Introduction to Programming Languages
Logic Programming

Bernard Sufrin

Version of 17th February, 2000

The Study of Programming Languages

- ⇨ Syntax – about structure (form) of phrases
  - ⇨ Concrete Syntax – written form (strings)
  - ⇨ Abstract Syntax – essential structure (trees)
- ⇨ **Semantics – about meaning**
- ⇨ Pragmatics – about implementation methods, useability, reliability, ...

Why bother to study programming languages systematically?

- ⇨ When we use a language we don't want to be a victim of its quirks.
- ⇨ When we set out to design a language we need to know what choices we can make. ➥

Some important questions

Declarative *versus* Imperative?

⇨ Declarative – "What"

Programmer describes problems, and/or the relationships between problems and solutions.

⇨ Imperative – "How"

Programmer gives instructions which indicate precisely how to compute solutions to problems.

In reality there is a spectrum.....

➥

Questions concerning computational content

- ⇨ *Are operations naturally encapsulated with data*?
  Procedure-Oriented: (*e.g.* Pascal, C, Modula II, CAML)
  Object-Oriented: (*e.g.* Java, C^{++}, Oberon, Modula III, OCAML)
- ⇨ *Can operations be treated as data*?
  First-order: (*e.g.* Obj, FP, Tcl)
  Higher-order: (*e.g.* (O)CAML, Haskell)
- ⇨ *Are expressions evaluated at most once & when needed*?
  Strict: (*e.g.* (O)CAML)
  Lazy: (*e.g.* Haskell, Prolog)
- ⇨ *Do non-local variables acquire their meaning at definition time or at invocation time*?
  Static Binding: (*e.g.* C, C^{++}, Scheme, Haskell)
  Dynamic Binding: (*e.g.* Postscript, TeX, Lisp)
- ⇨ *Does computation implicitly involve search*?
  Functional: (*e.g.* Haskell, Lisp, Scheme, FP)
  Logical/Relational: (*e.g.* Prolog, Mercury, Andorra)

Questions concerning modularity

- ⇨ *How straightforward is "implementation hiding"*?
  Straightforward: (*e.g.* Haskell, Modula, Java, (O)CAML)
  Awkward: (*e.g.* C, Postscript, Lisp, Scheme)
- ⇨ *How does the language support code re-use?*
  Inheritance: classes can be extended (*e.g.* Java, OCAML, Eiffel)
  Genericity: modules can have parameters (*e.g.* OCAML, Eiffel)

Questions concerning types

- ⇨ *Is type consistency checked as late as possible*?
  Static Typing: (*e.g.* Java, Eiffel, OCAML, CAML, Pascal)
  Dynamic Typing: (*e.g.* Scheme, Lisp)
- ⇨ *How does the language support the definition of operations with type-dependent meanings?*
  - ⇨ Ad-hoc Polymorphism
  - ⇨ Parametric Polymorphism
  - ⇨ Type Classes
  - ⇨ Abstract (Virtual) Class Members

Course Plan:

1. Logic Programming
2. Interpreters and Operational Semantics
3. Understanding Type Systems and Type Inference
4. Understanding Modules and Classes
5. From lambda calculus to functional programming
   (2 supplementary lectures: 1999/2000 only)

Background reading on Prolog (in order of preference):

- ⇨ Leon Sterling and Ehud Shapiro: the Art of Prolog *MIT Press, 1994*. (Especially Chapters 1-3, 6-9). This book is encyclopædic in its coverage.
- ⇨ Michael Spivey: An Introduction to Logic Programming through Prolog. *Prentice-Hall International, 1996*. (A lucid and helpful introduction to principles and semantics of Prolog, together with an implementation description).
- ⇨ Ivan Bratko: Prolog Programming for Artificial Intelligence. *Addison-Wesley, 1995*

Background reading for the rest of the course:

- ⇨ Ravi Sethi: Programming Languages, Concepts and Constructs. *Addison Wesley, 1990* (An underappreciated book.)

  and perhaps
- ⇨ Carlo Ghezzi, Mehdi Jazayeri: Programming Language Concepts. *John Wiley, 1997* (A little too anecdotal for my taste, but at least it is still in print.)

  You may also find useful – though they don't cover exactly the same material
- ⇨ Course notes written by Oege de Moor for "Principles of Programming Languages, Part II" – an ancestor of this course.

Declarative Programming

Describing problems and solutions

⇨ As relations (logic programming)

```
greataunt(X, Y)  if
   exists Z . grandparent(X, Z) and sister(Z, Y).

isrot(US, VS) if
   exists XS, YS . XS++YS=US and YS++XS=VS
```

⇨ As functions (functional programming)

```
greataunt(x, y)  = y ‘elem‘ map sister (grandparents x)

isrotation(u, v) = v ‘elem‘ rots u

rots  xs     = zipWith (++) (tails xs) (inits xs)
inits []     = [[]]
inits (x:xs) = []: map (x:) (inits xs)
tails []     = [[]]
tails (x:xs) = (x:xs): tails xs
```

Logic Programming

⇨ **Prolog** was the first Logic-Programming Language, and is the most widely known and implemented, though not completely standardized.

⇨ Logic Programming is based on computation by deduction ➥

⇨ Relations can be specified so that automated deduction is possible.

⇨ Eg: `append(XS, YS, ZS)` meaning $XS +\!\!+ YS = ZS$

```
append(nil,  YS, YS)    :- .
append(X:XS, YS, X:ZS) :- append(XS, YS, ZS).
```

➥

⇨ Eg: `isrot(XS, YS)` meaning $\exists US, VS \cdot \left(\begin{array}{lcll} XS & = & US +\!\!+ VS & \wedge \\ YS & = & VS +\!\!+ US & \end{array} \right)$

```
isrot(XS, YS) :- append(US, VS, XS), append(VS, US, YS).
```

➪ Proving a fact about append by deduction is straightforward

```
    append(1:2:nil, 3:nil, 1:2:3:nil)
⇐     {append.2}
    append(2:nil, 3:nil, 2:3:nil)
⇐     {append.2}
    append(nil, 3:nil, 3:nil)
⇐     {append.1}
    true
```

