

Def.:
Tilstand, assignment var-værdi
EksPLICIT beregningsrækkefølge

Def.: Klasser til beskriv. af data
Dynamisk oprettelse med "new"
Udbredelse af imperativ paradigme

Datalogi C
18. marts 2002
Jens Christiansen

Prolog, et logisk programmeringssprog

Formål: Et indblik i at programmeringssprog er mere og andet end imperativ/objektorienteret, f.eks. deklarativt

Egenskaber ved Prolog:

- ingen eksplicite typer eller klasser
- regel-baseret, bygger på matematisk logik
- stærk udtrykskraft/funktionalitet pr. programlinje
- interaktiv, eksperimenterede programmeringsstil

Def.: (Ideal om) at program er beskrivelse af data og deres egenskaber

NB: Ikke svært, forførende let... i starten!
Kræver meget øvelse at mestre det!

Baggrund for Prolog

PROgramming in LOGic

Syntaks: delmængde af 1.-ordens prædikatalogik

Deklarativ semantik: logisk konsekvens fra logik

Procedural semantik:

Resolution, bevisregel m. unification; Robinson, 1965

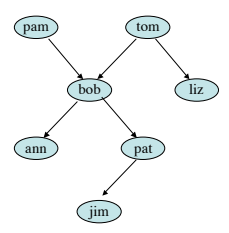
A.Colmerauer & co. (Marseille), ca. 1970: »Prolog«

D.H.D. Warren: Effektiv compiler abstrakt maskine "WAM", 1975,

Udbredelse af sproget R.Kowalski »Logic for Problem solving«, 1979,

Program er en *beskrivelse af data*

```
parent( pam, bob). % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



Test program

Begreber:

- prædikater: **parent**
 - beskriver relation
 - bestemt ved fakta, regler, tilsammen klausuler
- atomer: **tom, bob, x, y**
- variable: **x, y, Tom**
- mål: **parent(A, a)**
- Forespørgsler....

Uheldig betegnelse
»atomer« er noget
andet i mat.logik!

Forespørgsler

Simple forespørgsler

?- **parent(x, y).**

... giv mig værdier for **x** og **y** så **parent(x, y)** logisk konsekvens af program

Sammensat forespørgsel

?- **parent(pam, x), parent(x, y).**

... giv mig **x** og **y**, så...

Test program

Procedural semantik

```
parent(pam, bob).
parent(tom, bob).
parent(tom, iiz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```

Unification term=term?
fra venstre mod højre
og med
backtracking
y=pat
≈ omgøre og prøve nye valg

Success!
Flere løsninger?

Slet ikke flere løsninger her :(

Regler

```
female(pam).  
male(tom).  
male(bob).  
female(liz).  
female(pat).  
female(ann).  
male(jim).
```

Procedural semantik

som før + omskrive delmål vha. regler

Deklarativ semantik ≈ logisk konsekvens

med regler læst som f.eks.

$\exists x,y,x: p(x,y) \wedge f(x) \rightarrow m(x,y)$

```
mother(X, Y):-  
  parent(X, Y),  
  female(X).
```

Hele fidusen:

procedural ≈ deklarativ

(undtaget når proc. går i uendelig løkke)

Test program

En rekursiv regel

```
predecessor(X, Z):-  
  parent(X, Z).
```

Fungerer fint men gør i løkke
hvis rækkeflg. byttes

```
predecessor(X, Z):-  
  parent(X, Y),  
  predecessor(Y, Z).
```

```
?- predecessor(tom, pat).
```

Test program

Flere eksempler på regler

```
offspring(Y, X):-  
  parent(X, Y).
```

```
grandparent(X, Z):-  
  parent(X, Y),  
  parent(Y, Z).
```

```
sister(X, Y):-  
  parent(Z, X),  
  parent(Z, Y),  
  female(X),  
  dif(X, Y).
```

Benytter lidt
anden procedurel
semantik

Indbygget »constraint«
i SICStus Prolog

Test program

Overvej

- Hvor mange linjer Java-kode skal der til for at lave denne familiedatabase?
- Hvad vi har set indtil nu er allerede et kraftigt modelleringsværktøj
 - et eksempel...

