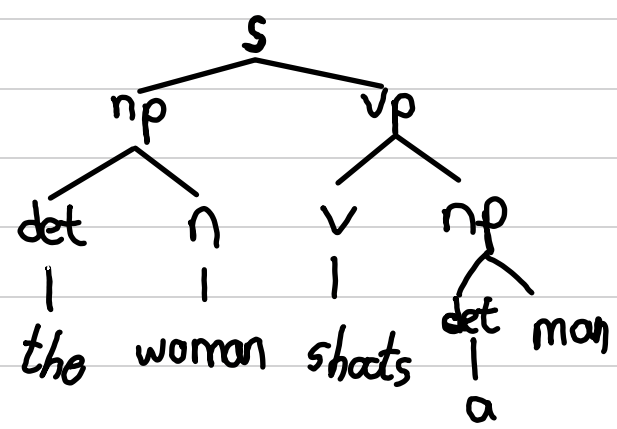
Week 8

- Prolog ada notasi spesial untuk define grammars yaitu DCG
- CFG adalah powerful mechanism dan bisa handle most syntactic aspects of natural languages (Such as English/Italian)
- s → np vp
  np → det n
  vp → v np
  vp → v
  det → *the*
  det → *a*
  n → *man*
  n → *woman*
  v → *shoots*

  --> / → symbol is used to define rules

  s, np, vp, det, n, v is non-terminal symbol

  the, a, woman, man, shoots in italic is terminal symbol
- np = noun phrase det = determiner v = verb
  vp = verb phrase n = noun s = sentence
- In linguistic grammar, non-terminal symbols correspond to grammatical categories
  terminal symbols = lexical items / simply words / alphabet
- Grammar contains 9 context free rules
- Context free rules terdiri atas
  - A single non-terminal symbol
  - Followed by →
  - Followed by finite sequence of terminal/non-terminal symbols
- s → np vp
  np → det n
  vp → v np
  vp → v
  det → *the*
  det → *a*
  n → *man*
  n → *woman*
  v → *shoots*



- **Parse trees** = Trees representing the syntactic structure of a string
  - ↳ Penting karena kasih info tentang string & structure

- Kalau dikasih a string of words, a grammar, dan :
  - ↳ Kalau bisa bangun parse trees, maka the string is **grammatical**
  - ↳ Kalau tidak bisa bangun parse trees, maka the string is **ungrammatical**

- **Language generated by a grammar** consists all strings that grammar classifies as grammatical.
  - ↳ Contoh : For instance
    - a woman shoots a man
    - a man shoots

    belong to language generated by our little grammar

- **Context free recogniser** = Program which correctly tells us apakah string belongs to the language generated by a CFG.
  = Program that correctly classifies strings as grammatical/ungrammatical

- **Context free parser** = Correctly decides whether a string belongs to the language generated by a CFG dan kasih tahu struk-turnya.
  - ↳ Intinya : Recogniser say yes/na
    Parser also gives parse tree

- **Context free language** = Language generated by CFG.

- s → np vp = Concate np-list with vp-list resulting in s-list.

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).      det([a]).
n([man]).      n([woman]).      v([shoots]).
```

```
?- s([the,woman,shoots,a,man]).
yes
```

```
?- s(S).
S = [the,man,shoots,the,man];
S = [the,man,shoots,the,woman];
S = [the,woman,shoots,a,man]
```

```
?- np([the,woman]).
yes
?- np(X).
X = [the,man];
X = [the,woman]
```

CFG Recognition Using append/3
CFG Recognition Using difference lists (lebih efisien)

```
s(A-C):- np(A-B), vp(B-C).
np(A-C):- det(A-B), n(B-C).
vp(A-C):- v(A-B), np(B-C).
vp(A-C):- v(A-C).
det([the|W]-W).      det([a|W]-W).
n([man|W]-W).   n([woman|W]-W).   v([shoots|W]-W).
```

- bisa jadi, cuman tanda

```
?- s([the,man,shoots,a,man]-[ ]).
yes
```

```
?- s(X-[ ]).
S = [the,man,shoots,the,man];
S = [the,man,shoots,a,man];
....
```

- Contoh usage of difference lists :

[a,b,c]-[ ] is list [a,b,c]
[a,b,c,d]-[d] is list [a,b,c]
[a,b,c|T]-T is list [a,b,c]
X-X is empty list

- DCG for a formal language → Kayak bahasa pada umumnya untuk define & study
  - ↳ Define by $a^n b^n$ or $a^n b^n c^n$
    - ↳ s → [].
      s → l,s,r.
      l → [a].
      r → [b].

```
?- s([a,a,a,b,b,b],[ ]).
yes
?- s([a,a,a,a,b,b,b],[ ]).
no
```

Week 9

- Extending the grammar by add rules for pronoun, add a rule saying that noun phrase can be pronouns.

```
?- s([she,shoots,him],[ ]).
yes
?- s([a,woman,shoots,him],[ ]).
yes
?- s([a,woman,shoots,he],[ ]).
yes
?- s([her,shoots,a,man],[ ]).
yes
s([her,shoots,she],[ ]).
yes
```

```
s --> np, vp.
np --> det, n.
np --> pro.
vp --> v, np.
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
```

```
pro --> [he].
pro --> [she].
pro --> [him].
pro --> [her].
```

- DCG ignores basic facts about English
  - ↳ *she* and *he* are subject pronoun & can't be used in object position
  - ↳ *her* and *him* are object pronoun & can't be used in subject position
  - ∞ Extend DCG with information about subject and object

```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
```

```
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
?- s([she,shoots,him],[ ]).
yes
?- s([she,shoots,he],[ ]).
no
?-
```

- s → np, vp. is really syntactic sugar for s(A,B) :- np(A,C), vp(C,B).
- s → np(subject), vp. translates into s(A,B) :- np(subject, A, C), vp(C,B).

