DCG <-> BNF

verbcalc

semantic grammar

mapping BNF to DCG

- rules to recognize numerical constants in C

```
constant -->
      dotted_digits; floating_constant ; character_constant
         ; string_constant.
floating_constant --> digits, exponent
         ; dotted_digits, ( exponent ; []).
dotted_digits --> digits, "."; digits, ".", digits ; ".", digits.
digits --> digit ; digit, digits.
exponent --> ("e" ; "E"), ("+" ; "-" ; []), digits.

digit --> [D], {is_dec_digit(D)}.  % a terminal symbol
```

grammars as declarative statements

- note the *denotational* character of a grammar rule:
  - it can be interpreted directly as stating what a sequence of a certain type is made of.
    - *A constant is a (sequence of) dotted_digits or a floating_constant or a character_constant or a string_constant.*
- but grammar rules have a procedural aspect:
  - digits --> digit; digits, digit. doesn't work.

semantics in the syntax

- DCG grammars can combine *semantic* with syntactic features:
  - i. e., rules can compute on syntactic constituents to check a constraint or compute a result
- use {*any goal*} as a subgoal in a DCG rule
- if not in {}, a goal in a DCG rule *must be the goal (head) of grammar rule.*

semantic goal

```
digit --> [D], {is_dec_digit(D)}.

is_dec_digit(D) :-
        member(D, "0123456789").
```


cleaner output

- | ?- X="a".
  X = [97]
- to avoid the ASCII codes in the output, when working with character strings, convert using string_to_list:
  ?- string_to_list(X, "a").
  X = "a"
- or use *atoms* rather than strings (as in verbcalc)

an 'easy' example

```
?- trace, constant("12", []).
  Call: (9) constant([49, 50], []) ?
  Call: (10) digits([49, 50], []) ?
  Call: (11) digit([49, 50], []) ?
  Call: (12) 'C'([49, 50], _L287, _L288) ?
  Exit: (12) 'C'([49, 50], 49, [50]) ?
  Call: (12) is_dec_digit(49) ?
  Exit: (12) is_dec_digit(49) ?
  Call: (12) []=[50] ?
  Fail: (12) []=[50] ?
     . . .
  Call: (11) digits([50], []) ?
  Call: (12) digit([50], []) ?
  Call: (13) 'C'([50], _L375, _L376) ?
  Exit: (13) 'C'([50], 50, []) ?
  . . .
  Exit: (11) digits([50], []) ?
  Exit: (10) digits([49, 50], []) ?
  Exit: (9) constant([49, 50], []) ?
```


lots of steps

```
[debug]  ?- trace, constant("2.7e-11",[]).
  Call: (9) constant([50, 46, 55, 101, 45, 49, 49], []) ?
  Call: (10) digits([50, 46, 55, 101, 45, 49, 49], []) ?
  Fail: (10) digits([50, 46, 55, 101, 45, 49, 49], []) ?
  Call: (10) dotted_digits([50, 46, 55, 101, 45, 49, 49], []) ?
  Fail: (10) dotted_digits([50, 46, 55, 101, 45, 49, 49], []) ?
  Call: (10) floating_constant([50, 46, 55, 101, 45, 49, 49], []) ?
  Call: (11) digits([50, 46, 55, 101, 45, 49, 49], _L255) ?
  . .
  Exit: (11) digits([50, 46, 55, 101, 45, 49, 49], [46, 55, 101, 45, 49, 49]) ?
  Call: (11) exponent([46, 55, 101, 45, 49, 49], []) ?
  Fail: (11) exponent([46, 55, 101, 45, 49, 49], []) ?
   . .
  Fail: (11) digits([50, 46, 55, 101, 45, 49, 49], _L255) ?
  Call: (11) dotted_digits([50, 46, 55, 101, 45, 49, 49], _L256) ?
  Exit: (11) dotted_digits([50, 46, 55, 101, 45, 49, 49], [55, 101, 45, 49, 49]) ?
```

