THE SYNTAX OF PROLOG — FIRST APPROXIMATION

Predicates, clauses

- Example:

```
% A definition of the predicate with two clauses, the functor is: sum_tree/2
sum_tree(leaf(Val), Val).             %                     1. clause, fact
sum_tree(node(Left,Right), S) :-      %              head   \
        sum_tree(Left, S1),           % goal         \       |
        sum_tree(Right, S2),          % goal         | body | 2. clause, rule
        S is S1+S2.                   % goal         /       /
```

- Syntax:

⟨ Prolog progran ⟩	::=	⟨ predicate ⟩ . . .	
⟨ predicate ⟩	::=	⟨ clause ⟩ . . .	{with the same functor}
⟨ clause ⟩	::=	⟨ fact ⟩.␣ |	
		⟨ rule ⟩.␣	{functor of the clause = functor of the head}
⟨ fact ⟩	::=	⟨ head ⟩	
⟨ rule ⟩	::=	⟨ head ⟩ :- ⟨ body ⟩	
⟨ body ⟩	::=	⟨ goal ⟩, . . .	
⟨ goal ⟩	::=	⟨ term ⟩	
⟨ head ⟩	::=	⟨ term ⟩	

The format of Prolog programs

- The recommended form of Prolog programs:
  - Place clauses of a predicate one after the other, do not put any empty lines in between them. Separate predicates by empty lines.
  - Write the head of the clause at the beginning of the line, while each body goal with an indentation of a few spaces, preferably in a separate line.

Prolog terms

- Example — a clause head as a term:

```
%      sum_tree(node(Left,Right), S)    % compound term, functor is sum_tree/2
%      ________ ________________  _
%         |         |              |
% structure name    |              argument, variable
%                   \- argument, compound term
```

- Syntax:

⟨ term ⟩	::=	⟨ variable ⟩ |	{no functor}
		⟨ constant ⟩ |	{Functor: ⟨ constant ⟩ / 0 }
		⟨ compound term ⟩ |	{Functor: ⟨ structure name ⟩ / ⟨ arity ⟩ }
		(⟨ term ⟩)	{Because of operators, see later}
⟨ constant ⟩	::=	⟨ name constant ⟩ |	{also called ⟨ atom ⟩ }
		⟨ number constant ⟩	
⟨ number constant ⟩	::=	⟨ integer ⟩ |	
		⟨ float number ⟩	
⟨ compound term ⟩	::=	⟨ structure name ⟩ (⟨ argument ⟩, . . .)	
⟨ structure name ⟩	::=	⟨ name constant ⟩	
⟨ argument ⟩	::=	⟨ term ⟩	

Lexical elements

- Examples:

```
% variable:                  Fakt FAKT _fakt X2 _2 _
% name constant (atom):      fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% number constant:           0 -123 10.0 -12.1e8
% not a name constant:       !=, Istvan
% not a number constant:     1e8 1.e2
```

- Syntax:

⟨ variable ⟩ ::= ⟨ capital letter ⟩⟨ alphanumeric char ⟩... |
_ ⟨ alphanumeric char ⟩...

⟨ name constant ⟩ ::= '⟨ quoted character ⟩... ' |
⟨ lower case letter ⟩⟨ alphanumeric char ⟩... |
⟨ sticky char ⟩... | ! | ; | [ ] | { }

⟨ integer ⟩ ::= {signed or unsigned digit sequence }

⟨ float number ⟩ ::= { a sequence of digits with a compulsory decimal point in between, with an optional exponent}

⟨ quoted character ⟩ ::= {any non ' and non \ character} | \ ⟨ escape sequence ⟩

⟨ alphanumeric char ⟩ ::= ⟨ lower case letter ⟩ | ⟨ upper case letter ⟩ | ⟨ digit ⟩ | _

⟨ sticky char ⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Syntactic sweetener: operators

- Example:

```
% S is -S1+S2 is equivalent to the term: is(S, +(-(S1),S2))
```

- Terms with operators

  ⟨ compound term ⟩ ::=

	Grammar	Comment
	⟨ structure name ⟩ (⟨ argument ⟩, . . .)	{until now we had only this}
|	⟨ argument ⟩ ⟨ operator name ⟩ ⟨ argument ⟩	⟨ infix term ⟩
|	⟨ operator name ⟩ ⟨ argument ⟩	⟨ prefix term ⟩
|	⟨ argument ⟩ ⟨ operator name ⟩	⟨ postfix term ⟩

  ⟨ operator name ⟩ ::= ⟨ structure name ⟩ {if declared as an operator}

- Built-in predicates handling operators:
  - `op(Priority, Type, OpName)` or `op(Priority, Type, [OpName1,OpName2,...])`:
    - `Priority`: integer between 0–1200
    - `Type`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` - one of these name constants
    - `OpName`: any name constant
    - If priority is positive the operator(s) are defined, in case of 0 they are deleted.
  - `current_op(Priority, Type, OpName)`: lists the current operators.