```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].    pro(subject) --> [he].
n --> [man].      pro(subject) --> [she].
v --> [shoots].   pro(object) --> [him].
                  pro(object) --> [her].
```

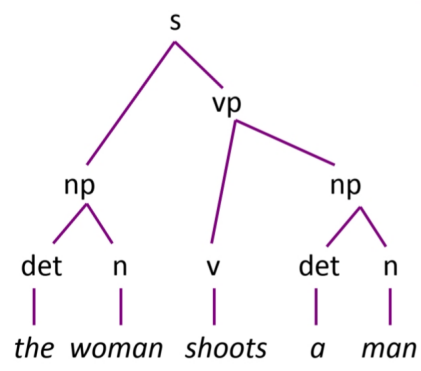
```
?- np(Type, NP, [ ]).
Type =_
NP = [the,woman];

Type =_
NP = [the,man];

Type =_
NP = [a,woman];

Type =_
NP = [a,man];

Type =subject
NP = [he]
```




```
s(np(det(the),n(woman)),
  vp(v(shoots),np(det(a),n(man)))))
```

```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
s(s(NP,VP)) --> np(subject,NP), vp(VP).
np(_,np(Det,N)) --> det(Det), n(N).
np(X,np(Pro)) --> pro(X,Pro).
vp(vp(V,NP)) --> v(V), np(object,NP).
vp(vp(V)) --> v(V)).
det(det(the)) --> [the].
det(det(a)) --> [a].
n(n(woman)) --> [woman].
n(n(man)) --> [man].
v(v(shoots)) --> [shoots].
pro(subject,pro(he)) --> [he].
pro(subject,pro(she)) --> [she].
pro(object,pro(him)) --> [him].
pro(object,pro(her)) --> [her].
```

```
?- s(T,[he,shoots],[]).
T = s(np(pro(he)),vp(v(shoots)))
yes
```

```
?- s(Tree,S,[]).
S = [the, woman, shoots, the, woman],
Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(the), n(woman))));
S = [the, woman, shoots, the, man],
Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(the), n(man))));
:
:
```

DCG for $a^n b^n c^n \backslash \{\varepsilon\}$

```
s(Count) --> as(Count), bs(Count), cs(Count).

as(0) --> [].
as(succ(Count)) --> [a], as(Count).

bs(0) --> [].
bs(succ(Count)) --> [b], bs(Count).

cs(0) --> [].
cs(succ(Count)) --> [c], cs(Count).
```

Example: DCG for $a^n b^n c^n \backslash \{\varepsilon\}$

```
s(Count) --> as(Count), bs(Count), cs(Count).

as(0) --> [].
as(NewCnt) --> [a], as(Cnt), {NewCnt is Cnt + 1}.

bs(0) --> [].
bs(NewCnt) --> [b], bs(Cnt), {NewCnt is Cnt + 1}.

cs(0) --> [].
cs(NewCnt) --> [c], cs(Cnt), {NewCnt is Cnt + 1}.
```

- 1 classic application of extra goals of DCGs in computational linguistics is seperate grammar rules from lexicon, yang berarti :
  - Eliminate all mention of individual words in DCG
  - Record all information about individual words in seperate lexicon

```
s --> np, vp.

np --> det, n.

vp --> v, np.
vp --> v.

det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [shoots].
```

➔

```
s --> np, vp.

np --> det, n.

vp --> v, np.
vp --> v.

det --> [Word],
    {lex(Word,det)}.

n --> [Word], {lex(Word,n)}.

v --> [Word], {lex(Word,v)}.
```

+

```
lex(the,     det).
lex(a,       det).
lex(woman,   n).
lex(man,     n).
lex(shoots,  v).
```

- Untuk tujuan linguistic, DCG have drawbacks such as :
  - Left-recursive rules
  - DCGs are interpreted top-down

- Listing noun phrases

```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
?- np(Type, NP, [ ]).
Type =_
NP = [the,woman];

Type =_
NP = [the,man];
:
:
Type =subject
NP = [he]
:
:
Type =object
NP = [him]
:
```

o

No Good

```
kalimat --> subyek, kata_kerja, obyek.
subyek --> manusia | kata_ganti.
manusia --> [pria] | [wanita].
kata_ganti --> [dia] | [mereka].
kata_kerja --> [makan] | [belajar].
obyek --> [nasi] | [soto] | [matematika] | [bahasa].
```

```
?- kalimat(S,[ ]).
S = [pria, makan, nasi];
S = [pria, makan, soto];
S = [pria, makan, matematika];
S = [pria, makan, bahasa];
:
```

Yeah

```
kalimat --> subyek, kata_kerja(X), obyek(X).
subyek --> manusia | kata_ganti.
manusia --> [pria] | [wanita].
kata_ganti --> [dia] | [mereka].
kata_kerja(makanan) --> [makan].
kata_kerja(pelajaran) --> [belajar].
obyek(makanan) --> [nasi].
obyek(makanan) --> [soto].
obyek(pelajaran) --> [matematika].
obyek(pelajaran) --> [bahasa].
```