Logiske kredsløb

(Abstraktion over) simple, fysiske kredse
f.eks. ca. $0V \approx 0$, ca. $5V \approx 1$



A	X
0	1
1	0

I Prolog:

`not(0,1).`

`not(1,0).`

Test program

Flere simple komponenter

`and(0, 0, 0).`
`and(0, 1, 0).`
`and(1, 0, 0).`
`and(1, 1, 1).`



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

`xor(0, 0, 0).`
`xor(0, 1, 1).`
`xor(1, 0, 1).`
`xor(1, 1, 0).`

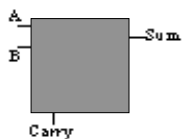
`or(0, 0, 0).`
`or(0, 1, 1).`
`or(1, 0, 1).`
`or(1, 1, 1).`

Test program

At bygge logiske kredse af komponenter

Eksempel: En »halfadder«
Lægge to bits A og B sammen:

Test program

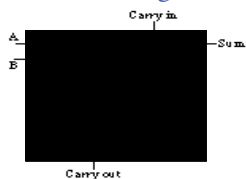


```
halfadder(A, B, Carry, Sum):-
```

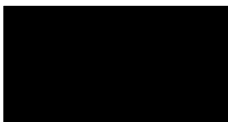


Bemærk: Analogi mellem Prolog-variabel og elektrisk leder

En full-adder, nu med gammel mente



```
fulladder(A, B, Carryin, Sum, Carryout):-
```



Test program

Prædikater i Prolog (ofte) »reversible«

Hvad får vi som output når vi fulladder input 0,1,1?

```
?- fulladder(0,1,1,S,C).  
C = 1, S = 0 ?
```

Hvilket input til fulladder giver output = 0, 1?

```
?- fulladder(X,Y,Z,0,1).  
X = 0, Y = 1, Z = 1 ? ;  
X = 1, Y = 0, Z = 1 ? ;  
X = 1, Y = 1, Z = 0 ? ;  
no
```

Reversibel: der er ingen skellen mellem input- og output-variable!
Andet ord for reversibel: Relationel

Test program

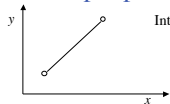
Begreber i Prolog, udvide med *strukturer*

- prædikater: **parent**
 - beskriver relation
 - bestemt ved fakta, regler, tilsammen klausuler
- atomer: **tom, bob, x, y**
- variable: **x, y, Tom**
- mål: **parent(A, a)**
- Argumenter til prædikater/mål kan også være *strukturer*:

```
point(1,1)
linje(point(1,1),point(3,3))
```

OBS: Ligner prædikater m. argument, men...

Et eksempel på anvendelse af strukturer



Intuitiv tolkning af struktur:

```
linje(point(1,1),point(3,3))
```

Hvor mange linjer Java-kode skal der til for at opnå samme funktionalitet????

Dette er et program:

```
vertical( seg(point(X,Y), point(X,Y1)) ).
horizontal( seg(point(X,Y), point(X1,Y)) ).
```

Forespørg:

```
?- vertical( seg(point(1,1),point(2,Y)) ).
no
?- horizontal( seg(point(1,1),point(2,Y)) ).
Y = 1 ?
```

Test program

Repetition af grundbegreber i Prologs syntaks

<term> ::= <atom> | <tal> | <variabel> | <struktur>

<atom> ::= a | b | muuh | 'X' | '!' | ...

<tal> ::= 0 | 1 | -1 | 4.13 | ...

<variabel> ::= X | A | Muuh | _a | _l | ...

<struktur> ::= <atom>(<term>, ..., <term>)

<mål> ::= <atom> | <struktur>

<klausul> ::= <faktum> | <regel>

<faktum> ::= <mål> .

<regel> ::= <mål> :- <mål>, ..., <mål> .

<program> ::= <klausul> ... <klausul>

<forespørgsel> ::= <mål>, ..., <mål> .