➪ Solving a problem with unknowns by deduction (I)

```
    append(1:2:nil, 3:nil, US)
⇔     {Preparing to use append.2  US = 1 : ZS'}
    append(1:2:nil, 3:nil, 1:ZS')
⇐     {append.2}
    append(2:nil, 3:nil, ZS')
⇔     {Preparing to use append.2  ZS' = 2 : ZS''}
    append(2:nil, 3:nil, 2:ZS'')
⇐     {append.2}
    append(nil, 3:nil, ZS'')
⇔     {Preparing to use append.1  ZS'' = 3 : nil}
    append(nil, 3:nil, 3:nil)
⇐     {append.1}
    true
```

➪ The relevant substitutions are

$$\begin{aligned} US &= 1 : ZS' \\ ZS' &= 2 : ZS'' \\ ZS'' &= 3 : \texttt{nil} \end{aligned}$$

So

$$US = 1 : 2 : 3 : \texttt{nil}$$

⇨ Solving a problem with unknowns by deduction (II)

```
    append(XS, 3:nil, 1:2:3:nil)
⇔     {preparing to use append.2  XS = 1 : XS'}
    append(1:XS', 3:nil, 1:2:3:nil)
⇐     {append.2}
    append(XS', 3:nil, 2:3:nil)
⇔     {preparing to use append.2  XS' = 2 : XS''}
    append(2:XS'', 3:nil, 2:3:nil)
⇐     {append.2}
    append(XS'', 3:nil, 3:nil)
⇐     {append.1  XS'' = nil}
    append(nil, 3:nil, 3:nil)
⇐     {true}
```

⇨ The relevant substitutions are

$$\begin{aligned} XS &= 1 : XS' \\ XS' &= 2 : XS'' \\ XS'' &= \texttt{nil} \end{aligned}$$

So

$$XS = 1 : 2 : \texttt{nil}$$

⇨ We used append to calculate a prefix!

⇨ Some relations can be defined for use in both directions!

Notation (Logpro)

Variables: words beginning with uppercase letters:

```
Foo
Monty_Python
```

Primitive Values: numbers and strings:

```
123
-456
"wretched oaf!"
```

Constants: words beginning with lowercase letters:

```
strange
good_Vibrations
```

Terms: constants, constants followed by bracketed term-lists:

```
it_is_raining
it_is_raining()                    (This is the same term as above)
wet(Here)
embattled(primitive(Teacher), spaceCadet(Pupil), Laboratory)
```

Terms: lists of terms

```
X:Y:3:nil
X:XS
```

Formulae: as for terms, also negated formulae

```
not kidding
```

Terms and formulae may use algebraic notation, provided the syntax of the operators is declared (see the Logpro Manual).

```
X := E+F --> X := E'+F
```

Example of a dialogue with the Logpro system.
(User input in this colour)

Definition of the `parent` relation as a collection of facts

```
18 % logpro
logpro-0,999 (16:16 Thursday Dec 23 1999)
-- parent(sid,    bill)     :- .
-- parent(liz,    bill)     :- .
-- parent(sid,    leah)     :- .
-- parent(liz,    leah)     :- .
-- parent(leah,   winifred) :- .
-- parent(leah,   albert)   :- .
-- parent(bill,   gerald)   :- .
-- parent(bill,   sarah)    :- .
```

We can ask for a statement (a *ground formula* – with no variables) to be verified

```
-- parent(liz, leah)?
yes?                                <Return>
--
```

If the statement cannot be verified, the system prompts for input directly.

```
-- parent(foo, bar)?
--
```

We can ask for the substitutions which satisfy an existential problem:

```
-- parent(sid, X)?

X = bill          ?       <Return>
X = leah          ?       <Return>
```

If there is more than one solution substitution, they are shown in turn.

After each solution is shown the system prompts with "?". We reply "<Return>" to continue the search for solutions, and "." to stop the search for solutions.

```
-- parent(X, Y)?
Y = bill     X = sid  ? <Return>
Y = bill     X = liz  ? <Return>
Y = leah     X = sid  ? <Return>
Y = leah     X = liz  ? <Return>
Y = albert   X = leah ? .
```

We can compose conjunctions of queries

```
-- parent(sid, X),
   parent(X,   Y)?

Y = gerald   X = bill ?  <Return>
Y = sarah    X = bill ?  <Return>
Y = winifred X = leah ?  <Return>
Y = albert   X = leah ?  <Return>
```

New relations are defined by (collections of) clauses:

⇨ "X has grandparent Z if one of X's parent's, Y, has parent Z."

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

⇨ "X has mother Y if X's parent is Y and Y is female."

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

"the clause head" "the clause body"

⇨ "Sid, Bill, Albert, and Gerald are male."

```
male(sid)    :- .
male(bill)   :- .
male(albert) :- .
male(gerald) :- .
```

⇨ "X is responsible if X is the culprit, or X manages a responsible person, Y."

```
responsible(X) :- culprit(X).
responsible(X) :- manages(X, Y), responsible(Y).
```

(Some) relations may be used in more than one direction ➥

For example, append

```
-- append(nil,  YS, YS) :- .
-- append(X:XS, YS, X:ZS) :- append(XS, YS, ZS).
```

```
-- append(X, Y, 1:2:3:4:nil)?          "Find X and Y st. X ++ Y = 1:2:..."
Y = nil X = 1 : 2 : 3 : 4 : nil ?      <Return>
Y = 4 : nil X = 1 : 2 : 3 : nil ?      <Return>
Y = 3 : 4 : nil X = 1 : 2 : nil ?      <Return>
Y = 2 : 3 : 4 : nil X = 1 : nil ?      <Return>
Y = 1 : 2 : 3 : 4 : nil X = nil ?      <Return>
--
```

```
   append(1:2:nil, X, 1:2:3:4:nil)? "Find X st. [1,2] ++ X = [1,2,3,4]"
X = 3 : 4 : nil ?                   "Here it is." <Return>
-- append(1:2:nil, X, 1:3:nil)?     "Find X st. [1,2] ++ X = [1,3]"
--                                  "None: I'm ready for your next question."
```

⇨ If there are no variables in the query, the solution is shown as "Yes".

⇨ If there is more than one solution, they are shown in turn.

⇨ If there are no solutions, the system just prompts for another query.

Some definitions can be surprisingly simple: for example, list membership

```
member(X, XS) :- append(As, X:Bs, XS).

-- member(jim, jim:jam:nil).
Yes? <Return>


-- member(X, jim:jam:nil)
X = jam ? <Return>
X = jim ? <Return>
```

Note that the system delivers as many answers as there are ways of solving the query.

```
-- member(jim, jim:jam:jim:nil).
Yes? <Return>
Yes? <Return>
```

Having such redundancy sometimes causes an exponential increase in runtime for a logic program.