```
?- kalimat(S,[ ]).
S = [pria, makan, nasi];
S = [pria, makan, soto];
S = [pria, belajar, matematika];
S = [pria, belajar, bahasa];
:
```

o

```
kalimat --> subyek, kata_kerja, obyek.
subyek --> [saya].
subyek --> [dia].
kata_kerja --> [makan].
obyek --> [sate].
obyek --> [soto].
```

```
?- kalimat(Tree, [dia, makan, soto],[]).
Tree = kalimat(subyek(dia), kata_kerja(makan), obyek(soto)).
```

```
kalimat(kalimat(Subyek, Kata_kerja, Obyek)) -->
    subyek(Subyek), kata_kerja(Kata_kerja), obyek(Obyek).
subyek(subyek(saya)) --> [saya].
subyek(subyek(dia)) --> [dia].
kata_kerja(kata_kerja(makan)) --> [makan].
obyek(obyek(sate)) --> [sate].
obyek(obyek(soto))  --> [soto].
```

Week 10

- Prolog ada predikat penting untuk comparing terms.

```
?- a==a.
yes

?- a==b.
no

?- a=='a'.
yes

?- a==X.
X = _443
no
```

```
?- X==X.
X = _443
yes

?- Y==X.
Y = _442
X = _443
no

?- a=U, a==U.
U = _443
yes
```

```
?- a \== a.
no

?- a \== b.
yes

?- a \== 'a'.
no

?- a \== X.
X = _443
yes
```

```
?- 2+3 == +(2,3).
yes

?- -(2,3) == 2-3.
yes

?- (4<2) == <(4,2).
yes
```

- 2 different uninstantiated variables are not identical terms.
  Variables instantiated with term T are identical to T.
- a & 'a' are the same, namun di program pasti beda karena membuat programming lebih pleasant dan lebih natural way of coding Prolog programs.
- +, -, <>, etc = Functors
  2+3 = Ordinary complex terms
  2+3 = +(2,3)

=	Unification predicate
\=	Negation of unification predicate
==	Identity predicate
\==	Negation of identity predicate
=:=	Arithmetic equality predicate
=\=	Negation of arithmetic equality predicate

- Lists as terms using | constructor

```
?- [a,b,c,d] == [a|[b,c,d]].
yes

?- [a,b,c,d] == [a,b,c|[d]].
yes

?- [a,b,c,d] == [a,b,c,d|[]].
yes

?- [a,b,c,d] == [a,b|[c,d]].
yes
```

- Lists are built out of 2 special terms such as :