and more steps

```
Call: (11) exponent([55, 101, 45, 49, 49], []) ?
Fail: (11) exponent([55, 101, 45, 49, 49], []) ?
Call: (11) [55, 101, 45, 49, 49]=[] ?
Fail: (11) [55, 101, 45, 49, 49]=[] ?
Exit: (11) dotted_digits([50, 46, 55, 101, 45, 49, 49], [101, 45, 49,
  49]) ?
Call: (11) exponent([101, 45, 49, 49], []) ?
Exit: (11) exponent([101, 45, 49, 49], []) ?
Exit: (10) floating_constant([50, 46, 55, 101, 45, 49, 49], []) ?
Exit: (9) constant([50, 46, 55, 101, 45, 49, 49], []) ?
```


extracting information

- mere recognition is usually not enough; to operate on the pieces, extend the rule with extra arguments:

```
dotted_digits(Integer, []) -->
    digits(Integer), ".".
dotted_digits(Integer, Fraction) -->
    digits(Integer), ".", digits(Fraction).
dotted_digits([], Fraction) -->
    ".", digits(Fraction).
```

parsing a decimal number

```
digits([Integer]) --> digit(Integer).
digits([D | Rest]) --> digit(D), digits(Rest).

digit([D]) --> [D], {is_digit(D)}.

| ?- digits(V, "12 - a number", Rest).
V = "1",
Rest = "2 - a number"
?- digits(V, "12 - a number", [" "|_]).
V = "12"
```


special vs. general tools

- compilers use specialized parsers which are heavily optimized
- Prolog parsers are easier to construct for a wider variety of tasks
  - formatting, e. g. dates,
  - parsing "small languages",
  - parsing natural language input;
- little optimization (often not needed)
- can be integrated with procedural C, C++ code

a "verbal calculator"

a small example

- high quality response to natural language input requires
  - large syntactic component
  - complex semantic analysis
- we look at a "toy" example:
  - a 'verbal calculator'
    - input: an English question
      - imagine it converted from speech input
    - output: a number (= 'meaning' of input)

sample input/output

- | ?- calc.
  |: What is 123 and 12345?

  12468

  |: add up 4 and -1234.

  -1230

a useful library routine

- use read_sent to simplify processing

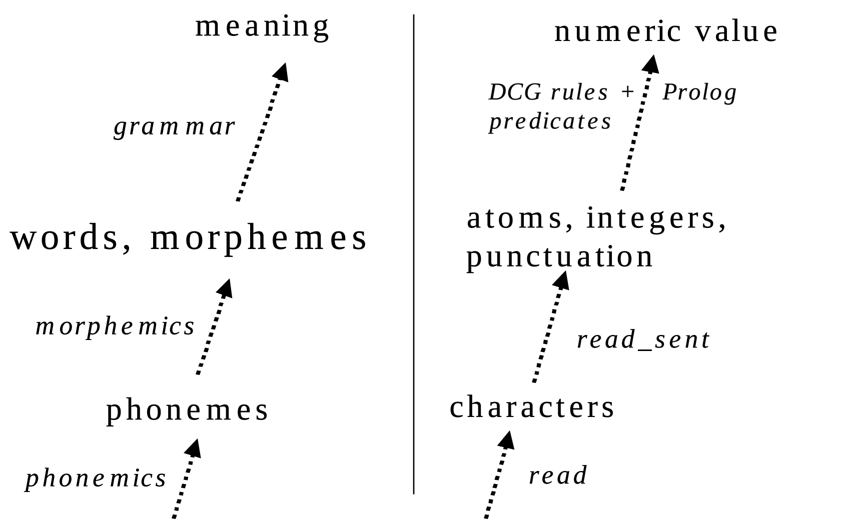
```
?- read_sent(S).
|: What is 123 and 12345?
S=[atom(what),atom(is),integer(123),atom(and),
        integer(12345),?]

?- read_sent(S).
|: Add up 4 and -1234.
S = [atom(add),atom(up),integer(4),atom(and),-,
        integer(1234),'.']
```

what's useful about read_sent?