If a conjunction of n goals is solved, and each has one redundant solution, then the conjunction may generate 2^n solutions.

Logical Meaning of a Definition

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

$$\forall X, Z \cdot \left(\begin{array}{l} grandparent(X, Z) \Leftarrow \exists Y \cdot \left(\begin{array}{l} parent(X, Y) \\ \wedge \\ parent(Y, Z) \end{array} \right) \end{array} \right)$$

```
elem(X, XS) :- append(As, X:Bs, XS).
```

$$\forall X, XS \cdot \Big(\; elem(X, XS) \Leftarrow \exists As, Bs \cdot append(As, X : Bs, XS) \; \Big)$$

```
append(nil,  YS, YS).
append(X:XS, YS, X:ZS) :- append(XS, YS, ZS).
```

$$\begin{array}{l} \forall \, YS \cdot append(nil, YS, YS) \\ \wedge \\ \forall \quad X, XS, YS, ZS \cdot \\ \quad \Big(\; append(X : XS, YS, X : ZS) \Leftarrow append(XS, YS, ZS) \; \Big) \end{array}$$

⇨ The meaning of "meaning" is not completely straightforward.

⇨ Logical meaning and operational meaning differ in Prolog.

Example: consider the logically equivalent definitions

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

and

```
ancestor3(X, Z) :- parent(X, Z).
ancestor3(X, Z) :- ancestor3(X, Y), parent(Y, Z).
```

The former gives a definite answer to all ground queries.

The latter does not terminate for queries where the former fails. ➥

We need to understand the *Operational Meaning* of Prolog to explain this.

Understanding Operational Meaning (how Prolog really works)

Substitutions and Unification

Definition: A *substitution* is an association of terms with variables. Here are three examples:

$$\begin{array}{llll} X = sid & Y = bill & & \\ X = sid & Y = bill & Z = sarah & \\ X = sid & Y = leah & Z = albert & \end{array}$$

We say (loosely) that the variables are $bound$ to the terms in the substitution.

To apply a substitution S to a formula (term) F (here written $F[\![S]\!]$), simply replace each variable in the term with the term (if any) with which it is associated in the substitution.

For example

$$parent(X, Y)[\![X = sid\ Z = gerald]\!] = parent(sid, Y)$$

To apply a substitution to a list of formulae, apply it to each of them

For example

$$\begin{array}{l} (parent(X, Y), \quad parent(Y, Z))[\![X = sid\ Z = gerald]\!] = \\ (parent(sid, Y), \ parent(Y, gerald)) \end{array}$$

We also define $S \circ T$: the *composition of substitutions* S and T such that for all formulae (terms) F

$$F[\![S \circ T]\!] = (F[\![T]\!])([\![S]\!])$$

Definition: Formulae (terms) F_1 and F_2 *unify* if there is a substitution S such that

$$F_1[\![S]\!] = F_2[\![S]\!]$$

S is called a *unifier* of F_1 and F_2 when this is so. Examples:

- ⇨ $parent(fred, Y)$ unifies with $parent(P, jim)$ with unifier $(P = fred \;\; Y = jim)$
- ⇨ $equal(X, X)$ unifies with $equal(succ(Y), succ(zero))$ with unifier $(X = succ(zero) \;\; Y = zero)$

Now consider the formulae

$$append(X : XS, YS, X : ZS) \;\text{ and }\; append(WS, 3 : nil, 1 : 2 : 3 : nil)$$

which are unified by the substitution, U

$$WS = 1 : XS \;\; YS = 3 : nil \;\; X = 1 \;\; ZS = 2 : 3 : nil$$

There are many other unifiers (for example, any extension of U which assigns to XS) but they are all more specific than U.

Definition: U is the *most general unifier* of formulae (terms) F_1 and F_2, iff for each unifier, V of F_1, F_2 there is a substitution S such that $V = [\![S \circ U]\!]$.

Not all pairs of terms have unifiers.

1. 3 and 5 – distinct primitives.
2. $asia$ and $africa$ – distinct constants.
3. $foo(3)$ and $baz(3)$ – distinct constructors.
4. $foo(3)$ and $foo(5)$ – distinct subterms.
5. $foo(X, X)$ and $foo(5, 6)$ – contradictory substitutions for X.
6. XS and $X : XS$ – infinite terms are not allowed.

➥

Finding Solutions

first approximation

Problems (goals) take the form $G_1, ... G_k$

procedure *solve*$((G_1,\ ...\ G_k))$
  if $k \neq 0$
    if the head of any clause unifies with G_1 **then**
      Make a fresh copy of all such clauses (see below)
      for each such copy A :- $B_1,\ ...\ B_j$
        Let σ be the most general unifier of A and G_1
        solve$((\underbrace{B_1[\![\sigma]\!],\ ...\ B_j[\![\sigma]\!]}_{\text{NB: } j \text{ may be } 0},\ G_2[\![\sigma]\!],\ ...\ G_k[\![\sigma]\!]))$
      end for
    else
      No clause applies, so this activation of *solve* has failed.
    end if
  else
    The search has succeeded
  end if
  Return to the calling *solve*, thereby exploring the next applicable clause.
end *solve*

⇨ A fresh copy of a definition is one whose variables have been systematically renamed to variables unused so far.

⇨ Fresh copies are made to keep the variables in a definition which is going to be used distinct from those in the goal it will be used on.

Finding Solutions

second approximation

The answer substitution α is accumulated as the search proceeds

procedure *solve*$((G_1,\ \ldots\ G_k),\ (\ \alpha\ ))$
  if $k \neq 0$
    if the head of any clause unifies with G_1 **then**
      Make a fresh copy of all such clauses
      for each such copy A :- $B_1,\ \ldots\ B_j$ in order of definition
        Let σ be the most general unifier of A and G_1
          solve$((B_1[\![\sigma]\!],\ \ldots\ B_j[\![\sigma]\!],\ G_2[\![\sigma]\!],\ \ldots\ G_k[\![\sigma]\!]),\ (\ \sigma \circ \alpha\ ))$
      end for
    else
      No clause applies, so this activation of *solve* has failed.
    end if
  else
    An answer has been found: it is the substitution $(\ \alpha\ )$
  end if
  Return to the calling *solve*, thereby exploring the next applicable clause.
end *solve*

⇨ "In order of definition" means "in the order in which the clause appeared when the definitions were made."

⇨ Prolog searches *depth-first* – witness the placing of the additional subgoals for the recursive *solve*.