  1) [] = Empty List
  2) '.' = A functor of arity 2 used to build non-empty lists

- 2 special terms itu disebut sebagai list constructors
- A recursive definition shows how they construct lists.
- Empty list has length 0
- A non-empty list is any term of the form .(term, list), where term is any Prolog term and list is any Prolog list.
- If list has length n, then length of .(term, list) is n+1.

- ?- .(a,[]) == [a].
  yes

  ?- .(f(d,e),[]) == [f(d,e)].
  yes

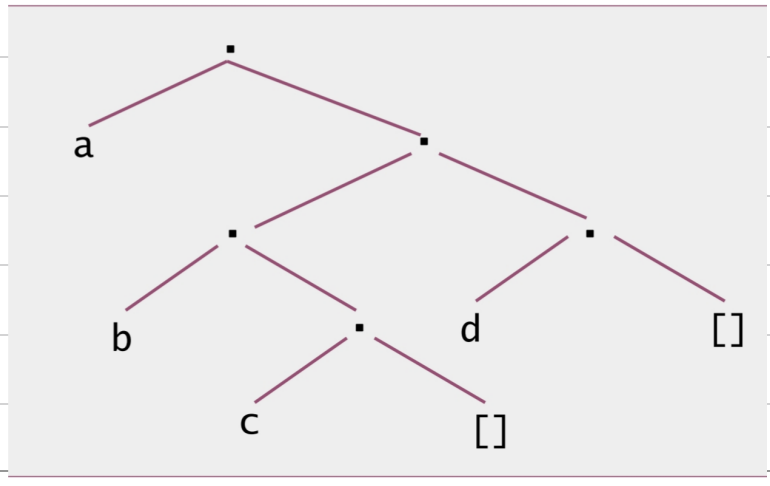
  ?- .(a,.(b,[])) == [a,b].
  yes

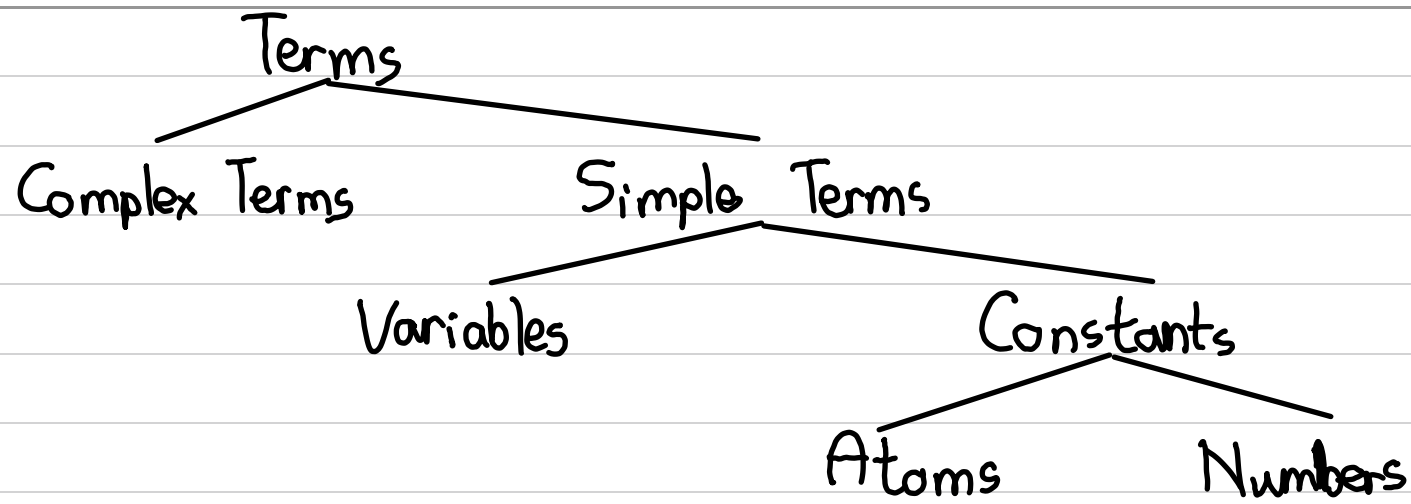
  ?- .(a,.(b,.(f(d,e),[]))) == [a,b,f(d,e)].
  yes

  a) Work seperti | notation.
  a) Trick bacanya adalah misalkan trees
     ↳ Internal nodes labeled with .
     ↳ All nodes have 2 daughter nodes
       ↳ Subtree under left daughter = head
       ↳ Subtree under right daughter = tail

  a) . = '[|]'

- [a,[b,c],d]





- atom/1 = Is the argument an atom?
- integer/1 = ... an integer?
- float/1 = ... a floating point number?
- number/1 = ... an integer / float?
- atomic/1 = ... a constant?
- var/1 = ... an uninstantiated variable?
- nonvar/1 = ... an instantiated variable / another term that is not an uninstantiated variable?

- Complex term itu lihatnya arity, functor, dan argument.
- functor/3 predicate gives the functor & arity of complex terms.
  - ?- functor(friends(lou, andy), F, A).
    F = friends
    A = 2
    yes
  - ?- functor([lou, andy, vicky], F, A).
    F = . atau F = [|]
    A = 2
    yes
  - ?- functor(14, F, A).
    F = 14
    A = 0
    yes
  - ?- functor(Term, friends, 2).
    Term = friends(_, _).
    yes

- Check complex term : complexTerm(X) :-
  nonvar(X),
  functor(X,_,A),
  A > 0.

- Predicate arg/3 untuk cek arguments of complex terms dan terdiri atas :
  - number N
  - complex term T
  - Nth argument of T