Deklarativ semantik; princippet

Klausul

$p(x) :- q(x), r(x).$
forstås som logisk formel
 $\exists X (p(X) \wedge q(X) \wedge r(X))$

Definition. Betydning af program P =

$\text{Model}(P) = \{ M \mid M \text{ grundet mål, } P \models M \}$
hvor "grundet" = variabelfri og \models er "logisk konsekvens"

Definition. En substitution er en afbildning fra en mgd. variable til termer

Definition. Et korrekt svar på forespørgsel M_1, \dots, M_n til program P er en substitution S, så
 $\{S(M_1), \dots, S(M_n)\} \subseteq \text{Model}(P)$

Obs: Forespørgsel $?- p(x)$ kan forstås $\exists X p(X)$

Procedural semantik, grundbegreber

Udførelsesrækkefølge ved eksempel

```
run:- write(runclause1), run1, run2.  
run:- write(runclause2), run3.  
run1:- write(run1).  
run2:- write(run2).  
run3:- write(run3).
```

Test program

Unification

$a = b$
 $X = a, a = Y$
 $f(a, X) = f(a, b) \dots$

Nu til det pragmatiske ...

Lister \approx blot specielle termer m. syntaktisk sukker

```
?- write([1,2,3,4,5,6]).  
[1,2,3,4,5,6]  
?- write_canonical([1,2,3,4,5,6]).  
'.'(1, '.'(2, '.'(3, '.'(4, '.'(5, '.'(6, [])))))  
?- [1,2,3,4,5,6] = [Hovede | Hale].  
Hale = [2,3,4,5,6],  
Hovede = 1
```

To ting i samme unification,
slet ingen fjøllede metodekald!

Standardprædikater til lister...

Husk i SICStus Prolog at skrive dette i din fil:

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

Eksempler

```
?- append([a,b],[c,d], L).
```

```
L = [a,b,c,d]
```

```
?- append(X,Y,[a,b,c]).
```

```
X = [], Y = [a,b,c] ? ;
```

```
X = [a], Y = [b,c] ? ;
```

```
X = [a,b], Y = [c] ? ;
```

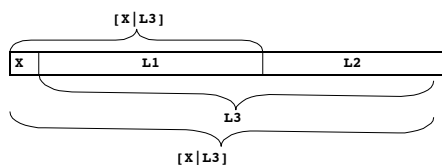
```
X = [a,b,c], Y = [] ? ;
```

OBS: Bratko kalder i sin bog »append« for »conco

Hvordan skrives append?

```
append([], L, L).
```

```
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```



Listemedlemskab... m.v.!

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

```
yes
```

... men også listekonstruktør:

```
?- member(a,L), member(b,L), member(c,L).
```

```
L = [a,b,c|_A]
```

Implementation af member

```
member(X, [X | _] ).
```

```
member(X, [_|L]):- member(X,L).
```

Operatorer, brugerudvidelig syntaks

```
:- op(700, xfx, sparker).
manden sparker hunden.
:- op(700, xfx, bider).
X bider Y :- Y sparker X.
```

Vigtigt: Kun syntaktisk sukker, ingen ny semantik

```
?- current_op(X, Y, Z).
X = 1200, Y = xfx, Z = :- ? ;
X = 1200, Y = xfx, Z = --> ? ;
...
X = 1000, Y = xfy, Z = ', ' ? ;
...
X = 500, Y = yfx, Z = + ? ;
...
X = 400, Y = yfx, Z = * ? ;
```

Eksempel på program med operatorer

```
:- op(700, xfx, er).
:- op(100, fx, [en,et]).
en mand er et menneske.
en kvinde er et menneske.
et menneske er et dyr.
en ko er et dyr.
peter er en mand.
X er Z :- X er Y, Y er Z.
```

Test program

Aritmetik, et stedbarn i Prolog

```
?- X is 2 + 2 * 3.
X = 8 ?
?- X is 2 + Y * 3.
! instantiation error in argument 2 of is/2
! goal: _79 is 2+_73*3
```

Kontrol af backtracking, »!« (cut)

```
loen(S, 0):- student(S), !.  
loen(S, 100000).  
student(peter).
```

Test program

```
likes(mary, X):- snake(X), !, fail.  
likes(mary, X):- animal(X).
```

```
snake(boa).  
animal(X):- snake(X).  
cow(boa).  
animal(X):- cow(X).  
animal(animal).
```

Negation-as-failure (et let bøjet søm)

Antag program

```
p(a).
```

Afprøve

```
?- \+ p(a).
```

```
no
```

```
?- \+ p(b).
```

```
yes
```

Men tvivlsom semantik ved variable:

```
?- x = b, \+ p(x).
```

```
x = b ?
```

```
yes
```

```
?- \+ p(x), x = b.
```

```
no
```

Andre faciliteter i Prolog

- Indbyggede prædikater til i/o
- Ekstralogiske prædikater, var(X), etc.
- assert og retract, selvmodificerende programmer
- Biblioteker til alt mulig, interface til Java og C
- Få kontrolværktøjer, »!« og et par stykker mere

Ellers har I set det hele!

- Minimalistisk kerne med enkel syntaks og semantik

Typiske anvendelser

- Kunstig intelligens, logik-baserede systemer
- Planlægning, ...
- Alle former for symbolske beregninger
- Sproganalyse

Sammenligning med Java (o.lign.)

	Prolog	Java
Udtrykskraft	Stor	Ringe
Programmeringshastighed	Høj	Meget sløv
Udførelseshastighed	Høj (rette opg.!)/Lav	Burde være høj
Fleksibilitet	Stor	Ringe
Genbrug	Nej (og ja)	Ja (og nej)
Eksperimentel progra.	Ja	Nej
Grafisk grænseflade	Ikke rigtigt	JA!
Sikkerhed/robusthed	Nej	Ja (nej)
Vejret i morgen?	NEJ	(Ja)
Dansk->Kinesisk	Ja	(Nej)
	<i>Laboratoriesprog</i>	<i>Programmer i drift</i>

Vigtige udvidelser til Prolog

- DCG, definite clause grammars
 - Grammatiknotation indbygget i Prolog
- Constraint-solvere
 - f.eks. ægte »logiske« regning med tal
 - rentesregning baglæns
 - SICStus har en lang række constraint-solvere indbygget
- Constraint Handling Rules
 - logisk sprog til at programmere constraint-solvere
 - generel bottom-up beregner, hvor Prolog er top-down...
- CGR Grammars
 - <http://www.ruc.dk/~henning/chrp>

Anvendelser i undervisning om programmeringssprog på RUC

- Semantik for sekventielt maskinsprog ved fortolker
- Semantik af **Se den?** d fortolker
- Compiler fra »While« til maskinsprog
- Opgave som har været stillet: **Se den?**
Skriv typechecker og fortolker for sprog med rekursion; test m. rekursiv quicksort.

SLUT?

Simple machine language:

Syntax and intuitive semantic by example:

```
[ push(2),
  store(t),
  7, fetch(t),
  push(2),
  add,
  store(x),
  fetch(t),
  push(1),
  subtract,
  store(t),
  fetch(t),
  push(0),
  equal,
  n_jump(7)]
```

Defining interpreter:

```
prog(program, final_stack, final_store)
  ~ from empty stack, all var's set to 0
sequence(
  RestProgram,    ~ to be interpreted
  WholeProgram,  ~ meaning of labels
  CurrentStack,  } when RestProgram
  CurrentStore,  } is started
  FinalStack,
  FinalStore).
```

Representation of stacks and stores:

```
[3, 4, 5]
[(a=7), (b=8)]
```

Defining interpreter. Little tool box:

```
! store(VariableID, Value, Store, UpdatedStore)

store(V, X, [], [(V=X)]).
store(V, X, [(V=_) | L], [(V=X) | L]) :- !.
store(V, X, [Bind|LO], [Bind|LI]) :-
  store(V, X, LO, LI).
```

Defining interpreter. Little tool box:

```
⌘ store(VariableID, Value, Store, UpdatedStore)

store(V,X,[],[(V=X)]).
store(V,X,[(V=_)|L],[(V=X)|L]):- !.
store(V,X,[Bind|LO],[Bind|L1]):-
  store(V,X,LO,L1).

⌘ fetch(VariableID, Value, Store)

fetch(Var,X,L):-
  member((Var=X),L) -> true
  ;
  X = 0.

⌘ no helpers needed for stacks
```

Defining interpreter/principle

```
⌘ prog(Program, FinalStack, FinalStore)

prog(Prog, FinalStack, FinalStore):-
  sequence(Prog,Prog,[],[],FinalStack,FinalStore).

⌘ sequence(RestProg, WholeProg, CurrentStack, CurrentStore, FinalStack, FinalStore).
⌘ Principle: One rule for each instruction; continue with tail recursive call, e.g.:
```

Defining interpreter/principle

```
⌘ prog(Program, FinalStack, FinalStore)

prog(Prog, FinalStack, FinalStore):-
  sequence(Prog,Prog,[],[],FinalStack,FinalStore).

