

# Meta-interpreters

- **Interpreted languages like Prolog can easily treat program code as data**
  - can use ‘clause’ to look at program code
  - assert, retract, abolish to create/remove code
  - =.., name, etc to look at terms themselves
- **meta-programming: a program which treats program code as data**
  - the program code executed can be a different language than the code the meta-program is written in
  - or it can be the same language
- **meta-interpreter: a meta-program that executes a program(“source program”) , possibly written in that language itself**
  - can be used to enhance your language execution
  - can create prototype languages as well
- **Prolog is extremely useful for meta-programming**

# User-defined operators

- **op(Precedence, Code, Op).**
  - **Precedence:** numeric code indicating binding strength of operator
    - » smaller number = higher binding
    - » used when parentheses are missing
  - **Code: position and associativity**
    - » prefix: fx, fy
    - » postfix: xf, yf
    - » infix: xfx, xfy, yfx, yfy
    - » “x”: argument must have ops of lower precedence
    - » “y”: argument must have ops of equal or lower precedence
  - **Op:** characters for operator

eg.

?- op(470, fy, 'not').

?- op(475, xfy, 'and').

?- op(480, xfy, 'or').

permits: not a and b or not c --> ((not a) and b) or (not c)

# Meta-interpreter

- **Example: a meta-interpreter for pure Prolog**

```
solve( true ) :- !.  
solve( not P ) :- !, \+ solve(P).  
solve( (P, Q) ) :- !, solve(P), solve(Q).  
solve( P ) :- clause(P, Body), solve(Body).
```

- **each clause catches one possible case**
- **'true' is used to represent the termination of a branch of execution**
  - note that `clause(P, Body)` returns `P=parent(mary, bob), B=true` for the fact: `parent(mary, bob)`.
- **the case for (P, Q) breaks up multiple goals; “,” is simply a builtin infix operator**
  - eg. `solve((A, B, C, D)) = solve(P, Q) --> P = A, Q = (B, C, D)` etc
- **the cuts are needed so that, during backtracking, true, not P, and (P,Q) won't be executed by the final case that uses clause**

# Meta-interpreters

- **The bulk of the work is done by ‘clause’**
  - clause actually does the unification of the current goal P with a clause
  - upon backtracking, clause will backtrack and unify P with the next clause
  - Note that, when P is unified with the clause Q :- B, the variable substitutions obtained via P=Q are automatically applied to B
- **The interesting part of meta-interpreters is that you can alter the language behaviour**
- **Example: a meta-interpreter that selects goals from right-to-left**

```
solve( true ) :- !.  
solve( not P ) :- !, \+ solve(P).  
solve( (P, Q) ) :- !, solve(Q), solve(P).  
solve( P ) :- clause(P, Body), solve(Body).
```

# Meta-interpreters

- **Example: A meta-interpreter that doesn't do any backtracking**

```
solve( true ) :- !.  
solve( not P ) :- !, \+ solve(P).  
solve( (P, Q) ) :- !, solve(P), solve(Q).  
solve( P ) :- clause(P, Body), !, solve(Body).
```

- **Example: adding some builtin predicates (no longer pure Prolog)**

```
solve( true ) :- !.  
solve( not P ) :- !, \+ solve(P).  
solve( (P, Q) ) :- !, solve(P), solve(Q).  
solve( write(X) ) :- !, write(X).  
solve( read(X) ) :- !, read(X).  
solve( P ) :- clause(P, Body), solve(Body).
```

# Meta-interpreters

- Example: print out a trace of your execution

```
solve( true ) :- !.  
solve( not(P) ) :- !, \+ solve(P).  
solve( (P, Q) ) :- !, solve(P), solve(Q).  
solve( P ) :-  
    (write('calling '), write(P) ; write(P), write('fails'), nl, !, fail),  
    clause(P, Body),  
    write('...succeeds'), nl,  
    solve(Body).
```

## Meta-interpreters

- **By creating new operators, you can even change the syntax of the source program -- great for creating a new language**
- **Example: a new syntax for Prolog**

```
?- op(700, xfy, and).
```

```
?- op(800, xfx, if).
```

```
solve( true ) :- !.
```

```
solve( not P ) :- !, \+ solve(P).
```

```
solve( P and Q ) :- !, solve(P), solve(Q).
```

```
solve( P ) :- P if Body, solve(Body).
```

```
grandmother(X, Y) if mother(X, Z) and mother(Z, Y).
```

- **Note how 'if' is just another predicate name:**
  - same as: `if(grandmother(X,Y), (mother(X,Z), mother(Z,Y))).`
  - we essentially let the meta-level Prolog do the backtracking for us!

# Metainterpreters

- **Example: a metainterpreter that constructs a logical proof tree**

`solve(true, true) :- !.`

`solve(not P, (not Proof)) :- !, \+ solve(P, Proof).`

`solve((P, Q), (ProofP, ProofQ)) :- !, solve(P, ProofP), solve(Q, ProofQ).`

`solve(P, (P <== ProofP)) :- clause(P, Body), solve(Body, ProofP).`

- **arg 2 contains pattern of proof**
- **Would be nice to print it out in a legible form...**

# Metainterps

```
prettyprint(E) :- prettyprint2(E, 0).
```

```
prettyprint2(not A, Indent) :-
```

```
    !,
```

```
    nl, tab(Indent),
```

```
    write('NOT '), prettyprint2(A, Indent).
```

```
prettyprint2((A,B), Indent) :-
```

```
    !,
```

```
    prettyprint2(A, Indent),
```

```
    nl, tab(Indent), write('AND'),
```

```
    prettyprint2(B, Indent).
```

```
prettyprint2(A <== true, Indent) :-
```

```
    !,
```

```
    nl, tab(Indent),
```

```
    write(A),
```

```
    write(' <== TRUE').
```

## Prettyprint (cont)

```
prettyprint2(A <== P, Indent) :-
```

```
    !,
```

```
    nl, tab(Indent),
```

```
    write(A), write(' <== '),
```

```
    Indent2 is Indent+3,
```

```
    prettyprint2(P, Indent2).
```

```
prettyprint2(A, Indent) :-
```

```
    nl, tab(Indent),
```

```
    write(A).
```

## Another metainterpreter: “Pascal”

- This example interprets a Pascal-like language.
- Grammar of language (Backus-Naur Form, or BNF):

$$E ::= V := A \mid E;E \mid \text{if}(B, E, E) \mid \text{while}(B, E)$$
$$A ::= \text{var} \mid \text{const} \mid A+A \mid A-A \mid A*A$$
$$B ::= \text{true} \mid \text{false} \mid A>A \mid A=A \mid A>=A$$

- Implement via operators (“:=“, “;”) and structures:  
if(B, E, F) and while(B, E)
- Memory: list of variable/value pairs:
- [(a, 0), (flag, 1), (value, 2014), ...]

## Pascal metainterpreter: “interp3”

- Idea: each statement in language affects the state of memory
- To interpret a program, determine how each statement changes memory
- This is an “operational semantics” approach.

eg. memory before: [(a, 1), (b, 2), (c, 5)]

    a := b\*c

    memory after: [(a, 10), (b, 2), (c, 5)]

eg. E1;E2   where initial memory = Mem1

    interpret(E1, Mem1) to generate Mem2

    then interpret (E2, Mem2) to generate MemFinal

eg. while(B, E): let init memory = Mem1

    if B = true then interpret(E, Mem1) to create Mem2

        and interpret(while(B, E), Mem2) to create MemF