- ```
  ?- arg(2,likes(lou,andy),A).
  A = andy
  yes
  ```

- String di Prolog itu adalah list of character codes

- ```
  ?- S = "Vicky".
  S = [86,105,99,107,121]
  yes
  ```

  ```
  ?- atom_codes(vicky,S).
  S = [118,105,99,107,121]
  yes
  ```

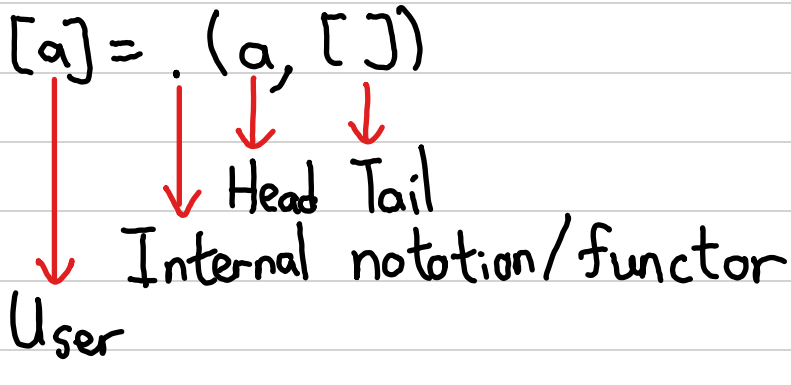
- Properties of operators :

  1) Infix operators : Functor writtens between their arguments
     ↳ + - == < > etc
  2) Prefix operators : Functor writtens before their arguments
     ↳ -
  3) Suffix operators : Functor writtens after their arguments
     ↳ ++ di C

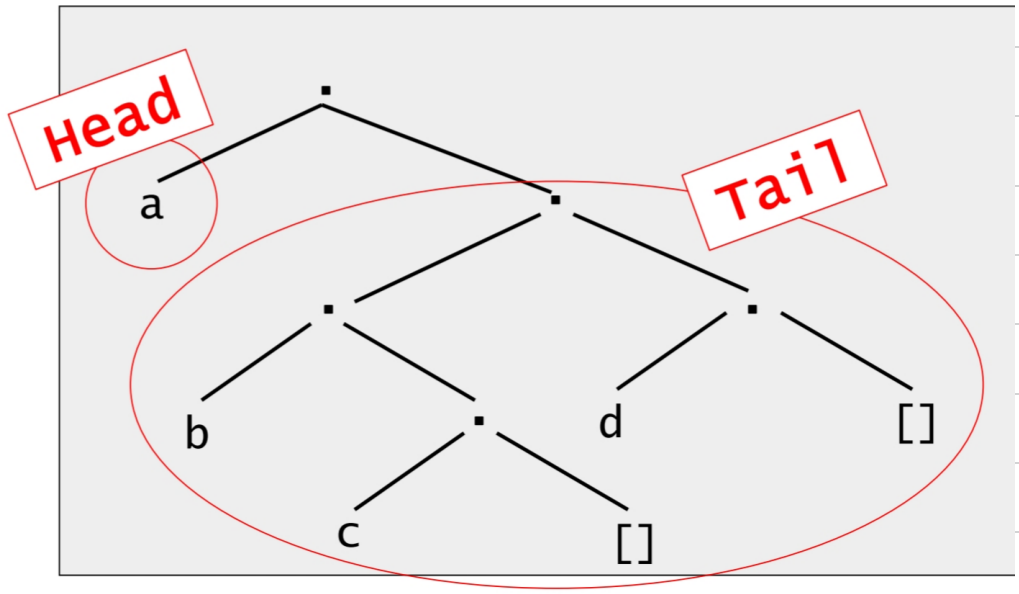
- Every functor has a certain precedence to work out ambiguous expressions, seperti 2+3*3 itu (2+3)*3 \/ 2+(3*3). Karena precedence + lebih besar dari *, maka + jadi main functor.

- Prolog pakai associativity to disambiguate operators with the same precedence value.
- Operators can be defined as non-associative, in which case use bracketing in ambiguous cases.
  ↳ :- -->
- Define operators : op(Precedence, Type, Name)
  ↳ Precedence : Number between 0 and 1200
  ↳ Type : The type of operator
- Types of Operators :
  - yfx = Left-associative, infix
  - xfy = Right-associative, infix
  - xfx = Non-associative, infix
  - fx = Non-associative, prefix
  - fy = Right-associative, prefix
  - xf = Non-associative, postfix
  - yf = Left-associative, postfix
-

1200	xfx	`-->, :-`
1200	fx	`:-, ?-`
1150	fx	`dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile`
1100	xfy	`;, |`
1050	xfy	`->, op*->`
1000	xfy	`,`
954	xfy	`\`
900	fy	`\+`
900	fx	`~`
700	xfx	`<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is`
600	xfy	`:`
500	yfx	`+, -, /\, \/, xor`
500	fx	`+, -, ?, \`
400	yfx	`*, /, //, rdiv, <<, >>, mod, rem`
200	xfx	`**`
200	xfy	`^`

- 

- Example: [a,[b,c],d]




```
?- .(a,.(b,.(c,[]))) = [a,b,c].
true.

.(a,.(b,.(c,[])))
.(a,.(b,[c])) =
.(a,[b,c]) =
[a,b,c]
```

UNIFIED

```
?- .(a,.(b,.(c,[]))) = [a,b|[c]].
true.

.(a,.(b,.(c,[]))) =
.(a,.(b,[c])) =
.(a,[b,c]) =
[a,b,c] =
[a,b|[c]]
```

UNIFIED

Example of complexTerm/3 :

```
?- complexTerm(mia).
no
?- complexTerm(friend(you,me)).
yes
```

Example of precedence :

```
?- S is 2 + 3 * 4 ** 2.
S = 50.

+  : Precedence value 500 → Main functor
*  : Precedence value 400 → Next functor
** : Precedence value 200 → Last functor

2 + 3 * 4 ** 2 = +(2,*(3,**(4,2)))
               = +(2,*(3,16))
               = +(2,48)
               = 50
?- 2 * 3 =:= 6 * 1.
true.

=:= : Precedence value 700 → Main functor
*   : Precedence value 400 → Next functor

2 * 3 =:= 6 * 1 = =:=(*(2,3),*(6,1))
                = =:=(6,*(6,1))
                = =:=(6,6)
                = true
```

Example of associative

```
?- S is 10 mod 6 mod 3.
S = 1.

mod : Left associative

10 mod 6 mod 3 = (10 mod 6) mod 3
               = 4 mod 3
               = 1

?- S is 2 ^ 3 ^ 0.
S = 2.

^ : Right associative

2 ^ 3 ^ 0 = 2 ^ (3 ^ 0)
          = 2 ^ 1
          = 2

?- S is 2 ** 3.
S = 8.

?- S is 2 ** 3 ** 0.
Error : Operator priority clash.