Standard built-in operators

Standard operators

1200	xfx	`:- -->`
1200	fx	`:- ?-`
1100	xfy	`;`
1050	xfy	`->`
1000	xfy	`','`
900	fy	`\+`
700	xfx	`< = \= =.. =:= =< == \== =\= > >= is @< @=< @> @>=`
500	yfx	`+ - /\ \/`
400	yfx	`* / // rem mod << >>`
200	xfx	`**`
200	xfy	`^`
200	fy	`- \`

Other built-in operators of SICStus Prolog

1150	fx	`dynamic multifile block meta_predicate`
900	fy	`spy nospy`
550	xfy	`:`
500	yfx	`#`
500	fx	`+`

Characteristics of operators

- An operator is characterised by its type and priority.
- The type determines the operator-class (the way the operator is placed) and the associativity:

Type			Class	Interpretation
left-assoc.	right-assoc.	not-assoc.		
`yfx`	`xfy`	`xfx`	infix	`X f Y` ≡ `f(X, Y)`
	`fy`	`fx`	prefix	`f X` ≡ `f(X)`
`yf`		`xf`	postfix	`X f` ≡ `f(X)`

- If multiple operators are present the parenthesisation depends on the priority and associativity:
  - `a/b+c*d` ≡ `(a/b)+(c*d)` because the priority of `/` and `*` is 400, which is **smaller** than the priority of `+` (500) (smaller priority = **stronger** binding).
  - `a+b+c` ≡ `(a+b)+c` as the `+` operator's type is `yfx`, thus it is left-associative (letter `y` is on the left side of `yfx`) — binds to the left, parentheses are from the left to the right
  - `a^b^c` ≡ `a^(b^c)` as `^` operator's type is `xfy`, therefore it is right-associative (binds to the right, parentheses are from the right to the left)
  - `a=b=c` syntactically bad, as the `=` operator's type is `xfx`, thus it is not-associative.

Operators: placing of brackets

- Let us set off from a fully parenthesised term containing multiple operators.
- The priority of a subterm is the priority of its (outermost) operator.
- If a term with priority *ap* appears as an argument to an operator with priority *op* then the parenthesises around it can be omitted if:
  - $ap < op$, for example `a+(b*c)` $\equiv$ `a+b*c` ($ap = 400, op = 500$)
  - $ap = op$, and the term is the right argument of a right-associative operator, for example `a^(b^c)` $\equiv$ `a^b^c` ($ap = 200, op = 200$)
  - $ap = op$, the left argument of a left-associative operator, for example `(1+2)+3` $\equiv$ `1+2+3`. Exception: if the operator of the left argument is right-associative, thus the previous condition can be applied.
- An example for the exception:
  - `:- op(500, xfy, +^).`
    ```
    | ?- :- write((1 +^ 2) + 3), nl.   ⇒ (1+^2)+3
    | ?- :- write(1 +^ (2 + 3)), nl.   ⇒ 1+^2+3
    ```
  - Thus: in case of conflict the associativity of the first operator „wins”.

Operators —additional comments

- It is not allowed to have operators with the same name and in the same class at the same time.
- We can define operators in the text of a program with directives, for example:

```
:- op(500, xfx, --).                :- op(450, fx, @).

sum_tree(@V, V).                    (...)
```

- The twofold role of the „comma”
  - separates the arguments of the structure-term
  - works as an operator of priority 1000, type `xfy`, e.g. in clause bodies:
    `(p :- a,b,c) = :-(p,','(a,','(b,c)))`
  - the „naked” comma (`,`) is not allowed as a name constant, but as an operator it can be used.
  - In a structure-argument the term with a priority higher than 999 should be placed in brackets:

```
| ?- write_canonical((a,b,c)).  ⇒ ','(a,','(b,c))
| ?- write_canonical(a,b,c).    ⇒ ! procedure write_canonical/3 does not exist
```

- For the unambiguous analysis, the Prolog standard stipulates, that
  - an operator as an operand has to be placed in brackets, for example: `Comp = (>)`
  - infix and postfix operator with the same name cannot exist.
- These restrictions are not compulsory in many Prolog systems.

Use of operators

- What are operators good for?
  - convenient writing of arithmetic procedures , like `X is (Y+3) mod 4`
  - symbolic processing of expressions (like symbolic derivation)
  - for writing down clauses themselves (`:-` and `','` are both operators)
  - clauses can be handled over to meta-predicates, like `asserta( (p(X):-q(X),r(X)) )`
  - to make heads and procedure calls more readable:

    ```
    :- op(800, xfx, [grandparent, parent]).

    Gy grandparent N :-  Gy parent Sz, Sz parent N.
    ```

  - to make data structures more readable, like

    ```
    :- op(100, xfx, [.]).

    acid(sulphur, h.2-s-o.4).
    ```

- Why are operators bad?
  - It is a single global resource, it can cause problems in a larger project.

Arithmetic in Prolog

- Operators make it possible to write arithmetic expression in the usual way as we do in mathematics or in other programming languages.
- The `is` built-in predicate expects an arithmetic expression on its right side (2. argument), evaluates it, and unifies the result with the argument on the left side.
- The `=:=` built-in predicate expects an arithmetic expression on both sides, evaluates them, and fails if the values are not equal.
- Examples:

```
| ?- X = 1+2, write(X), write('  '), write_canonical(X), Y is X.
⇒             1+2                        +(1,2)    ⟹ X = 1+2, Y = 3 ? ; no
| ?- X = 4, Y is X/2, Y =:= 2.    ⟹ X = 4, Y = 2.0 ? ; no
| ?- X = 4, Y is X/2, Y = 2.      ⟹ no
```

- **Important**: the terms composed of arithmetical operators (+,-,. . .) are **compound Prolog terms**. Only the built-in arithmetic predicates evaluate these!
- The Prolog terms are basically symbolic, the arithmetic evaluation is the „exception”.