⌘ sequence(RestProg, WholeProg, CurrentStack, CurrentStore, FinalStack, FinalStore).
⌘ Principle: One rule for each instruction; continue with tail recursive call, e.g.:
```

```
sequence([push(N)|Rest], Prog, S0, LO, S1, L1):-
  sequence(Rest, Prog, [N|S0], LO, S1, L1).
```

Defining interpreter/principle

```
† prog(Program, FinalStack, FinalStore)

prog(Prog, FinalStack, FinalStore):-
  sequence(Prog,Prog,[],[],FinalStack,FinalStore).

† sequence(ResProg,WholeProg,CurrentStack,CurrentStore,FinalStack,FinalStore).
† Principle: One rule for each instruction; continue with tail recursive call, e.g.:

sequence([push(N)|Rest], Prog, S0, L0, S1, L1):-
  sequence(Rest, Prog, [N|S0], L0, S1, L1).

sequence([fetch(Var)|Rest], Prog, S0, L0, S1, L1):-
  fetch(Var,X,L0), sequence(Rest, Prog, [X|S0], L0, S1, L1).
```

Defining interpreter/principle

```
† prog(Program, FinalStack, FinalStore)

prog(Prog, FinalStack, FinalStore):-
  sequence(Prog,Prog,[],[],FinalStack,FinalStore).

† sequence(ResProg,WholeProg,CurrentStack,CurrentStore,FinalStack,FinalStore).
† Principle: One rule for each instruction; continue with tail recursive call, e.g.:

sequence([push(N)|Rest], Prog, S0, L0, S1, L1):-
  sequence(Rest, Prog, [N|S0], L0, S1, L1).

sequence([fetch(Var)|Rest], Prog, S0, L0, S1, L1):-
  fetch(Var,X,L0), sequence(Rest, Prog, [X|S0], L0, S1, L1).

sequence([add|Rest], Prog, [X,Y|S0], L0, S1, L1):-
  YplusX is Y + X, sequence(Rest, Prog, [YplusX|S0], L0, S1, L1).
```

Defining interpreter/jumps and labels

```
sequence([jump(E)|_], P, S0, L0, S1, L1):-
  append(_, [E|Continuation], P),
  sequence(Continuation, P, S0, L0, S1, L1).

† The trick: P = [..., E | Continuation]
```

Defining interpreter/jumps and labels

```
sequence([jump(E)|_], P, SO, LO, S1, L1):-  
    append(_, [E|Continuation], P),  
    sequence(Continuation, P, SO, LO, S1, L1).  
  
% The trick: P = [..., E | Continuation]  
  
sequence([in_jump(E)|_], P, [0|SO], LO, S1, L1):-  
    append(_, [E|Continuation], P),  
    sequence(Continuation, P, SO, LO, S1, L1).  
  
sequence([in_jump(_)|Rest], P, [1|SO], LO, S1, L1):-  
    sequence(Rest, P, SO, LO, S1, L1).
```

Defining interpreter/jumps and labels

```
sequence([jump(E)|_], P, SO, LO, S1, L1):-  
    append(_, [E|Continuation], P),  
    sequence(Continuation, P, SO, LO, S1, L1).  
  
% The trick: P = [..., E | Continuation]  
  
sequence([in_jump(E)|_], P, [0|SO], LO, S1, L1):-  
    append(_, [E|Continuation], P),  
    sequence(Continuation, P, SO, LO, S1, L1).  
  
sequence([in_jump(_)|Rest], P, [1|SO], LO, S1, L1):-  
    sequence(Rest, P, SO, LO, S1, L1).  
  
sequence([Label|Rest], P, SO, LO, S1, L1):-  
    integer(Label),  
    sequence(Rest, P, SO, LO, S1, L1).
```

Defining interpreter

- The rule than beginners always forgets and which makes the interpreter say "no":

Defining interpreter/the termination rule

- The rule that beginners always forget and which makes the interpreter say "no":

```
sequence([], _, S, L, S, L).
```

Defining interpreter/testing it

```
?- prog([push(3), push(2), subtract], S, L).  
L = [], S = [1] ?
```

```
?-
```

Defining interpreter/testing it

```
?- prog([push(3), push(2), subtract], S, L).  
L = [], S = [1] ?
```

```
?- prog([push(3), push(2), subtract, store(a)], S, L).  
L = [(a=1)], S = [1] ?
```

Defining interpreter/testing it

```
?- prog( [push(2),
        store(t),
        7, fetch(x),
        push(2),
        add,
        store(x),
        fetch(t),
        push(1),
        subtract,
        store(t),
        fetch(t),
        push(0),
        equal,
        n_jump(7)], S, L).