Example

```
solve(parent(sid,X), male(X)), ())
|
|σ is X=bill
+-solve((male(bill)), (X=bill)
| |
| |σ is ()
| +-solve((), (X=bill))
|   succeeds
|
|σ is X=leah
+-solve(male(leah)), (X=leah))
  fails
```

Key

⇨ Alternatives at a given level of $solve$ are grouped vertically

⇨ Nested invocations of $solve$ are indented horizontally

Example

```
solve((grandparent(sid, G)), ())
|
|σ is G=Z'
+-solve((parent(sid, Y'), parent(Y', Z')), (G=Z'))
  |
  |σ is Y'=bill
  +-solve((parent(bill, Z')), (G=Z' Y'=bill))
  | |
  | |σ is Z'=gerald
  | +-solve((), (Z'=gerald G=gerald Y'=bill))
  | | succeeds
  | |
  | |σ is Z'=sarah
  | +-solve((), (Z'=sarah G=sarah Y'=bill))
  |   succeeds
  |
  |σ is Y'=leah
  +-solve((parent(leah, Z')), (G=Z' Y'=leah))
    |
    |σ is Z'=winifred
    +-solve((), (Z'=winifred G=winifred Y'=leah))
    | succeeds
    |
    |σ is Z'=albert
    +-solve((), (Z'=albert G=albert Y'=leah))
      succeeds
```

Equality

There is no built-in equality relation, but we can define one as follows.

```
#infix3 =
X=X :- .
```

This succeeds if its two actual parameters can be unified.

```
-- 3=3?
yes ?
-- all:my:cats=all:my:cats?
yes ?
```

It may, of course, bind variables

```
-- X:XS=all:my:cats?
XS = my : cats X = all ?
```

Data Representation: everything is a term

Prolog data are represented as *terms*.

There is no `data` declaration for composite term constructors in Prolog.

The following are all Prolog terms.

```
plus(3, 5)
minus(plus(3, 5), multiply(a, 4))
object(density(1)&mass(kg(56))&accel(cmss(200)))
rectangle(position(X;Y)&dimension(H;W))
3+5
(3+5)-(a*4)
```

Example: We define the relation *is* to implement basic arithmetic notation for use within our programs.

```
#infix1 is
#infix4 +
#infix5 *
V is V   :- num(V).
V is E+F :- Ev is E, Fv is F, sum(Ev, Fv, V).
V is E*F :- Ev is E, Fv is F, prod(Ev, Fv, V).
```

Observe

- ⇨ Use of built-in relations `sum` and `prod`.
- ⇨ Use of built-in predicate `num` to test for a Logpro number.

```
-- X is 3+4*5+6?
X = 29 ?
```

Technique: Answer Parameters

The most obvious definition of list reversal is (as in FP)

```
rev(nil, nil) :- .
rev(X:XS, YS) :- rev(XS, RS), append(RS, X:nil, YS).
```

This leads (as expected) to quadratic behaviour:

```
-- rev(1:2:3:4:nil R)?
after 15 inferences
R = 4 : 3 : 2 : 1 : nil ?
-- rev(1:2:3:4:5:nil, R)?
after 21 inferences
R = 5 : 4 : 3 : 2 : 1 : nil ?
```

The FP solution would be to use an "accumulating parameter"

```
rev2(nil, nil) :- .                    /* WRONG */
rev2(X:XS, YS) :- rev2(XS, X:YS).      /* WRONG */
```

A couple of traces show why this doesn't quite work

```
rev2(1:2:nil, nil)
rev2(2:nil, 1:nil)
rev2(nil, 2:1:nil)
```
fails

```
rev2(1:2:nil, R)
rev2(2:nil, 1:R)
rev2(nil, 2:1:R)
```
fails

Adding an answer parameter

The predicate `rev3(XS, YS, ANSWER)` holds if `ANSWER` is the result of appending the reverse of `XS` to `YS`.

```
rev3(nil,  YS, YS)      :- .
rev3(X:XS, YS, ANSWER)  :- rev3(XS, X:YS, ANSWER).
```

Which is linear

```
-- rev3(1:2:3:4:nil, nil, R)?
after 5 inferences
R = 4 : 3 : 2 : 1 : nil ?
-- rev3(1:2:3:4:5:nil, nil, R)?
after 6 inferences
R = 5 : 4 : 3 : 2 : 1 : nil ?
```

So the sensible definition for `rev` is

```
rev(XS, YS) :- rev3(XS, nil, YS).
```

A trace shows the operational behaviour

```
rev3(1 : 2 : 3 : nil, nil, R)
rev3(2 : 3 : nil, 1 : nil, R)
rev3(3 : nil, 2 : 1 : nil, R)
rev3(nil, 3 : 2 : 1 : nil, R)
succeeds
R = 3 : 2 : 1 : nil
```

Technique: Open Lists

Variables can be used as placeholders at the end of a list.

For example: $1 : 2 : 3 : 4 : \mathit{End}$ represents a list starting $1 : 2 : 3 : 4$ and "open" at the end.

Appending the list $5 : 6 : \mathit{nil}$ to it may be done by binding End to $5 : 6 : \mathit{nil}$

Represent an *open list* as a pair $L \mid \mathit{Lend}$ with Lend as the suffix of L

Unification of the End variable is a bit like assignment.

Appending such lists is a constant-time operation

```
#infix1 |
appol(L|Lend, Lend|Mend, L|Mend) :- .
```

```
-- appol(1:2:3:X|X, 4:5:Y|Y, R)?
After 2 inferences
R = 1 : 2 : 3 : 4 : 5 : Mend.1 | Mend.1
Y = Mend.1
X = 4 : 5 : Mend.1 ?
```

"Closing" an open list is easy: bind nil to the end marker.

```
ol2l(L|nil, L) :- .
```

Applications: anything where "++-elimination" would be used in FP.

Negation as Failure

The predicate `not P` is solved by attempting to solve `P`

⇨ If a solution to `P` is found, then `not P` fails.

⇨ If no solution to `P` is found, then `not P` succeeds.

Examples:

```
X /= Y :- not X=Y.
nonmember(X, XS) :- not member(X, XS).
```

A membership predicate that succeeds no more than once.

```
element(X, nil)  :- fail.
element(X, Y:YS) :- X=Y.
element(X, Y:YS) :- X/=Y, element(X, YS).
```

The method only works for *ground formulae*.