** : Non associative

                 = (2 ** 3) ** 0 ??
2 ** 3 ** 0 <
                 = 2 ** (3 ** 0) ??
```

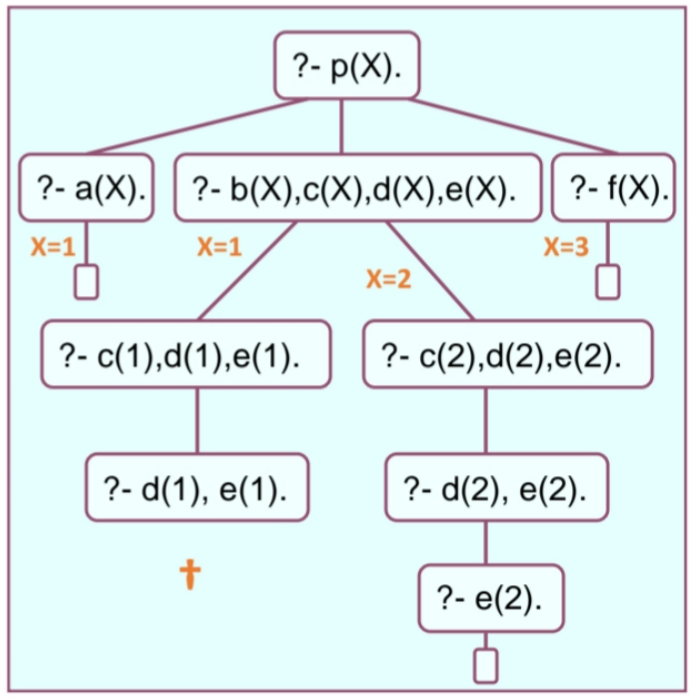
Week 11

- **Backtracking** is a characteristic feature of Prolog, but can lead to inefficiency such as :
  - Waste time exploring possibilities that lead nowhere.
  - Would be nice to have some control.

- Cut predicate `!`/0 offers a way to control back-tracking.
  - ↳ Contoh : `p(x) :- b(x), c(x), !, d(x), e(x).`

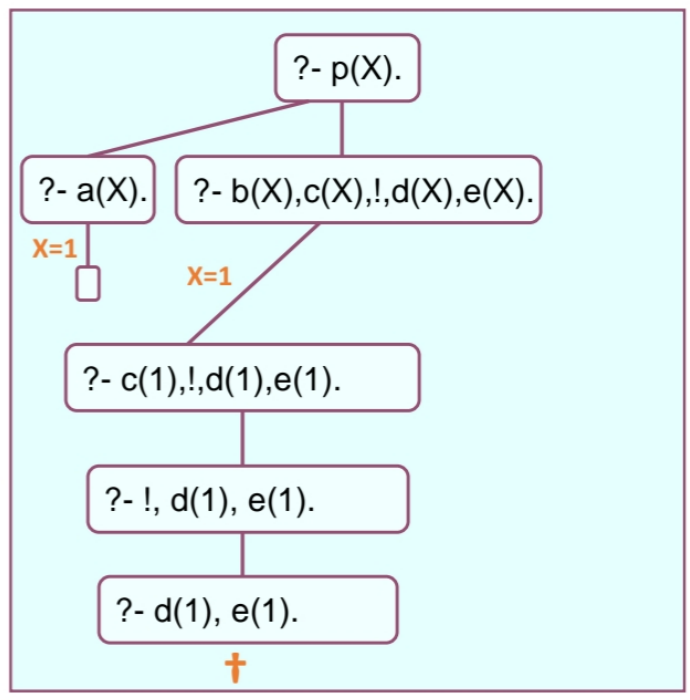
-
```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).  b(2).
c(1).  c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X=1;
X=2;
X=3;
no
```



-
```
p(X):- a(X).
p(X):- b(X), c(X), !, d(X), e(X).
p(X):- f(X).
a(1).
b(1).  b(2).
c(1).  c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X=1;
```



- Cut is a goal that always succeeds.

- The cut only commits us to choices made since the parent goal was unified with the left-hand side of the clause containing the cut.

- q :- $p_1, \ldots, p_n, !, r_1, \ldots, r_n$
  - ↳ When reach the cut, it commits us :
    - a) To this particular clause of q
    - b) To the choices made by $p_1, \ldots, p_n$
    - c) NOT to choices made by $r_1, \ldots, r_n$

- In `max(X, Y, Y) :- X =< Y.`
  `max(X, Y, X) :- X > Y.`

  `?- max(3, 4, Y)`, maka akan menghasilkan `Y = 4`, tapi ketika di `max(X, Y, X)`, akan tak berarti.

Maka, diperlukan cuts : `max(X, Y, Y) :- X =< Y, !.`
`max(X, Y, X) :- X > Y.`

- ↳ Jika `X =< Y` succeeds, the cut commits us to this choice and the second clause of `max/3` tidak diconsidered.
- ↳ Jika `X =< Y` fails, Prolog goes to second clause.

- Cuts that do not change the meaning of a predicate are called green cuts

- Cut that change the meaning of a predicate are called red cuts

- Programs containing red cuts :
  - Are not fully declarative
  - Can be hard to read
  - Can lead to subtle programming mistakes.

- When Prolog fails, it tries to backtrack.

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,a).
yes
?- enjoys(vincent,b).
no
?- enjoys(vincent,c).
yes
?- enjoys(vincent,d).
yes
```

- The cut-fail combination seems to be offering us some form of negation. It is called negation as failure
  - neg(Goal) :- Goal, !, fail.
    neg(Goal).
- neg() = \+
- Negation as failure is not logical negation.
- Intinya kalau ketemu cuts seperti :

```
enjoys(X,Y) :- bigKahunaBurger(X), !, a(Y). .... (1)
enjoys(X,Y) :- a(X), b(Y). .... (2)
```

  - Kalau bigKahunaBurger(X) true, maka (2) tidak dijalankan dan Prolog hanya membaca a(Y).
  - Kalau bigKahunaBurger(X) false, maka setelah ! Prolog tidak baca.

```
enjoys(X) :- bigKahunaBurger(X), !, fail. .... (1)
enjoys(X) :- a(X) .... (2)
```

  - Kalau bigKahunaBurger(X) true, maka (2) tidak dijalankan dan Prolog hanya membaca fail, dan otomatis kalau bigKahunaBurger(X) true sudah pasti respon Prolog false.
  - Kalau bigKahunaBurger(X) false, maka (2) dibaca.