Classical symbolic expression processing: differentiation

- Let’s write a Prolog predicate which calculates the derivate of a term made up of numbers and the `x` name constant.

```
% deriv(Expr, D):  D derivate of Expr according to the x.
deriv(x, 1).
deriv(C, 0) :-                  number(C).
deriv(U+V, DU+DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-      deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).
  ⟹  D = 1*x+x*1+1 ? ; no

| ?- deriv((x+1)*(x+1), D).
  ⟹  D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no

| ?- deriv(I, 1*x+x*1+1).
  ⟹  I = x*x+x ? ; no

| ?- deriv(I, 0).
  ⟹  no
```

Example with operators: substitution value of a polynomial

- Polynomial: a Prolog term built from numbers and the ‘x’ name constant, using the ‘+’ and ‘*’ operators.
- The task: calculate the value of a polynomial for a given x value.

```
% value_of(Expr, X, E): E is the value of the polynomial Expr,
% with the substitution x=X
value_of(x, X, E) :-
        E = X.
value_of(Expr, _, E) :-
        number(Expr), E = Expr.
value_of(K1+K2, X, E) :-
        value_of(K1, X, E1),
        value_of(K2, X, E2),
        E is E1+E2.
value_of(K1*K2, X, E) :-
        value_of(K1, X, E1),
        value_of(K2, X, E2),
        E is E1*E2.

| ?- value_of((x+1)*x+x+2*(x+x+3), 2, E).
E = 22 ?
```

THE SEMANTICS AND EXECUTION OF PROLOG PROGRAMS

Declarative Semantics —the Logic Form of Clauses

- The notion of general clause as introduced in mathematical logic:
  $F_1, \ldots, F_n \text{:-} T_1, \ldots, T_m.$ $\qquad \forall \overline{X}(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$
- Definite clause or Horn clause: a clause whose head contains at most one element ($n \leq 1$).
- Classification of Horn clauses
  - If $n = 1, m > 0$, then the clause is called a **rule**, eg.
    `grandparent(U,G):-parent(U,P),parent(P,G).`
    logic form: $\forall UNSz(\texttt{grandparent}(U,G) \leftarrow \quad \texttt{parent}(U,P) \wedge \texttt{parent}(P,G))$
    equivalent form: $\forall UN \quad (\texttt{grandparent}(U,G) \leftarrow \exists Sz(\texttt{parent}(U,P) \wedge \texttt{parent}(P,G)))$
  - in case of $n = 1, m = 0$ the clause is a **fact**, eg.
    `parent('Imre', 'István').`
    its logic form is invariable.
  - In case of $n = 0, m > 0$ the clause is a **query**, eg.
    `:- grandparent('Imre', X).`
    logic form: $\forall X \neg \texttt{grandparent}(\texttt{'Imre'}, X)$, namely $\neg \exists X \texttt{grandparent}(\texttt{'Imre'}, X)$
  - If $n = 0, m = 0$, then it is an **empty clause**, denoted by: $\Box$. Logically this is an empty disjunction, which is equivalent to false.

The Role of Functions in Prolog

- The role of function symbols
  - The Prolog is based on so called equality-free logic, so we can not state that two terms (of first order logic) are equal.
  - This is the reason that the function symbols can be *only* be used as constructor-functions: $f(x_1,\ldots,x_n) = z \Leftrightarrow (z = f(y_1,\ldots,y_n) \land x_1 = y_1) \land \ldots \land (x_n = y_n))$
  - Example `leaf(X) = Z` ⇔ `Z = leaf(Y)` ∧ `X = Y`, namely `leaf(X)` is a new entity, different from any other entities.
- Example:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1), sum_tree(Right, S2), S is S1+S2.

| ?- sum_tree(node(leaf(1),leaf(2)), Sum).     ⇒     Sum = 3 ?
| ?- sum_tree(Tree, 3).                        ⇒     Tree =leaf(3) ?
```

  - The term `node(leaf(1),leaf(2))` in the query is taken apart by the procedure.
  - Pattern matching (unification) is bidirectional: it is able to both construct and disassemble.

Declarative Semantics of Prolog

- Declarative Semantics
  - An auxiliary concept: an **instance** of a term/statement: a term/statement obtained by substituting certain variables in it.
  - The execution of a query is **successful**, if an instance of the body of the query is logical **consequence** of the program (i.e. the conjunction of the clauses in the program).
  - The result of the execution is the **substitution** which produces the instance.
  - A query may execute successfully in multiple ways.
  - The execution of a query **fails** if none of its instances is a consequence of the program.
- Example:

```
parent('Imre', 'István').                              (p1)
parent('Imre', 'Gizella').                             (p2)
parent('István', 'Géza').                              (p3)
parent('István', 'Sarolt').                            (p4)
parent('Gizella', 'Civakodó Henrik').                  (p5)
parent('Gizella', 'Burgundi Gizella').                 (p6)

grandparent(Gy, N) :- parent(Gy, Sz), parent(Sz, N).  (gp)