⇨ Success of `P(X)` just means that `P(`$Term$`)` holds for $some\ Term$.

⇨ So invalid to conclude that all instances of `not P(X)` are false.

Example: given

```
unwed_student(X) :- not wed(X), student(X).
wed(jim)     :- .
student(pip) :- .
```

The query `unwed_student(X)` would fail, even though `pip` is an unwed student.

The solution is to define

```
unwed_student(X) :- student(X), not wed(X).
```

thereby ensuring that X is bound before $not\ wed(X)$ is solved.

Some implementations, including Logpro, check that the negated formula is a ground formula before starting to solve it.

Others leave it as the programmer's responsibility to check

Example: the following succeeds because X is bound to 5 before the `member` predicate is solved

```
X=5, not member(X, 1:2:3:4:nil)?
```

Case Study: Route Finding
(contrived example exploring operational meaning)

Partial map of Oxford

```
walk(n, stgiles, balliol,   oucs)   :- .
walk(e, keble,   oucs,      oucl)   :- .
walk(n, parks,   oucl,      eng)    :- .
walk(s, banbury, eng,       oucs)   :- .
walk(s, parks,   oucl,      museum) :- .
```

Abstraction of the map

```
go(A, B) :- walk(Dir, Rd, A, B).
```

Accessiblity relation

```
access(From, From) :- .
access(From, To)   :- go(From, Place), access(Place, To).
```

Where can we get to from balliol?

```
-- access(balliol, Where)?
Where = balliol ?
Where = oucs ?
Where = oucl ?
Where = eng ?
Where = oucs ?
Where = oucl ?
Where = eng ?
Where = oucs ? .
```

The solutions are evidently repeating! What happened to the museum?

This can be remedied

```
-- access(balliol, Where)?
Where = balliol ?
Where = oucs ?
Where = oucl ?
Where = museum ?
Where = eng ?
Where = oucs ?
Where = oucl ? .
```

⇨ Superficial Remedy: change fact order (move the last `walk` higher)

⇨ Objection: a highly unstable "method" which doesn't scale.

Diagnosis:

⇨ Depth-first search gets us trapped in the Keble Rd. Triangle.

⇨ Remedy only worked accidentally because it suited the DFS order.

⇨ A bidirectional go would make it impossible to avoid such traps.

```
go(A, B) :- walk(Dir, Rd, A, B).
go(A, B) :- walk(Dir, Rd, B, A).
```

Depth-First Search is *Incomplete*
There may be solutions to a given problem which it can never find

Order of predicates *within* clauses is operationally important.

The following variant of `access` generates all solutions

```
access'(From, From) :- .
access'(From, To)   :- access'(Place, To), go(From, Place).
```

... but its performance is exponential in path length.

Improving the route finder

Keep record (Via) of places passed so far

```
route(From, From, Via, Route) :- rev(Via, Route).
route(From, To,   Via, Route) :-
        go(From, Next),
        not member(Next, Via),
        route(Next, To, Next:Via, Route).
```

The "front end" to this is $route/3$

```
route(From, To, Route) :- route(From, To, From:nil, Route).
```

This yields all solutions and then stops.

Exchanging the order of the last two formulae of the second clause of $route/4$ yields all solutions, and then diverges.

Cuts

Consider the relation defined by

```
nodups(XS,   RS)  :- nd(XS, nil, RS).
nd(nil,  YS, RS)  :- rev(YS, RS).
nd(X:XS, YS, RS)  :- member(X, YS),     nd(XS, YS, RS).
nd(X:XS, YS, RS)  :- not member(X, YS), nd(XS, X:YS, RS).
```

(output list is the same as input list without duplicate elements)

Clauses 2 and 3 of `nd` are mutually exclusive.

Multiple `member` tests are inefficient.

Prolog provides a (controversial) remedy for this kind of inefficiency.

We can rewrite `nd` using the Cut (written `!`) thus

```
nodups(XS,   RS)  :- nd(XS, nil, RS).
nd(nil,  YS, RS)  :- rev(YS, RS).
nd(X:XS, YS, RS)  :- member(X, YS), !, nd(XS, YS, RS).
nd(X:XS, YS, RS)  :- nd(XS, X:YS, RS).
```

Semantics: Cut succeeds and commits to all choices made since the parent goal was unified with the head of the clause in which Cut appears.

A cut

- ⇨ prunes all clauses $below$ it. A goal unified with a clause of a relation that executes a cut does not produce solutions using clauses that occur below that clause in the relation. ($c.f.$ `nd` clause 2 above)
- ⇨ prunes all alternative solutions to the goals which appear to its left in its clause. Thus (for example) a conjunctive goal followed by a cut will produce at most one solution.
- ⇨ $does\ not$ affect the goals to its right in its clause. They can produce more than one solution if they backtrack.

Examples: given

```
choose(1) :-.
choose(2) :-.
choose(3) :-.

below(X)     :- choose(X).
below(4)     :- .
cbelow(X)    :- !, choose(X).
cbelow(4)    :- .
det(X)       :- choose(X), !.
conj(X, Y)   :- choose(X), choose(Y).
dconj(X, Y)  :- choose(X), !, choose(Y).
ddconj(X, Y) :- choose(X), choose(Y), !.
```

we can expect the following behavious

`below(X)`	yields `X = 1 and X = 2 and X = 3 and X = 4`
`below(4)`	succeeds
`cbelow(X)`	yields `X = 1 and X = 2 and X = 3`
`cbelow(4)`	fails
`det(X)`	yields `X = 1`
`conj(X, Y)`	yields all 9 pairs
`dconj(X, Y)`	yields `X=1 Y=1 and X=1 Y=2 and X=1 Y=3`
`ddconj(X, Y)`	yields `X=1 Y=1`

Green Cuts

Example: Minimum is essentially deterministic

```
minimum(X, Y, X) :- X<=Y, !.
minimum(X, Y, Y) :- X>Y,  !.
```

Once $X \leq Y$ or $X > Y$ has succeeded there is no possibility of the other test succeeding.

Example: Merge is essentially deterministic

```
merge(X:XS, Y:YS, X:ZS)   :- X<Y, !, merge(XS,   Y:YS, ZS).
merge(X:XS, Y:YS, Y:ZS)   :- X>Y, !, merge(X:XS, YS,   ZS).
merge(X:XS, Y:YS, X:Y:ZS) :- X=Y, !, merge(XS,   YS,   ZS).
merge(XS,   nil, XS)      :- !.
merge(nil,  YS,   YS)     :- !.
```

A *green cut* is a cut whose placement doesn't alter the declarative meaning of a program (in the sense that exactly the same solutions are still found).