Week 12

- Prolog ada 5 basic manipulation commands yaitu :
  - assert /1
  - asserta /1
  - assertz /1
  - retract /1
  - retractall /1

  } Adding information

  } Removing information

-
```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).
```

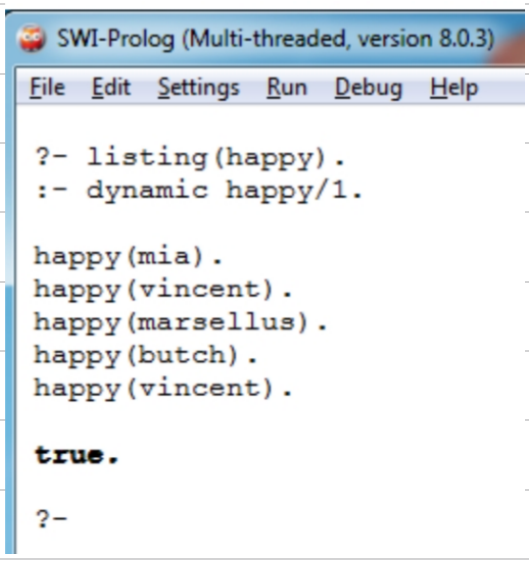
```
?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?- assert(happy(vincent)),
   assert(happy(marsellus)),
   assert(happy(butch)),
   assert(happy(vincent)).
yes
?-
```

- ?-listing. = Menampilkan semua complex term dengan rulenya masing-masing.

- Kalau ingin mengecek fact / complex term yang ada pada Prolog dari yang sudah di-assert, maka dapat dilakukan dengan :

  ?-listing(happy).

  ↓ hasilnya



Sehingga bisa ambil datanya →

```
?- happy(X).
X = mia ;
X = vincent ;
X = marsellus ;
X = butch ;
X = vincent.

?-
```

- Hence, the database manipulations have changed the meaning of the predicate happy/1.

- Predicates which meaning changing during runtime are called dynamic predicates.
  - ↳ happy/1 is a dynamic predicate
  - ↳ Some Prolog interpreters require a declaration of dynamic predicates.
- Ordinary predicates are sometimes referred to as static predicates.

```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- assert( (naive(X):- happy(X)).
yes
?-
```

- retract/1 : Remove 1 clause
  retractall/1 : Remove several clauses simultaneously

```
happy(mia).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(marsellus)).
yes
?- retract(happy(vincent)).
yes
```

```
happy(mia).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(X)).
```

?- retractall(happy(X))

```
naive(A):- happy(A).
```

```
?- retract(happy(X)).
X=mia;
X=butch;
X=vincent;
no
?-
```

- asserta/1 : Places asserted material at the beginning of database
  assertz/1 : Places asserted material at the end of database

- Jika ingin mengecek rule dapat dengan syntax :
  ?- listing(naive)
  ↓ hasilnya

SWI-Prolog (Multi-threaded, version 8.0.3)

File Edit Settings Run Debug Help

```
?- listing(happy),listing(naive).
:- dynamic happy/1.

happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

:- dynamic naive/1.

naive(A) :-
    happy(A).

true.

?-
```

- Database manipulation useful for storing the results to computations, in case we need to recalculate the same query. This is called memoisation / caching

-
```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

```
?- addAndSquare(3,7,X).
```

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).
```

```
?- addAndSquare(3,7,X).
X=100
yes
?- addAndSquare(3,4,X).
X=49
yes
```

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
   assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).
```

```
?- retractall(lookup(_, _, _)).
```

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
   assert(lookup(X,Y,Res)).
```

```
?- retractall(lookup(_, _, _)).
yes
?-
```

Before doing memoisation, it is important to :

Create file : “db.pl”

Insert the followings :

```
:- dynamic lookup/3.
addAndSquare(X,Y,Res):-
  lookup(X,Y,Res), !.
addAndSquare(X,Y,Res):-
  Res is (X+Y)*(X+Y),
  assert(lookup(X,Y,Res)).
```

Save the file

Load and compile buffer the file

then →

Append the followings will be helpful :

```
listall :-
  listing(lookup),listing(addAndSquare).
list1 :- listing(lookup).
list2 :- listing(addAndSquare).
```

Save the file

Load and compile buffer the file

Query `listall`, `list1` and `list2` to filter the listing only to the necessary targets (`addAndSquare` and/or `lookup`)

Red cut

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
   assert(lookup(X,Y,Res)).
```

Green cuts

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    \+ lookup(X,Y,Res), !,
    Res is (X+Y) * (X+Y),
     assert(lookup(X,Y,Res)).
```

→ Mirip IF EXISTS

A word of warning on database manipulation :

- Often is a useful technique
- But can lead to dirty, hard to understand code
- It is non declarative, non logical
- So should be cautiously

- There may be many solutions to a Prolog query, but Prolog generates solusi one by one. Terkadang, all solutions pengen ada to a query in one go, dapat dengan function built-in Prolog yaitu:

findall/3, bagof/3, setof/3

Collect all solutions to a query and put them into a single list

- findall(O, G, L): Produces a list L of all the objects O that satisfy the goal G.
  - ↳ Always succeed
  - ↳ Unifies L with empty list if G cannot be satisfied.