:- grandparent('Imre', N).                              (goal)
```

  - from `(p1) + (p3) + (gp)` follows that `grandparent('Imre', 'Géza')`, so `(goal)` executes successfully using the `N = 'Géza'` substitution.
  - Another example of a successful execution: `(p1)+(p4)+(gp) N = 'Sarolt'`.

Declarative Semantics

- Why is declarative semantics a good thing?
  - The program is **decomposable**: It is possible to assign a meaning to single predicates (even to single clauses) separately.
  - The program is **verifiable**: in view of the intended meanings of predicates it can be checked if the clauses describe true statements.
  - It is very important to formulate the intended meaning of a predicate in the **head comment**. This is a declarative sentence which describes the relation between the arguments. Examples:
    - Head comments: `% parent(C,P):` `C` has parent `P`.
      `% grandparent(C,GP):` `C` has grandparent `GP`.

      ```
      grandparent(C, G) :- parent(C, P), parent(P, G).
      ```

      The meaning of the clause: If `C` has parent `P` ans `P` has parent `G`, then `C` has grandparent `G`. This is in accordance with our expectations and it is acceptable as a **true statement**.
    - Head comments: `% sum_tree(T, Sum):` Tree `T` has leaf sum `Sum`.
      `% E is Exp:` Arithm. expr. `Exp` has value `E`. (`is` is infix!)

      ```
      sum_tree(node(L,R), S) :- sum_tree(L, S1), sum_tree(R, S2), S is S1+S2.
      ```

      The meaning: If tree `L` has leaf sum `S1` and tree `R` has leaf sum `S2`, and arith. expression `S1+S2` has value `S`, then tree `node(L,R)` has leaf sum `S`. This is again a true statement.

Declarative Semantics (contd.)

- Why is declarative semantics insufficient?
  - The declarative semantics is based on general deduction.
  - There are several ways to do deduction, so this process needs search.
  - In case of an infinite search space the deduction enginge may fall into an **infinite loop**.
  - In case of finite search space the search may have very **poor efficiency**.
  - Some **built-in predicates** are able to work only under certain conditions. For instance: `S is S1+S2` signals error, if `S1` or `S2` is unknown. Because of this

    ```
    sum_tree(node(L,R), S) :- S is S1+S2, sum_tree(L, S1), sum_tree(R, S2).
    ```

    is logically correct, but leads to an error.
- Because of these, it is very important, that the Prolog programmer has to know the correct execution mechanism of Prolog, namely the **procedural semantics** of the language.
- Motto: **Think declaratively and check procedurally!**
  So: after You have written your declarative program, think it over if the procedural execution is correct (does not fall in infinite loop, is efficient, the built-in predicates are able to work, etc.)

Procedural Semantics of Prolog

- The execution mechanism of Prolog can be described in several ways:
  - The SLD resolution theorem proving method (very briefly see below)
  - Theorem proving approach based on goal-reduction (see next slides)
  - Backtrackabke procedures using pattern matching based call mechanism (details see later).
- The resolution theorem proving method used in Prolog :
  - SLD resolution: **L**inear resolution with a **S**election function for **D**efinite clauses.
  - The query **negates** the existence of the object looked for, eg. `'Imre'` has no grandparents:
    `:- grandparents('Imre', G).` $\equiv \neg\exists G$ `grandparents('Imre',`G`)`
  - We get a new query as the so called resolvent of the query and a program clause.
  - Such resolution steps are repeated until an empty clause is reached (backtracking at dead-ends).
  - By this we prove **indirectly** that the body of the query is a consequence of the program: „false” ($\Box$) is shown to follow from the negation of the body and the program.
  - This proof is constructive, ie.’ the variables of the query are instantiated — this the answer looked for eg. $G =$ `'Géza'`).
  - Further answers can be produced by other proofs.

Prolog as a Goal-Reduction Theorem Prover

- The main idea: The goal to be solved is reduced to subgoals from which it follows.
- Example program:

```
parent('Imre', 'István').                                   (p1)
parent('Imre', 'Gizella').                                  (p2)
parent('István', 'Géza').    (...)                          (p3)

grandparent(C, G) :- parent(C, P), parent(P, G).   (gp)
```

- the initial query: `:- grandparent('Imre', G).`
  (Now a query is considered as a statement to be proved.)
- We extend the query with one or more special goals to preserve the values of the variables:

```
:- grandparent('Imre', G), write(G).
```

- The query is **reduced** (see next slide) repeatedly, until only `write` goals remain:

```
[red. with (gp) clause]    :- parent('Imre', P), parent(P, G), write(GP).
[red. with (p1) clause]    :- parent('István', G), write(G).
[red. with (p3) clause]    :- write('Géza').
```

- We can read out the result of the run from the argument of `write`.

The reduction step

- The clauses used in the example and the query:

```
parent('Imre', 'István').                             (p1)
parent('István', 'Géza').                             (p3)
grandparent(C, G) :- parent(C, P), parent(P, G).      (gp)
:- grandparent('Imre', G), write(G).
```

- Reduction step: a query + a related clause $\Rightarrow$ new query.
- The reduction step is tried for **all** clauses of the predicate (one by one):
  - The **first** query goal is made the same as the clause head by variable substitution.
  - Both the clause and the query are **specialised** using this substitution. For `(gp)` this is:

    ```
    grandparent('Imre', G) :- parent('Imre', P), parent(P, G). (gp*)
    ```

  - The first goal is replaced by the body of the clause, ie. the goal is replaced by its pre-condition. The new query in the example: `parent('Imre', P), parent(P, G), write(G).`
- Next, we reduce the goal using clause `(p1)`, specializing the **query** by `P = 'István'`:
  `parent('István', G), write(G).`
  Because we reduced the query using a fact (with an empty body), its length decreases.
- Next a similar step can be made using `(p3)`, resulting in the final query: `write('Géza').`

Reduction step —further details

- Handling of variables
  - The scope of a variable is a single clause (cf. $\forall X_1 \ldots X_j(F \leftarrow T)$).
  - Before the reduction step the clause has to be copied, systematically replacing all variables by new ones (cf. recursion).
- **Unification**: Two terms/statements are made the same, by variable substitution.
  - The variables can be substituted by arbitrary terms, including other variables.
  - The unification produces the **most general** common form eg.

```
sum_tree(leaf(X), X)
sum_tree(T,       V)
```

has common form

```
sum_tree(leaf(X), X)
```

and not eg.

```
sum_tree(leaf(0), 0)
```

  - The result of the unification is the substitution, which results in the most general common form. This is unique, except for variable renaming. In the example: `T=leaf(X), V=X`.
  - Examples:

Call:	*Head:*	*Substitution:*
`grandparent('Imre', G)`	`grandparent(C, GP)`	`C ='Imre', GP = G`
`parent('Imre', Sz)`	`parent('Imre', 'István')`	`Sz='István'`
`parent('Imre', Sz)`	`parent('István', 'Géza')`	*not unifiable*
`love('István', Who)`	`love(X, X)`	`X = 'István', Who = 'István'`
`love(Who1, Who)`	`love(X, X)`	`X = Ki, Who = Who1`

Choice points, Backtracking

- We were „lucky” in the example, the sequence of the reduction steps led to a solution
- In general case we can get into a dead-end (into a non-reducible query), eg.

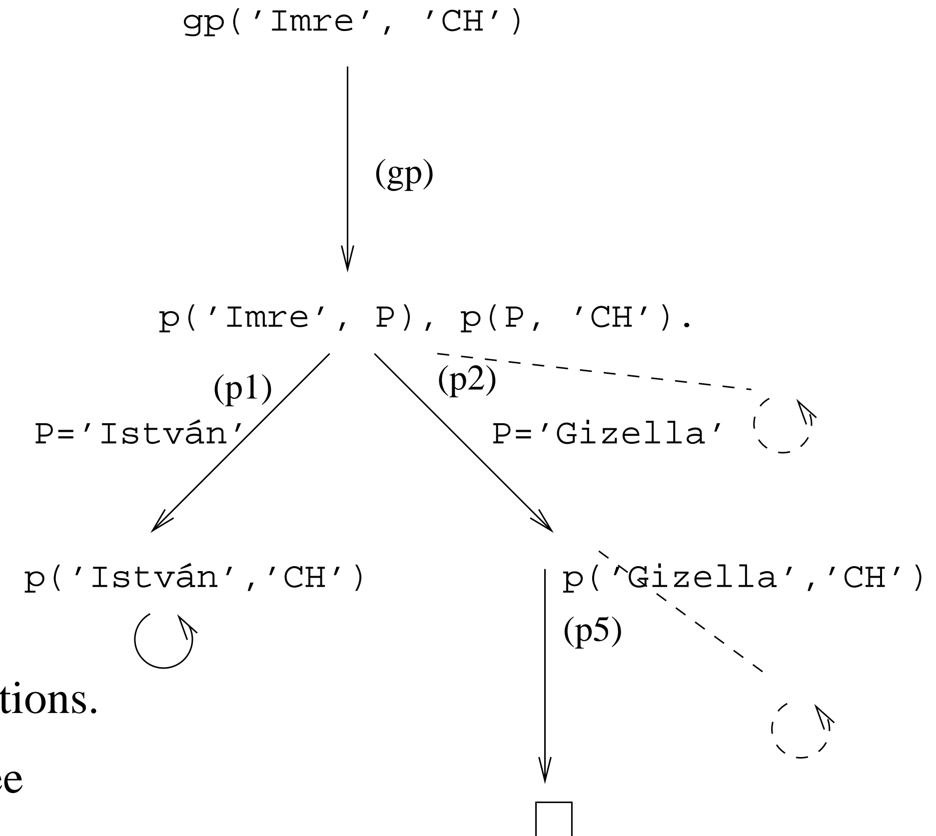
```
:- grandparent('Imre', 'Civakodó Henrik').                 (gp)
:- parent('Imre', Sz), parent(Sz, 'Civakodó Henrik').      (p1): parent('Imre', 'István')
:- parent('István', 'Civakodó Henrik').                    ???
```

- The 2nd query was reduced with clause `(p1)`, but to get to the solution we need `(p2)`: `parent('Imre', 'Gizella')` — Not only the the first clause has to be tried but all clauses!
- If a query is reduced with the non-last clause then a **choice point** is created, in which we store the query and the number of the clause used for the reduction.
- In case of a **dead-end** or **further solution** request: we go back to the choice point visited last (the youngest) and then continue the search among the **remaining clauses** (untried ones).
- If no new clause can be found at a choice point, then we backtrack further. If there are no more choice points the run of the query fails.
- In the above example: we backtrack to the step 2 and there we try the second `(p2)` clause:

```
(...) :- parent('Imre', P), parent(P, 'Civakodó Henrik').
(p1)  :- parent('Gizella', 'Civakodó Henrik').
(p5)  □
```

Visualisation of Backtracking with a Search Tree

```
p('Imre', 'István').        % (p1)
p('Imre', 'Gizella').       % (p2)
p('István', 'Géza').        % (p3)
p('István', 'Sarolt').      % (p4)
p('Gizella', 'CH').         % (p5)
p('Gizella', 'BG').         % (p6)

gp(Gy, N) :-
    p(Gy, Sz), p(Sz, N).  % (gp)
```



- The search tree
  - the nodes are execution states
  - labels appearing on
    - nodes are queries,
    - edges are clause numbers and substitutions.
- The Prolog search: traversal of the search tree
  - from left to right,
  - depth-first search.
- The dashed line denotes unsuccessful clause searches, so called *first argument indexing* eliminates the top one.

The trace of the search space

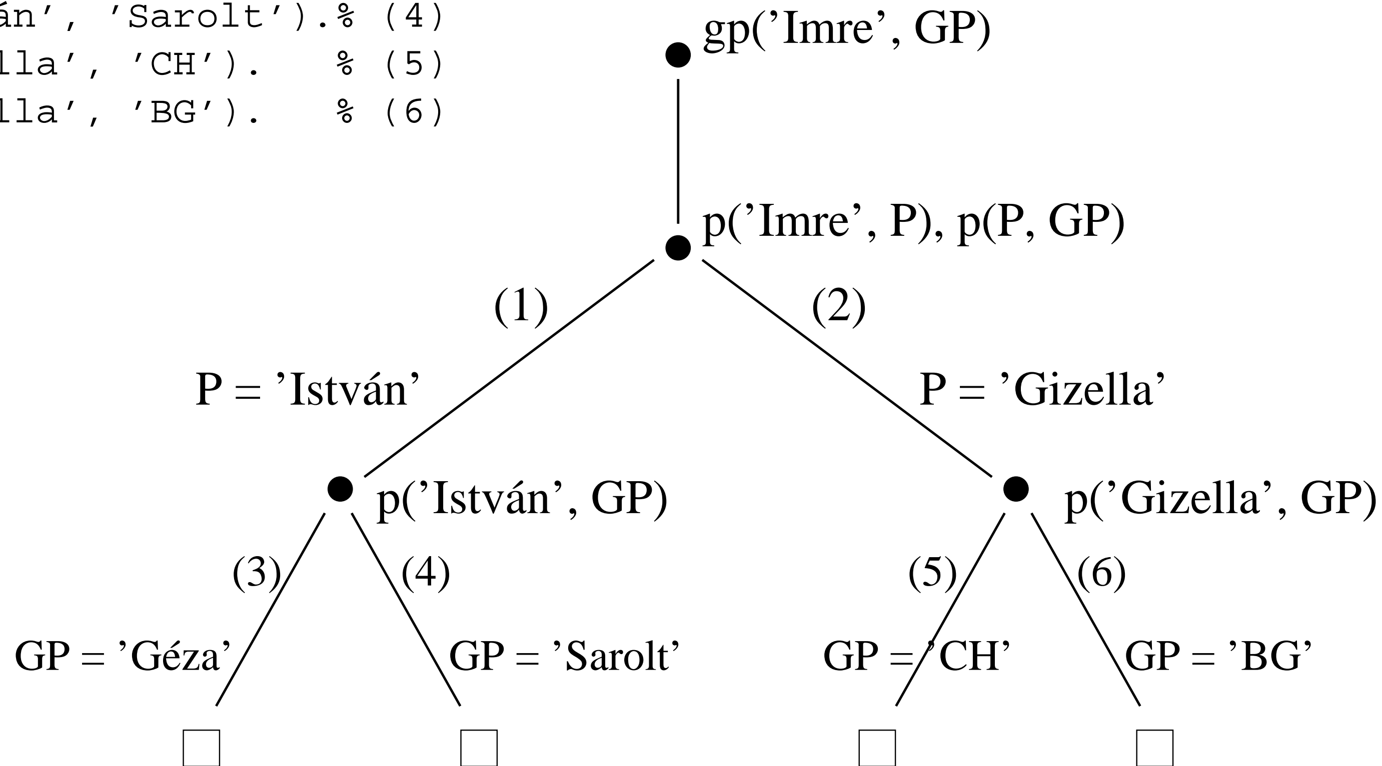
- This is an (edited) dialog with the reduction trace program. We eliminated some failing unifications.