- standarizes case
- extracts words
  - removes "white space"
- distinguishes between words and numbers
- simplifies processing of punctuation

linguistic parallels



some code

- the basic loop

```
:- consult('/cs/course.3401/read_sent.pl').
calc :-
   read_sent(Sentence),
   phrase(parse(Answer), Sentence),
   nl, write(Answer), nl, nl,
   calc.  % terminate with ^D.
```


the phrase predicate

phrase(Goal(A1, . .), Sequence) =
  Goal(A1, . . . , Sequence, []).

for a grammatical goal Goal.

Main advantage is separating the grammatical information

Goal(A1, . .)

from the data to which the rule is to be applied:

Sequence, []

a small beginning

```
parse(Answer) --> intro, number(N1), sumop,
   number(N2), ['?'],
      {Answer is N1 + N2}.
parse(Answer) --> diff(N1, N2), {Answer is N1 - N2}.

intro --> [atom(what), atom(is)] ; [atom(what), aposts].

diff(N1, N2) -->
    [atom(give), atom(me), atom(the), atom(difference),
     atom(between)], number(N1), [atom(and)],
    number(N2), ['.'].
```


integers only

```
sum(N1, N2) -->
    number(N1), sumop, number(N2).
% why isn't this number confused with built-in number
   predicate?
number(N) -->
   [integer(N)] ;
   ['-', integer(I)], {N is -I}.
sumop -->
   [atom(plus)] ; [atom(and)] ; ['+'].
```

(Note: read_sent can't handle decimal numbers.)

parsing a period

- ordinary notation can be subtle and ambiguous:

  "The number of errors is 2.

  "The cost is $2.10."

  "2. and 3. is 5."

  "Dr. Smith lives on Airdrie Dr."

look for patterns

- no small grammar can encompass variety of English numerical expressions

  but you can find lots of patterns: use them to

  -- reduce the number of rules

  --increase domain of the rules
- heuristic: write one rule and then see if you can find alternative phrasings for parts of it.
- Notice that we don't need surface grammar: verb, noun, etc.

semantic grammar

semantic grammar

- a semantic grammar for a fragment of English

  based on example from Covington, *Natural Language Processing for Prolog Programmers* , Prentice-Hall: 1994
- maps surface grammar of declarative sentence into a term (fact).

  "John runs."-> runs('John').
- here, meaning = a data structure;

  cf. meaning in *verbcalc* = numeric value

grammar fragment

```
s(Term) --> noun(Subject), verbPhrase(Subject^Term).
verbPhrase(Subject^Term) -->
   verb(Object^(Subject^Term)), noun(Object).
verbPhrase(Subject^Term) --> verb(Subject^Term).
verb(Y^(X^Term)) --> trans(Y^(X^Term)).
verb(X^Term) --> intrans(X^Term).

trans(Y^(X^Term)) -->transDict(V),
     {Term =.. [V, X, Y]}.
intrans(X^Term) -->
          intransDict(V), {Term =.. [V, X]}.
```


the dictionary

```
noun(N) -->
      [N], {member(N,
                ['Fido', 'Felix', dogs, cats, cars])}.

transDict(V) --> [V],
     {member(V, [chased, ate])}.
intransDict(V) --> [V],
    {member(V, [slept, 'woke up'])}.
```

toy examples

```
?- s(Meaning, ['Fido', chased, 'Felix'],[]).
Meaning = chased(Fido,Felix)

?- s(Meaning, [cars, chased,'Fido',],[]).
Meaning = chased(cars, Fido)

?- s(Meaning, ['Felix', slept],[]).
Meaning = slept(Felix)
```