs are translated by the Prolog interpreter into normal Prolog clauses.
- Prolog DCG:s can be used for generation as well as parsing. I.e. we can run the program backwards to generate sentences from the grammar.

Prolog Grammar Rules...

```
s      --> np, vp.
vp     --> v, np.
vp     --> v.
np     --> n.
n      --> [john].      n      --> [lisa].
n      --> [house].
v      --> [died].      v      --> [kissed].
```

```
?- s([john, kissed, lisa], []).
```

```
yes
```

```
?- s([lisa, died], []).
```

```
yes
```

```
?- s([kissed, john, lisa], []).
```

```
no
```

Prolog Grammar Rules...

`?- s(A, []).`

`A = [john,died,john] ;`

`A = [john,died,lisa] ;`

`A = [john,died,house] ;`

`A = [john,kissed,john] ;`

`A = [john,kissed,lisa] ;`

`A = [john,kissed,house] ;`

`A = [john,died] ;`

`A = [john,kissed] ;`

`A = [lisa,died,john] ;`

`A = [lisa,died,lisa] ;`

`A = [lisa,died,house] ;`

`A = [lisa,kissed,house] ;`

`A = [lisa,died] ;`

Implementing Prolog Grammar Rules

- Prolog turns each grammar rule into a clause with one argument.

- The rule $S \rightarrow NP VP$ becomes

`s(Z) :- np(X), vp(Y), append(X, Y,`

- This states that Z is a sentence if X is a noun phrase, Y is a verb phrase, and Z is X followed by Y .

Implementing Prolog Grammar Rules...

```
s(Z) :- np(X), vp(Y), append(X,Y,Z).  
np(Z) :- n(Z).  
vp(Z) :- v(X), np(Y), append(X,Y,Z).  
vp(Z) :- v(Z).  
n([john]).    n([lisa]).    n([house]).  
v([died]).    v([kissed]).
```

```
?- s([john,kissed,lisa]).
```

```
yes
```

```
?- s(S).
```

```
S = [john,died,john] ;
```

```
S = [john,died,lisa] ; ...
```

Implementing Prolog Grammar Rules...

- The append's are expensive — Prolog uses **difference lists** instead.
- The rule

$s(A, B) \text{ :- np}(A, C), \text{ vp}(C, B).$

says that there is a sentence at the beginning of A (with B left over) if there is a noun phrase at the beginning of A (with C left over), and there is a verb phrase at the beginning of C (with B left over).

Implementing Prolog Grammar Rules...

```
s(A,B)      :- np(A,C), vp(C,B).  
np(A,B)     :- n(A,B).  
vp(A,B)     :- v(A,C), np(C,B).  
vp(A,B)     :- v(A,B).  
n([john|R],R). n([lisa|R],R).  
v([died|R],R). v([kissed|R],R).
```

```
?- s([john,kissed,lisa], []).  
yes
```

```
?- s([john,kissed|R], []).  
R = [john] ;  
R = [lisa] ;...
```


Generating Parse Trees

- DCGs can build parse trees which can be used to construct a semantic interpretation of the sentence.
- The tree is built bottom-up, when Prolog returns from recursive calls. We give each phrase structure rule an extra argument which represents the node to be constructed.

Generating Parse Trees...

<code>s (s (NP , VP))</code>	<code>--> np (NP) , vp (VP) .</code>
<code>vp (vp (V , NP))</code>	<code>--> v (V) , np (NP) .</code>
<code>vp (vp (V))</code>	<code>--> v (V) .</code>
<code>np (np (N))</code>	<code>--> n (N) .</code>
<code>n (n (john))</code>	<code>--> [john] .</code>
<code>n (n (lisa))</code>	<code>--> [lisa] .</code>
<code>n (n (house))</code>	<code>--> [house] .</code>
<code>v (n (died))</code>	<code>--> [died] .</code>
<code>v (n (kissed))</code>	<code>--> [kissed] .</code>

Generating Parse Trees...

● The rule

$s(s(NP, VP)) \rightarrow np(NP), vp(VP).$

says that the top-level node of the parse tree is an s with the sub-trees generated by the np and vp rules.

?- s(S, [john, kissed, lisa], []).