Cuts and Negation

Cut is related to negation in the sense that `not` might be defined

```
not X :- X, !, fail.
not X :- .
```

(in a Prolog system in which parameters can be executed)

If not were not already defined in Logpro we would write something like

```
not X :- call(X), !, fail.
not X :- .
```

We could also use cut to define (in Logpro)

```
if(Guard, Succ, NonSucc) :- call(Guard), !, call(Succ).
if(Guard, Succ, NonSucc) :- call(NonSucc).
```

Red Cuts

In the case of `nd` and `if`, the rule order is *essential* for the program to behave as intended. Permuting the rule order doesn't just change the *order* in which solutions appear, it may change the meaning of the program (in the sense that some solutions may no longer be found, the program may not terminate, or some wrong solutions may be found).

Such cuts are called `red cuts` and have to be used with great care.

For example, the following innocuous-looking relation is wrong

```
minimum(X, Y, X) :- X<=Y, !.
minimum(X, Y, Y) :- .
```

Case Study: a rule-based term-rewriting system

Problem: We want to be able to apply a variety of algebraic rewriting or simplification rules to (Prolog) terms.

For example, given the rewrite rules

```
(A+B)+C --> A+(B+C)).
0+X     --> X.
X+0     --> X.
```

we would transform the term $(C + (0 + B)) + (0 + 0)$ into $C + B$.

Constraints: we want to be able to deal with new kinds of terms and to add new rules easily, *i.e.* without changing the logic of the program.

The program we present here uses many features of Prolog fairly creatively, but it is far from efficient.

An important built-in predicate of *Prolog*/*LogPro* it uses is

```
functor(Term, Operator, SubTerms)
```

Examples:

`functor((a+b)+c, Op, St)` has answer `Op=(+), St=(a+b):c:nil`
`functor(Term, int, X:nil)` has answer `Term=int(X)`

and

`functor(nice(one, cyril), nice, one:cyril:nil)` succeeds

1. Infrastructure:

Selecting a numbered element of a list.

```
nth(0,       X:XS, X) :- .
nth(succ(N), X:XS, Y) :- nth(N, XS, Y).
```

Note that `nth` can also be used to search a list

```
-- nth(R, a:b:c:(d+e):f, (d+e))?
R = succ(succ(succ(0))) ?
```

Updating a numbered element of a list

```
replnth(0,       X:XS, X', X':XS) :- .
replnth(succ(N), X:XS, X', X:XS') :- replnth(N, XS, X', XS').
```

Example

```
-- replnth(succ(succ(succ(0))), a:b:c:d:f, d+e, R)?
R = a:b:c:(d+e):f ?
```

2. Following paths within terms

A path is a list of numbers.

To select the subterm designated by a path within a term

```
select(nil,  T, T)  :- .     -- end of the path
select(N:NS, T, T') :-
      struct(T),             -- it must be a structure
      functor(T, OP, Sts),   -- explode it
      nth(N, Sts, St),       -- get the Nth subterm
      select(NS, St, T').    -- and recursively select
```

This only works if the topmost term is a structure.

```
struct(T) :- not atom(T), not num(T), not str(T).
```

Examples

```
-- select(succ(0):succ(0):nil, x+(y+z), R)?
R = z ?
```

Remarkably, `select` may be used to search ➥

```
-- select(R, a+b+c+d, c)?
R = succ(0) : succ(0) : 0 : nil ?
```

... even for patterns!

```
-- select(R, a+b+c+d, X+Y)?
Y = b + c + d X = a R = nil ?
Y = c + d X = b R = succ(0) : nil ?
Y = d X = c R = succ(0) : succ(0) : nil ?
```

3. Changing subterms designated by a path within a term

```
assign(U, nil,  T, U)  :- .          -- end of the path
assign(U, N:NS, T, T') :-
      struct(T),                     -- must be a structure
      functor(T, OP, Sts),           -- explode it
      nth(N, Sts, St),               -- get the Nth subterm
      assign(U, NS, St, St'),        -- recursively assign within it
      replnth(N, Sts, St', Sts'),    -- construct new list of subterms
      functor(T', OP, Sts').         -- rebuild the term
```

4. The "rewriting engine"

Rewrites $Tree$ as $Tree'$ if there is a rule $Pat \longrightarrow Repl$ whose pattern occurs within $Tree$

```
#infix0 -->

rewrite(Tree, Tree') :-
  Pat --> Repl,                      -- choose a rewrite rule
  select(Path, Tree, Pat),           -- find where the pattern occurs
  !,                                 -- commit to rule and path
  assign(Repl, Path, Tree, Tree').   -- update the tree
```

5. Rules have logical variables embedded in them, for example

```
(A+B)+C --> A+(B+C).
0+X     --> X.
0*X     --> 0.
1*X     --> X.
X+0     --> X.
X*0     --> 0.
X*1     --> X.
```

Term matching/replacement implemented "for free" by Prolog matching/variables

- ⇨ each call of `-->` makes a fresh copy of the pattern and replacement
- ⇨ `select` matches the pattern, binding the logical variables
- ⇨ occurences of the logical variables in the replacement share these bindings

Remark: $select/3$ is redundant

```
select(Path, Tree, Tree') :- assign(Path, Tree', Tree, Tree).
```

6. Rewrite repeatedly until rewriting no longer possible

```
rew(Tree, Tree'') :-
   rewrite(Tree,   Tree'),
   show(Tree, "-->"),          -- show user what's happening
   !,                          -- commit to this rewrite
   rew(Tree', Tree'').         -- continue rewriting

rew(Tree, Tree)   :-
   not rewriteable(Tree),
   show(Tree, ".").            -- show user that we finished

rewriteable(Tree) :- rewrite(Tree, Tree').
```

6. Examples

```
-- t2(T), rew(T, T')?
((0 + b) + c) + d * 1 * ((f + g) + h) -->
(0 + b) + c + d * 1 * ((f + g) + h) -->
0 + b + c + d * 1 * ((f + g) + h) -->
0 + b + c + d * 1 * (f + g + h) -->
b + c + d * 1 * (f + g + h) -->
b + c + d * (f + g + h) .

T' = b + c + d * (f + g + h)
T  = ((0 + b) + c) + d * 1 * ((f + g) + h) ?

-- t1(T), rew(T, T')?
((a + b) + 0) + d -->
(a + b) + 0 + d -->
a + b + 0 + d -->
a + b + d .

T' = a + b + d
T = ((a + b) + 0) + d ?
```

Tutorial Exercises

A binary tree takes one of the forms

`leaf(`E`)`	where E is a term
T_1 `**` T_2	where T_1, T_2 are binary trees

A numeric (binary) tree has number constants at the leaves.