```
|| ?- grandparent('Imre', 'Civakodó Henrik').
G0:   grandparent('Imre','Civakodó Henrik') ?                 <--- continues when RET is pressed
|                                              Trying clause 1 of grandparent/2 ... successful
| (1) {Child_1 = 'Imre', GrandParent_1 = 'Civakodó Henrik'}<--- variable renaming
|
G1:   parent('Imre',Parent_1), parent(Parent_1,'Civakodó Henrik') ?
|                                              Trying clause 1 of parent/2 ... successful
|   (1) {Parent_1 = 'István'}
|
|-----G2:   parent('István','Civakodó Henrik') ?
(...)                                                         <--- G3-G8 6 unsuccessful clause matches
|     |<<<< Failing back to goal G1                           <--- Does 'Imre' has other parents?
|                                              Trying clause 2 of parent/2 ... successful
|   (2) {Parent_1 = 'Gizella'}
|
|-----G9:   parent('Gizella','Civakodó Henrik') ?
|     |                                        Trying clause 5 of parent/2 ... successful
|     |   (5) {}
|     |-----G14:  [] ?                                        <--- empty clause, success
|     |     |++++ Solution:  ?
|     |<<<< Failing back to goal G1                   <--- see previous slide, bottom dashed line
|
|<<<< No more choices                                 <--- see previous slide, top dashed line
```

Search Tree —Another Example

```
p('Imre', 'István').   % (1)
p('Imre', 'Gizella'). % (2)
p('István', 'Géza').   % (3)
p('István', 'Sarolt').% (4)
p('Gizella', 'CH').    % (5)
p('Gizella', 'BG').    % (6)
```

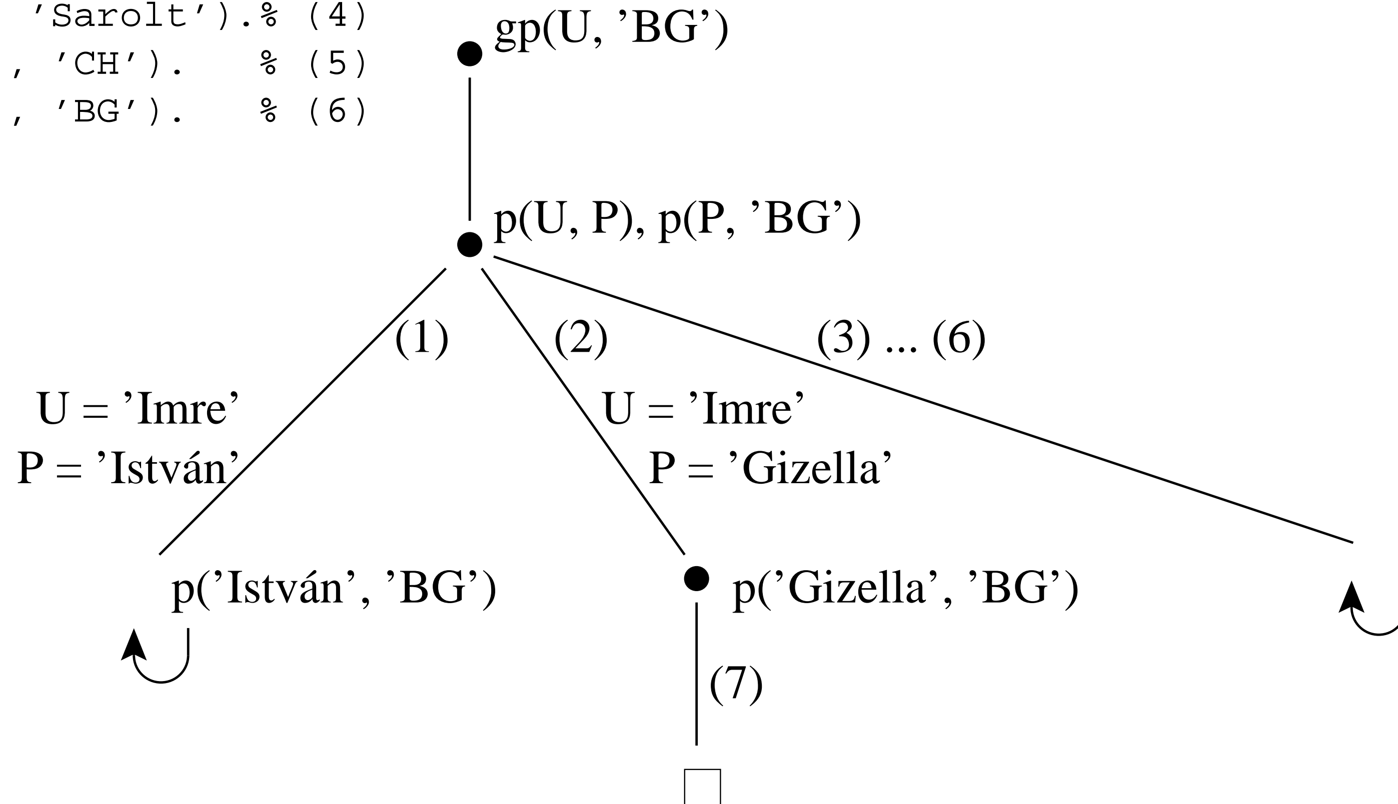
```
gp(Gy, N) :-
    gp(Gy, Sz), sz(Sz, N).
```



Search Tree —yet Another Example

```
p('Imre', 'István').  % (1)
p('Imre', 'Gizella'). % (2)
p('István', 'Géza').  % (3)
p('István', 'Sarolt').% (4)
p('Gizella', 'CH').   % (5)
p('Gizella', 'BG').   % (6)
```