S=s(np(n(john)),vp(n(kissed),np(n(lisa))))

?- s(S, [lisa, died], []).

S=s(np(n(lisa)),vp(n(died)))

?- s(S, [john, died, lisa], []).

S=s(np(n(john)),vp(n(died),np(n(lisa))))

Generating Parse Trees...

- We can of course run the rules backwards, turning parse trees into sentences:

```
?- s(s(np(n(john)),vp(n(kissed),  
    np(n(lisa)))), S, []).  
S=[john, kissed, lisa]
```

Ambiguity

- An ambiguous sentence is one which can have more than one meaning.

Lexical ambiguity:

homographic

- spelled the same
- *bat* (wooden stick/animal)
- *import* (noun/verb)

polysemous

- different but related meanings
- *neck* (part of body/part of bottle/narrow strip of land)

homophonic

- sound the same

● fo/too/two

Ambiguity...

Syntactic ambiguity:

- More than one parse (tree).
- Many missiles have many war-heads.
- “Duck” can be either a verb or a noun.
- “her” can either be a determiner (as in “her book”), or a noun: “I liked her dancing”.

Ambiguity...

$s(s(NP, VP)) \rightarrow np(NP), vp(VP).$
 $vp(vp(V, NP)) \rightarrow v(V), np(NP).$
 $vp(vp(V, S)) \rightarrow v(V), s(S).$
 $vp(vp(V)) \rightarrow v(V).$
 $np(np(Det, N)) \rightarrow det(Det), n(N).$
 $np(np(N)) \rightarrow n(N).$
 $n(n(i)) \rightarrow [i].$
 $n(n(duck)) \rightarrow [duck].$
 $v(v(duck)) \rightarrow [duck].$
 $v(v(saw)) \rightarrow [saw]. \quad n(n(saw)) \rightarrow [saw].$
 $n(n(her)) \rightarrow [her].$
 $det(det(her)) \rightarrow [her].$

?- $s(S, [i, saw, her, duck], []).$

Pascal Declarations

```
?- decl([const, a, =, 5, ;,  
        var, x, :, 'INTEGER', ;], []).
```

yes

```
?- decl([const, a, =, a, ;, var, x,  
        :, 'INTEGER', ;], []).
```

no

```
decl --> const_decl, type_decl,  
        var_decl, proc_decl.
```


Pascal Declarations

```
% Constant declarations
const_decl --> [ ].
const_decl -->
    [const], const_def, [;], const_defs.

const_defs --> [ ].
const_defs --> const_def, [;], const_defs.
const_def --> identifier, [=], constant.

identifier --> [X], {atom(X)}.
constant --> [X], {(integer(X); float(X))}.
```

Pascal Declarations...

```
% Type declarations
type_decl --> [ ].
type_decl --> [type], type_def, [;], type_defs.

type_defs --> [ ].
type_defs --> type_def, [;], type_defs.
type_def --> identifier, [=], type.

type --> ['INTEGER']. type --> ['REAL'].
type --> ['BOOLEAN']. type --> ['CHAR'].
```

Pascal Declarations...

```
% Variable declarations
var_decl --> [ ].
var_decl --> [var], var_def, [;], var_defs.

var_defs --> [ ].
var_defs --> var_def, [;], var_defs.
var_def  --> id_list, [:], type.

id_list  --> identifier.
id_list  --> identifier, [','], id_list.
```

Pascal Declarations...

```
% Procedure declarations
proc_decl --> [ ].
proc_decl --> proc_heading, [;], block.
proc_heading --> [procedure], identifier,
                formal_param_part.
formal_param_part --> [ ].
formal_param_part --> ['('],
                formal_param_section, [')'].
formal_param_section --> formal_params.
formal_param_section --> formal_params, [;],
                formal_param_section.
formal_params --> value_params.
formal_params --> variable_params.
value_params --> var_def.
variable_params --> [var], var_def.
```

Pascal Declarations – Building Trees

```
decl(decl(C, T, V, P)) -->  
    const_decl(C), type_decl(T),  
    var_decl(V), proc_declaration(P).
```

```
const_decl(const(null)) --> [ ].  
const_decl(const(D, Ds)) -->  
    [const], const_def(D), [;], const_defs(Ds).
```