Exercise 1
Define a predicate `mintip/2`, such that

`mintip(T, N)` holds if `N` is the smallest leaf in the numeric tree `T`

Simulate, by means similar to those we demonstrated on slide 12, the computation which solves

```
mintip((leaf(3)*leaf(2))*(leaf(1)*leaf(4)), N)?
```

(*Answer on page* I)

Exercise 2
The program $repmin(T, T')$ takes a numeric tree, T, and yields an identically structured binary tree T' whose leaves all have the value of the minimum tip of T.

```
repmin(T, R) :- mt(T, V, V, R).

mt(leaf N,   N,   X,   leaf X)   :- .
mt(T1 ** T2, Min, X,   T1'**T2') :- mt(T1,    Min',  X, T1'),
                                    mt(T2,    Min'', X, T2'),
                                    min(Min', Min'', Min).


min(V, W, V) :- V<W.
min(V, W, W) :- V>=W.
```

The workhorse `mt` might could be specified as: $mt(T, Min, X, T')$ holds when T' is an identically structured tree to T with all data in the leaves replaced by the variable X, and Min is the minimum value in the tree T.

(turn over)

Simulate, by means similar to those we demonstrated on slide 12, the computation which solves

```
repmin((leaf(3)*leaf(2))*(leaf(1)*leaf(4)), N)?
```

Explain what is happening here.
(*Answer on page* I)

Exercise 3
Define a predicate $select$ such that $select(Path, Tree, Tree')$ holds if $Tree'$ is the subtree of $Tree$ designated by $Path$, where $Path$ is a list containing an l to indicate "left subtreee" and r to indicate "right subtree".

For example

```
select(l:l:r:nil,
       ((leaf a ** leaf b) ** leaf c) ** leaf d,
       leaf b)?
```

Make sure that the predicate is defined in such a way that it can be used to determine $Path$ given $Tree$ and $Tree'$.

What kind of answers would you expect to a query in which $Tree$ is a variable, but $Tree'$ and $Path$ are not, such as:

```
select(l:r:l:r:l:r:nil, T, leaf 3)?
```

(*Answer on page* I)

Exercise 4
Define a predicate $assign$ such that $assign(New, Path, Tree, Tree')$ holds if $Tree'$ is the same tree as $Tree$, except that New is to be found at $Path$.
(*Answer on page* I)

Exercise 5
Can the predicate $assign$ be used to implement the predicate $select$? If so, explain how.
(*Answer on page* I)

Note 1 (Page 1)
"Part of the unique attraction of computer science stems from the fact that most of its notions permit a multitude of simultaneous perspectives, connecting form and content, theory and practice, etc. Thus, the study of programming involves syntax, semantics, and pragmatics. Syntactically, a program can be specified by grammar formalisms on several levels. Semantically, a program can be characterized as a static entity such as a mathematical model and by dynamic computations such as derivation traces. Pragmatically, a program can be run by an interpreter, in an abstract machine, and as native code of a real computer. In order to obtain insights into new programming paradigms one can start off from either of these ends or from somewhere in the middle." (Harold Boley)

Note 2 (Page 2)
Even though most of us will never design a fully-fledged programming language the fact remains that "little" languages are everywhere. Some obvious examples that spring immediately to mind are: (a) the language design implicit in the choices made by the designer of any GUI, (b) the query languages to be found in most database systems, (c) the search language to be found in most mail user agents.

Note 3 (Page 3)
These are opposite ends of a (long) spectrum: most programming languages are neither purely imperative nor purely declarative. Many declarative languages have (either explicitly or implictly) imperative features – for example the input/output facilities of Haskell. Moreover most high-level imperative languages have declarative features – expression evaluation is the most obvious place where we specify the what rather than the how.

Note 4 (Page 9)
There are many implementations of Prolog, some of which are very fast. There is no Prolog language standard, so in this course we will use two interpreted implementations developed at Oxford.

PicoProlog	fast, superbly-documented Pascal implementation, austere
LogPro	slow, concise CAML implementation

Note 5 (Page 9)
The first clause means "appending nil to YS yields YS."

The second clause means "Appending $X : XS$ to YS yields $X : ZS$ if appending XS to YS yields ZS."

Note 6 (Page 14)
The fact that the prompt -- came in response to the question indicates that that statement could not be verified.

Note 7 (Page 17)
Later we will be able to be more precise about the circumstances when this is possible.

Note 8 (Page 20)
Here are two traces of failing queries

```
ancestor(faith, hope)?
ancestor(faith, hope) :-  parent(faith, hope). fails
ancestor(faith, hope) :-  parent(faith, Z.2), fails ancestor(Z.2, hope).


ancestor3(faith, hope)?
ancestor3(faith, hope) :-  parent(faith, hope). fails
ancestor3(faith, hope) :-  ancestor3(faith, Y.2), parent(Y.2, hope).

   ancestor3(faith, Y.2)  :-  parent(faith, Y.2). fails
   ancestor3(faith, Y.2)  :-  ancestor3(faith, Y.12), parent(Y.12, Y.2).

      ancestor3(faith, Y.12) :-  parent(faith, Y.12). fails
      ancestor3(faith, Y.12) :-  ancestor3(faith, Y.22), parent(Y.22, Y.12).


         ancestor3(faith, Y.22) :-  parent(faith, Y.22). fails
         ancestor3(faith, Y.22) :-  ancestor3(faith, Y.32), parent(Y.32, Y.22).
          etc., etc.
```

Note 9 (Page 23)
If two formulae (terms) unify then they have a most general unifier.

Later in the course we will present an algorithm which finds the most general unifier of two formulae (terms) if there is one.

Note 10 (Page 24)
If infinite terms were allowed, then the MGU of XS and $X : XS$ would be the following substitution, which maps XS to an infinite term $XS = X : X : X : X$.....

Note 11 (Page 25)
We use "solution of a formula" to mean: "a substitution which makes the formula true." elsewhere.