L = [t=0,x=4], S = [] ?
```

How many lines of Java do you have to write for this interpreter???

Defining interpreter

- Want a replay with a tracer for our machine language?

Defining interpreter/testing it with trace

- Want a replay with a tracer for our machine language?
- We make interpreter into tracer as follows:

Add this rule *before* any other **sequence** rule:

```
sequence([Inst|_|_...|_...|_...]):-
    write(Inst), write(' '), fail.

?- prog([push(3),push(2),subtract],S,L).
push(3) push(2) subtract
L = [], S = [1] ?
```

Defining interpreter/testing it with trace

```
?- prog([push(2),
        store(t),
        7, fetch(x),
        push(2),
        add,
        store(x),
        fetch(t),
        push(1),
        subtract,
        store(t),
        fetch(t),
        push(0),
        equal,
        n_jump(7)], S, L).
```

Defining interpreter/testing it with trace

```
?- prog([push(2),
        store(t),
        7, fetch(x),
        push(2),
        add,
        store(x),
        fetch(t),
        push(1),
        subtract,
        store(t),
        fetch(t),
        push(0),
        equal,
        n_jump(7)], S, L).
push(2) store(t) 7 fetch(x) push(2) add store(x) fetch(t)
push(1) subtract store(t) fetch(t) push(0) equal n_jump(7)
fetch(x) push(2) add store(x) fetch(t) push(1) subtract
store(t) fetch(t) push(0) equal n_jump(7)
L = [t=0,x=4], S = [] ?
```

Possible exercises for students

- Add the following instructions to the interpreter:
`jump_subroutine(ToLabel, ReturnLabel)`
– push ReturnLabel on stack; jump to ToLabel
`return`
– take top stack element as address and jump to it
and test it!

Possible exercises for students

- Add the following instructions to the interpreter:

```
jump_subroutine(ToLabel, ReturnLabel )  
  - push ReturnLabel on stack; jump to ToLabel  
return  
  - take top stack element as address and jump to it
```

and test it!

- Design and implement instructions for subroutines with parameters.

Possible exercises for students

- Add the following instructions to the interpreter:

```
jump_subroutine(ToLabel, ReturnLabel )  
  - push ReturnLabel on stack; jump to ToLabel  
return  
  - take top stack element as address and jump to it
```

and test it!

- Design and implement instructions for subroutines with parameters.

≈ I.e., students play with formal language specifications in an easy-to-use, executable environment!

Se eksempel på compiler?

SLUT?

Continuing the course: Compiling

- Abstract syntax trees of source language (while-programs) written directly as Prolog terms, e.g.

```
( a:= 221 ; b:= 493 ;  
  while( a =\= b,  
    if( a > b, a:= a-b, b:= b-a))
```

Continuing the course: Compiling

- Abstract syntax trees of source language (while-programs) written directly as Prolog terms, e.g.

```
( a:= 221 ; b:= 493 ;  
  while( a =\= b,  
    if( a > b, a:= a-b, b:= b-a))
```
- Requires an operator def. (others are predefined)

```
:- op(800,xfx, :=).
```

Continuing the course: Compiling

- Abstract syntax trees of source language (while-programs) written directly as Prolog terms, e.g.

```
( a:= 221 ; b:= 493 ;  
  while( a =\= b,  
    if( a > b, a:= a-b, b:= b-a))
```
- Requires an operator def. (others are predefined)

```
:- op(800,xfx, :=).
```
- We introduce a small toolbox of two help predicates:

```
new_label(Var) - instantiates Var to unused label  
new notation for easy concatenation of instructions and code sequences
```

Compiler tool box: New labels

```
new_label(N):-  
  retract( next_label(N) ),  
  N1 is N+1,  
  asserta( next_label(N1)).  
  