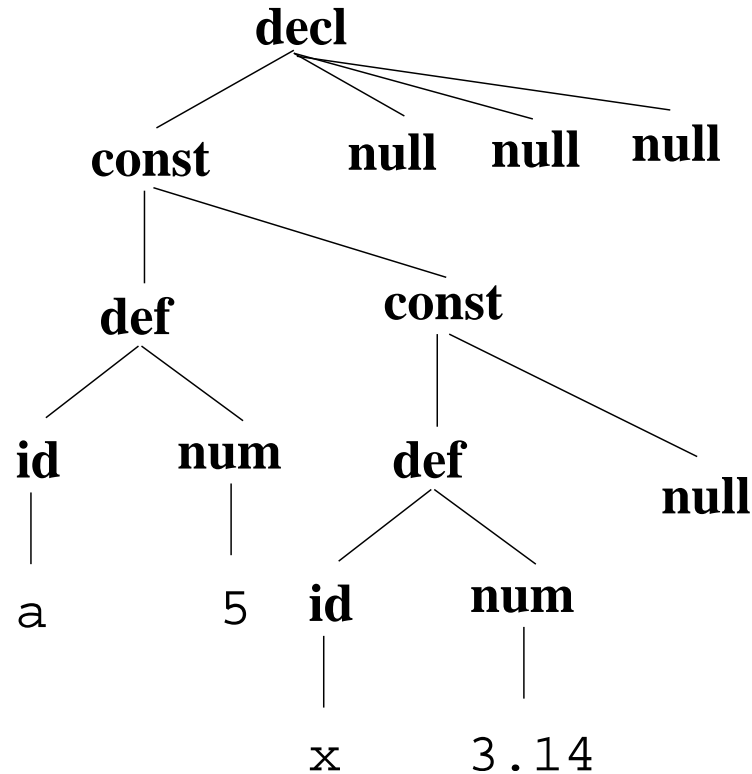
Pascal Declarations – Building Trees...

```
const_defs(null) --> [ ].
const_defs(const(D, Ds)) -->
    const_def(D), [ ; ], const_defs(Ds).

const_def(def(I, C)) --> ident(I), [=], const(C).

ident(id(X)) --> [X], {atom(X)}.
const(num(X)) --> [X], {(integer(X); float(X))}.
```

Pascal Declarations – Example Parse



Pascal Declarations – Example Parse...

?- decl(S, [const, a, =, 5, ;, x, =, 3.14, ;], [])

```
S = decl(  
    const(def(id(a), num(5)),  
          const(def(id(x), num(3.14)),  
                null)),  
    null, null, null)
```


Number Conversion

?- number(V, [sixty, three], []).

V = 63

?- number(V, [one, hundred, and, fourteen], []).

V = 114

?- number(V, [nine, hundred, and, ninety, nine], []).

V = 999

?- number(V, [fifty, ten], []).

no

Number Conversion...

`number(0) --> [zero].`

`number(N) --> xxx(N).`

`xxx(N) --> digit(D), [hundred], rest_xxx(N1),
 {N is D * 100+N1}.`

`xxx(N) --> xx(N).`

`rest_xxx(0) --> []. rest_xxx(N) --> [and], xx(N).`

`xx(N) --> digit(N).`

`xx(N) --> teen(N).`

`xx(N) --> tens(T), rest_xx(N1), {N is T+N1}.`

`rest_xx(0) --> []. rest_xx(N) --> digit(N).`

Number Conversion...

<code>digit(1) --> [one].</code>	<code>teen(10) --> [ten].</code>
<code>digit(2) --> [two].</code>	<code>teen(11) --> [eleven].</code>
<code>digit(3) --> [three].</code>	<code>teen(12) --> [twelve].</code>
<code>digit(4) --> [four].</code>	<code>teen(13) --> [thirteen].</code>
<code>digit(5) --> [five].</code>	<code>teen(14) --> [fourteen].</code>
<code>digit(6) --> [six].</code>	<code>teen(15) --> [fifteen].</code>
<code>digit(7) --> [seven].</code>	<code>teen(16) --> [sixteen].</code>
<code>digit(8) --> [eight].</code>	<code>teen(17) --> [seventeen].</code>
<code>digit(9) --> [nine].</code>	<code>teen(18) --> [eighteen].</code>
	<code>teen(19) --> [nineteen].</code>
<code>tens(20) --> [twenty].</code>	<code>tens(30) --> [thirty].</code>
<code>tens(40) --> [forty].</code>	<code>tens(50) --> [fifty].</code>
<code>tens(60) --> [sixty].</code>	<code>tens(70) --> [seventy].</code>
<code>tens(80) --> [eighty].</code>	<code>tens(90) --> [ninety].</code>