We also use "solve" to mean "find the solutions"

Note 12 (Page 25)
The step in which G_1 is replaced by $B_1, \ldots$ is called a *resolution* step, and G_1 is called the *resolvent*. The choice of G_1 as resolvent at each stage is specific to the design of Prolog.

Note 13 (Page 26)
It is worth considering just how intuitively graspable Prolog would be if the search were organised breadth-first, or using some other universal principle.

There is a sort of logic programming, employed largely in tactical theorem-proving systems, in which the specifics of the proof-search strategy can be decided (as it were) inference step by inference step. Isabelle and Jape are examples of such proof systems.

Note 14 (Page 30)
The uniformity of program and data representation is one of the very appealing features of Prolog. It supports a style of programming in which programs can easily be constructed and analysed by programs, as well as being executed by them. One needs to add metaprogramming primitives to the language if this is to be done properly. Although we won't be using them here, Logpro and other Prolog implementations have primitives such as

`call(Term)`	attempts to solve the given term
`functor(T, Op, Args)`	T composed with connective Op and arguments $Args$

Note 15 (Page 33)
Open lists are akin to *Difference Lists* of which there are many accounts in the literature (not all of them terribly easy to understand).

A difference list $(L \mid S)$ represents the list L with suffix S stripped from it.

In other words $x_1 : x_2 : \ldots : x_m : y_1 : \ldots y_n : nil \mid y_1 : \ldots y_n : nil$ represents $x_1 : x_2 : \ldots : x_m$.

Thus an object of the form $L \mid S$ where S is not a suffix of L cannot be interpreted as a difference list.

On the other hand, an object of the form $x_1 : x_2 : \ldots : x_m : End \mid End$ can be interpreted as a difference list – it's simply that we don't know what the suffix is yet.

Finally, the rule for appending open lists also works for difference lists.

It is tempting to define the difference-list to list relation as

```
dl2l(L|S, R) :- append(R, S, L).
```

but if $L \mid S$ is an open difference list, *i.e.* if S is an unbound variable, then this does not terminate. If we are content to destroy the openness of an open difference list, then we can modify the definition as follows:

```
dl2l(L|nil, L) :- !.
dl2l(L|S, R)   :- append(R, S, L).
```

The first clause will match either an open difference list (the unification with nil will close it), or one with suffix component nil. In either case, the corresponding list is just L. Since in these circumstances there are (of course) no others, we place the cut in order to prevent the second clause being explored.

The definition is still flawed: in order for the $append$ in the second clause to terminate, the suffix S must itself be "closed" – we leave the details of the changes required as an exercise.

Note 16 (Page 33)
The `appol` predicate is, perhaps, rather too concisely formulated for easy understanding. A logically identical formulation (which is less efficient only by a constant factor) is

```
appol(L|Lend, M|Mend, R|Rend) :- R=L, Lend=M, Rend=Mend.
```

Where = has the usual meaning (namely `X=X.`).

Note 17 (Page 34)
This method is based on the *closed world assumption*, namely that the facts needed in the execution of a program are exactly those which are provable in the program.

Note 18 (Page 34)
Note that the first clause of the definition of $element$ could have been omitted without changing the meaning of that relation.

Note 19 (Page 39)
The three parameter route relation ($route/3$ in Prolog jargon), is just a "front end" for the workhorse $route/4$. It can occasionally be helpful to overload names in this way rather than having to invent slightly distinct names for workhorse relations.

Note 20 (Page 41)
An appropriately-placed cut can improve the efficiency of a program by ensuring that search paths which cannot lead to a solution do not get explored. Such cuts also save space in Prolog interpreters by making it unneccessary to store information needed for backtracking. The space saving can be considerable for certain kinds of program.

Note 21 (Page 44)
Logpro's inbuilt relation `call` succeeds exactly when the formula which is its argument succeeds.

Note 22 (Page 45)
The given program succeeds on (2, 5, 5) (for example) and is therefore incorrect.

The "correctness" argument might have been: "if $X \leq Y$ then the minimum is X otherwise it's Y and there's no point in doing another comparison.

Note 23 (Page 46)
To be precise, $functor(Term, Op, Subtrees)$ holds when $Term$ has principal connective Op, and subterms as in the term-list $SubTerms$.

Notice that $functor$ can be used both for exploding a term into its constituents, or for "imploding" a term from its constituents.

Note 24 (Page 47)
For slightly embarrassing technical reasons connected with the *LogPro* implementation we will have to use Peano numbers to count. Since we will rarely see a number this will not matter.

Note 25 (Page 48)
It's because we want to use $select$ to search that we chose the recursive/constructive representation of numbers that we did. It turns out that any countable set with a successor relation will suffice. In fact the set can be finite, provided it has as many elements as there are expected to be subterms of the

"widest" of composite terms. In practice 10 or so will suffice, and we could have used the roman numerals i-x, or the digits 0-9, with the usual successor relation.

Exercise 1 (Page 53)

```
mintip(leaf(X), Val, Val)   :- Val < X.
mintip(leaf(X), Val, X)     :- Val >= X.
mintip(T1 ** T2, Val, Res)  :- mintip(T1, Val, Val'), mintip(T2, Val', Res).
```

Exercise 2 (Page 54)

mt is traversing the tree and building an isomorphic tree in which the variable (the V of the body of *repmin*) is replacing each leaf datum. When the recursive calls of *mt* are done, the *min* relation assigns to *Min* the minimum of the two subtrees' data. The topmost call of *mt* (by *repmin*) makes the logical variable which is distributed into the new tree the same as the logical variable to which the calculated minimum is assigned.

Exercise 3 (Page 54)

A reasonable definition of this predicate is

```
select(nil, T, T).
select(l:XS, T'**T'', T) :- select(XS, T', T).
select(r:XS, T'**T'', T) :- select(XS, T'', T).
```

I'd expect an answer in which *Tree* (in the example case T) is bound to a minimal tree containing the given path, with distinct variables at all the leaves save that indicated by the path. At that point I'd expect to find *Tree′*. In the example case, we'd get something like

```
T = T1 ** (T2 ** (T3 ** leaf "Hey" ** T4) ** T5) ** T6
```

Exercise 4 (Page 54)

```
assign(U, nil,   T,  U).
assign(U, l:XS,  T'**T'', V**T'') :- assign(U, XS, T',  V).
assign(U, r:XS,  T'**T'', T'**V)  :- assign(U, XS, T'', V).
```

Exercise 5 (Page 54)

```
select(Path, Tree, Tree') :- assign(Path, Tree', Tree, Tree).
```