:- dynamic next_label/1.  
  
next_label(0).
```

Compiler: While-prog's -> machine code

- A predicate for each syntactic category *cat*:
`cat(SyntaxTree, CompiledCode)`

Compiler: While-prog's -> machine code

- A predicate for each syntactic category *cat*:
`cat(SyntaxTree, CompiledCode)`

- One rule for each syntax rule, e.g.,

```
program(P, C):- statement(P, C).
```

Compiler: While-prog's -> machine code

- A predicate for each syntactic category *cat*:
`cat(SyntaxTree, CompiledCode)`

- One rule for each syntax rule, e.g.,

```
program(P, C):- statement(P, C).
```

```
statement( (S1 ; S2), C):-  
    statement(S1, C1), statement(S2, C2),  
    C <- C1 + C2.
```

Compiler: While-prog's -> machine code

- A predicate for each syntactic category *cat*:
cat(*SyntaxTree*, *CompiledCode*)
- One rule for each syntax rule, e.g.,

```
program(P, C):- statement(P, C).  
  
statement( (S1 ; S2), C):-  
    statement(S1, C1), statement(S2, C2),  
    C <- C1 + C2.  
  
statement( (Var := Expr), C):-  
    expression( Expr, C1),  
    C <- C1 + store(Var).
```

Compiler: While-prog's -> machine code

- Rules for expressions, e.g.

```
expression( Number, C):-  
    integer(Number),  
    C <- push(Number).
```

Compiler: While-prog's -> machine code

- Rules for expressions, e.g.

```
expression( Number, C):-  
    integer(Number),  
    C <- push(Number).  
  
expression( Variable, C):-  
    atom(Variable),  
    C <- fetch(Variable).
```

Compiler: While-prog's -> machine code

- Rules for expressions, e.g.

```
expression( Number, C):-  
    integer(Number),  
    C <- push(Number).  
  
expression( Variable, C):-  
    atom(Variable),  
    C <- fetch(Variable).  
  
expression( (Expr1 + Expr2), C):-  
    expression( Expr1, C1), expression( Expr2, C2),  
    C <- C1 + C2 + add.
```

Compiler: While-prog's -> machine code

- Rules for structured statements, if:

```
statement( if( Cond, Statement1, Statement2), C):-  
    condition(Cond, Ccond),  
    statement(Statement1, C1), statement(Statement2, C2),  
    new_label(L2), new_label(L_end),  
    C <- Ccond +  
        n_jump(L2) +  
        C1 +  
        jump(L_end) +  
        L2 + C2 +  
        L_end.
```

Compiler: While-prog's -> machine code

- Rules for structured statements, while:

```
statement( while( Cond, Statement), C):-  
    condition(Cond, Ccond),  
    statement( Statement, C1),  
    new_label(L_start), new_label(L_end),  
    C <- L_start + Ccond +  
        n_jump(L_end) +  
        C1 +  
        jump(L_start) +  
        L_end.
```

Testing the compiler

```
?-program(  
  a:=221;b:= 493;  
  while(a=\b,  
    if(a>b, a:=a-b, b:=b-a  
  )), C).
```

Testing the compiler

```
?-program(  
  a:=221;b:= 493;  
  while(a=\b,  
    if(a>b, a:=a-b, b:=b-a  
  )), C).  
  
C = [  
  push(221),  
  store(a),  
  push(493),  
  store(b),  
  14, fetch(a),  
  fetch(b),  
  not_equal,  
  n_jump(15),  
  fetch(a),  
  fetch(b),  
  greater,  
  n_jump(12),  
  fetch(a),  
  fetch(b),  
  subtract,  
  store(a),  
  jump(13),  
  12, fetch(b),  
  fetch(a),  
  subtract,  
  store(b),  
  13, jump(14),  
  15]
```

Testing the compiler and running code

```
?-program(  
  a:=221;b:= 493;  
  while(a=\b,  
    if(a>b, a:=a-b, b:=b-a  
  )), C),  
prog(C, S, L).
```

Testing the compiler and running code

```
?-program(  
  a:=221;b:= 493;  
  while(a!=b,  
    if(a>b, a:=a-b, b:=b-a  
    )), C),  
prog( C, S, L).
```

```
C = [push(221),  
     store(a),  
     ...]  
L = [a=17,b=17],  
S = [] ?
```

*How many lines of Java do
you have to write for this
compiler???*