Expression Evaluation

- Evaluate infix arithmetic expressions, given as character strings.

```
?- expr(X, "234+345*456", []).  
X = 157554
```

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.  
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.  
expr(Z) --> term(Z).
```

```
term(Z) --> num(X), "*", term(Y), {Z is X * Y}.  
term(Z) --> num(X), "/", term(Y), {Z is X / Y}.  
term(Z) --> num(Z).
```

Expression Evaluation...

- Prolog grammar rules are equivalent to recursive descent parsing. Beware of left recursion!
- Anything within curly brackets is “normal” Prolog code.

```
num(C) --> "+", num(C).
```

```
num(C) --> "-", num(X), {C is -X}.
```

```
num(X) --> int(0, X).
```

```
int(L, V) --> digit(C), {V is L * 10 + C}.
```

```
int(L, X) --> digit(C), {V is L * 10 + C},  
                int(V, X).
```

```
digit(X) --> [C], {"0" =< C, C =< "9", X is C - "0"}.
```

Summary

- Read Clocksin & Mellish, Chapter 9.
- Grammar rule syntax:
 - A grammar rule is written `LHS --> RHS`. The left-hand side (LHS) must be a non-terminal symbol, the right-hand side (RHS) can be a combination of terminals, non-terminals, and Prolog goals.
 - Terminal symbols (words) are in square brackets: `n --> [house]`.
 - More than one terminal can be matched by one rule: `np --> [the,house]`.

Summary...

- Grammar rule syntax (cont):
 - Non-terminals (syntactic categories) can be given extra arguments: `s(s(N, V)) --> np(N), vp(V) ..`
 - Normal Prolog goals can be embedded within grammar rules: `int(C) --> [C], {integer(C)}.`
 - Terminals, non-terminals, and Prolog goals can be mixed in the right-hand side: `x --> [y], z, {w}, [r], p.`
- Beware of left recursion! `expr --> expr ['+']`
`expr` will recurse infinitely. Rules like this will have to be rewritten to use right recursion.

Homework

- Write a program which uses Prolog Grammar Rules to convert between English time expressions and a 24-hour clock (“Military Time”).
- You may assume that the following definitions are available:

```
digit(1) --> [one].    ....  
digit(9) --> [nine].  
teen(10) --> [ten].    ....  
teen(19) --> [nineteen].  
tens(20) --> [twenty]. ....  
tens(90) --> [ninety].
```

```
?- time(T, [eight, am], []).  
    T = 8:0 % Or, better, 8:00
```


Homework...

```
?- time(T, [eight, thirty, am], []).
```

```
T = 8:30
```

```
?- time(T, [eight, fifteen, am], []).
```

```
T = 8:15
```

```
?- time(T, [eight, five, am], []).
```

```
no
```

```
?- time(T, [eight, oh, five, am], []).
```

```
T = 8:5 % Or, better, 8:05
```

```
?- time(T, [eight, oh, eleven, am], []).
```

```
no
```

```
?- time(T, [eleven, thirty, am], []).
```

```
T = 11:30
```

```
?- time(T, [twelve, thirty, am], []).
```

```
T = 0:30 % !!!
```

Homework...

```
?- time(T,[eleven,thirty,pm],[ ]).  
    T = 23:30  
?- time(T,[twelve,thirty,pm],[ ]).  
    T = 12:30 % !!!  
?- time(T,[ten,minutes,to,four,am],[ ]).  
    T = 3:50  
?- time(T,[ten,minutes,past,four,am],[ ]).  
    T = 4:10  
?- time(T,[quarter,to,four,pm],[ ]).  
    T = 15:45  
?- time(T,[quarter,past,four,pm],[ ]).  
    T = 16:15  
?- time(T,[half,past,four,pm],[ ]).  
    T = 16:30
```