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(X,descend(martha,X),L).
L=[charlotte,caroline,laura,rose]
yes
?- findall(f:X,descend(martha,X),L).
L=[f:charlotte,f:caroline,f:laura,f:rose]
yes
```

```
?- findall(d,descend(martha,X),L).
L=[d,d,d,d]
yes
```

- bagof(O, G, L): Produces a list L of all the objects O that satisfy the goal G
  - ↳ Only succeeds if the goal G succeeds
  - ↳ Binds free variables in G

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- bagof(Chi,descend(Mot,Chi),L).
Mot=caroline
L=[laura, rose];
Mot=charlotte
L=[caroline,laura,rose];
Mot=laura
L=[rose];
Mot=martha
L=[charlotte,caroline,laura,rose];
no
```

setof(O,G,L) % Produces a sorted list L of all the objects O that satisfy the goal G.
- ↳ Only succeeds if the goal G succeeds
- ↳ Binds free variables in G
- ↳ Remove duplicates from L
- ↳ Sorts the answers in L

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                descend(Z,Y).
```

```
?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose,
    caroline, laura, rose, laura, rose,
    rose]
yes

?- setof(Chi,Mot^descend(Mot,Chi),L).
L=[caroline, charlotte, laura, rose]
yes
```

Week 13

- Prolog baca predicate definitions that are stored in a file is using the square brackets, supaya tidak redefine each time panggil predikat yang basic (member/2, append/3).

- ```
  ?- [myFile].
  {consulting(myFile.pl)...}
  {myFile.pl consulted, 233 bytes}
  yes
  ```

  ```
  ?- [myFile1, myFile2, myFile3].
  {consulting myFile1.pl...}
  {consulting myFile2.pl...}
  {consulting myFile3.pl...}
  ```

  consult file > 1

- ensure_loaded ([myFile1, myFile2]).

  Check whether predicate definitions are known already

- ```
  % This is the file:  printActors.pl

  printActors(Film):-
     setof(Actor,starring(Actor,Film),List),
     displayList(List).

  displayList([]):- nl.
  displayList([X|L]):-
      write(X), tab(1),
      displayList(L).
  ```

  ```
  % This is the file:  printMovies.pl

  printMovies(Director):-
     setof(Film,directed(Director,Film),List),
     displayList(List).

  displayList([]):- nl.
  displayList([X|L]):-
      write(X), nl,
      displayList(L).
  ```

  ```
  % This is the file main.pl

  :- [printActors].
  :- [printMovies].
  ```

  ```
  ?- [main].
  {consulting main.pl}
  {consulting printActors.pl}
  {printActors.pl consulted}
  {consulting printMovies.pl}
  The procedure displayList/1 is being
      redefined.
  Old file: printActors.pl
  New file: printMovies.pl
  Do you really want to redefine it?
  (y, n, p, or ?)
  ```

- Built-in module:
  - module/1 & module/2: To create a module/library
  - use_module/1 & use_module/2: To import predicates from a library
    - 1st argument = Name of module
    - 2nd/optional = List of exported predicate

```
% This is the file:  printActors.pl
:- module(printActors,[printActors/1]).

printActors(Film):-
   setof(Actor,starring(Actor,Film),List),
   displayList(List).

displayList([]):- nl.
displayList([X|L]):-
    write(X), tab(1),
    displayList(L).
```

```
% This is the file:  printMovies.pl
:- module(printMovies,[printMovies/1]).

printMovies(Director):-
   setof(Film,directed(Director,Film),List),
   displayList(List).

displayList([]):- nl.
displayList([X|L]):-
    write(X), nl,
    displayList(L).
```

```
% This is the revised file main.pl

:- use_module(printActors).
:- use_module(printMovies).
```

```
% This is the revised revised file main.pl

:- use_module(printActors,[printActors/1]).
:- use_module(printMovies,[printMovies/1]).
```

- Many of the most common predicates are predefined by Prolog interpreters. (member/2 & append/3 come as part of a library)
- Library is a module defining common predicates, and can be loaded using the normal predicates for importing modules.
- Ketika specifying nama library, tentu module itu adalah library, dan Prolog akan lihat tempat yang benar, namely a directory where all libraries are stored.

```
?- use_module(library(lists)).
```

- Untuk write & append file harus open stream :

```
...
open('hogwarts.txt', write, Stream),
write(Stream, 'Hogwarts'),
close(Stream),
...
```

```
...
open('hogwarts.txt', append, Stream),
write(Stream, 'Hogwarts'),
close(Stream),
...
```

- Other useful predicates : tab/2, nl/1, format/3

- *houses.txt:*

```
gryffindor.
hufflepuff.
ravenclaw.
slytherin.
```

```
main:-
    open('houses.txt',read,S),
    read(S,H1),
    read(S,H2),
    read(S,H3),
    read(S,H4),
    close(S),
    write([H1,H2,H3,H4]), nl.
```

- Pada read/2 hanya bekerja pada Prolog terms dan akan mengakibatkan run-time error ketika 1 mencoba untuk baca at the end of a file.

- Built_in predicate at_end_of_stream/1 checks whether the end of a stream has been reached. It will succeed when end of stream is reached, otherwise it will fail

-

```
main:-
    open('houses.txt',read,S),
    readHouses(S,Houses),
    close(S),
    write(Houses), nl.

readHouses(S,[]):-
     at_end_of_stream(S).

readHouses(S,[X|L]):-
     \+ at_end_of_stream(S),
     read(S,X),
     readHouses(S, L).
```

- get_code/2 reads the next available char of the stream, with 1st argument = stream & 2nd argument = char code.