```
gp(Gy, N) :-
    p(Gy, P), p(P, N).
```



PROCEDURAL MODELS OF PROLOG

Procedural models of Prolog execution

- A procedure is a set of clauses with the same functor
- A procedure is called via pattern matching (unification) of the call and a clause head.
- Models of Prolog execution:
  - Procedure-reduction model
    - Essentially this is the same as the goal-reduction model.
    - The base step: the reduction of a sequence of calls (i.e. a query) using a clause (this is the already known reduction step).
    - Backtrack: going back to an earlier query and trying it with another clause.
    - Advantages of the model: can be defined exactly, the search space is explicit.
  - Procedure-box model
    - Main idea: execution consists of passing through “ports” of nested procedure boxes.
    - Basic ports of a procedure box: entry, (successful) exit, failure.
    - Backtrack: asks for a new solution from an already exited procedure (“redo” port).
    - Advantages of the model: it is similar to the traditional recursive procedure model, the built-in Prolog tracing mechanism is based on this.

The Procedure-reduction Execution Model

- The main idea of the reduction execution model
  - A state of the execution: a query
  - The execution consists of two kinds of steps:
    - reduction step: a query + a clause $\longrightarrow$ a new query
    - backtrack (in case of dead-end): back to the last choice point
  - Choice point:
    - creation: at a reduction step which uses a non-last clause
    - activation: at backtracking, return to the query of the choice point and try **further** clauses for matching.
      (therefore the choice point has to store the serial number of the clause used matched, in addition to the query)
    - the number of the choice points can be reduced by so called *indexing*
- The reduction model can be represented with a search tree
  - During the execution the nodes of the tree are traversed using depth-first search
  - The prolog execution engine has to store the choice points on the path from the root to the current node of the search tree—this the choice point stack.

The base element of the reduction model: reduction step

- Reduction step: reduce a query to another query
  - reduction with a program clause (if the first goal calls a user-defined procedure):
    - The clause is **copied**, every variable systematically changed to a new variable.
    - The query is split into the first call and a residual query.
    - The first call is **unified** with the clause head.
    - The necessary substitutions are performed on the **body** of the clause and on the residual of the **query**
    - The new query: clause body prepended to the residual query
    - If the call and head of the clause cannot be unified then the reduction step fails.
  - reduction of a built-in call (the first goal calls a built-in procedure)
    - The query is split into the first call and a residual query.
    - The built-in procedure call is executed
    - This can be successful (and may substitute variables) or it can fail.
    - In case of success the substitutions are performed on the residual query.
    - The new query: the residual query
    - If the call of the built-in procedure fails then then the reduction step fails.

The Execution Algorithm of Prolog

1. *(Initialization:)* The stack is empty, `QU := query`
2. *(Built-in procedure:)* If the first call of `QU` is built-in then execute it,
   a. If it fails ⇒ step 6.
   b. If it is succeeds, `QU` := the result of reduction step ⇒ step 5.
3. *(Initial value for the clause counter)* `I = 1`.
4. *(Reduction step:)* Let us consider the list of clauses applicable to the first call of `QU`. If there is no indexing, then this list will contain all clauses of the predicate, with indexing this will be a filtered subsequence. Assume the list has `N` elements.
   a. If `I > N` ⇒ step 6.
   b. Reduction step between the `I`th clause of the list and `QU` query.
   c. If this fails, then `I := I+1` ⇒ step 4.
   d. If `I < N` (non-last clause), then push `<QU,I>` on the stack.
   e. `QU` := the result of reduction step
5. *(Success:)* If `QU` is empty, then execution ends with success, otherwise ⇒ step 2.
6. *(Failure:)* If the stack is empty, then execution ends with failure.
7. *(Backtrack:)* If the stack is not empty, then pop `<QU,I>` from the stack, `I := I+1`, and ⇒ step 4.

Indexing (preview)

- What is indexing?
  - Fast selection of the clauses applicable to a call (clauses potentially matching the call),
  - involving a **compile time** classification of the clauses of the procedure
- Most of the Prolog systems, including SICStus Prolog, do first argument indexing.
- The indexing is based on the outermost functor of the head argument.
  - in case of a number or a name constant `C`: the functor is `C/0`;
  - in case of structure `R` with `N` arguments: the functor is `R/N`;
  - in case of variable the functor is not defined (the clause is associated with all functors).
- Implementation of indexing:
  - At compile time: for each functor the list of the applicable clauses is built
  - At runtime: the appropriate clause list is obtained in practically constant time.
  - *Important:* if the list has a single element, no choice point is created!
- Example: the `parent('István', X)` call selects a two element clause list, but for `szuloje(X, 'István')` all 6 clauses are kept in the list (because the SICStus Prolog indexes only on the first argument)

Reduction model —advantages and disadvantages

- Advantages
  - (relatively) simple and (relatively) precise definition
  - the search space is explicit, and graphically describable
- Disadvantages
  - It hides the exit from a predicate, pl.

```
p :- q, r.              G0:   p ?
q :- s, t.              G1:   q, r ?
s.                      G2:   s, t, r ?
t.                      G3:   t, r ?
r.                      G4:   r ?       ⇐ exit from q
                        G5:   [] ?
```

  - it does not reflect the real execution mechanism of Prolog implementations
  - it cannot be used to trace a „real" Prolog program (long queries)
- That is why another model is needed :
  - Procedure box model
  - (it is also called the 4-port-box or Byrd box model)
  - the built in trace function of most Prolog systems is based on this model

Procedure-Box model

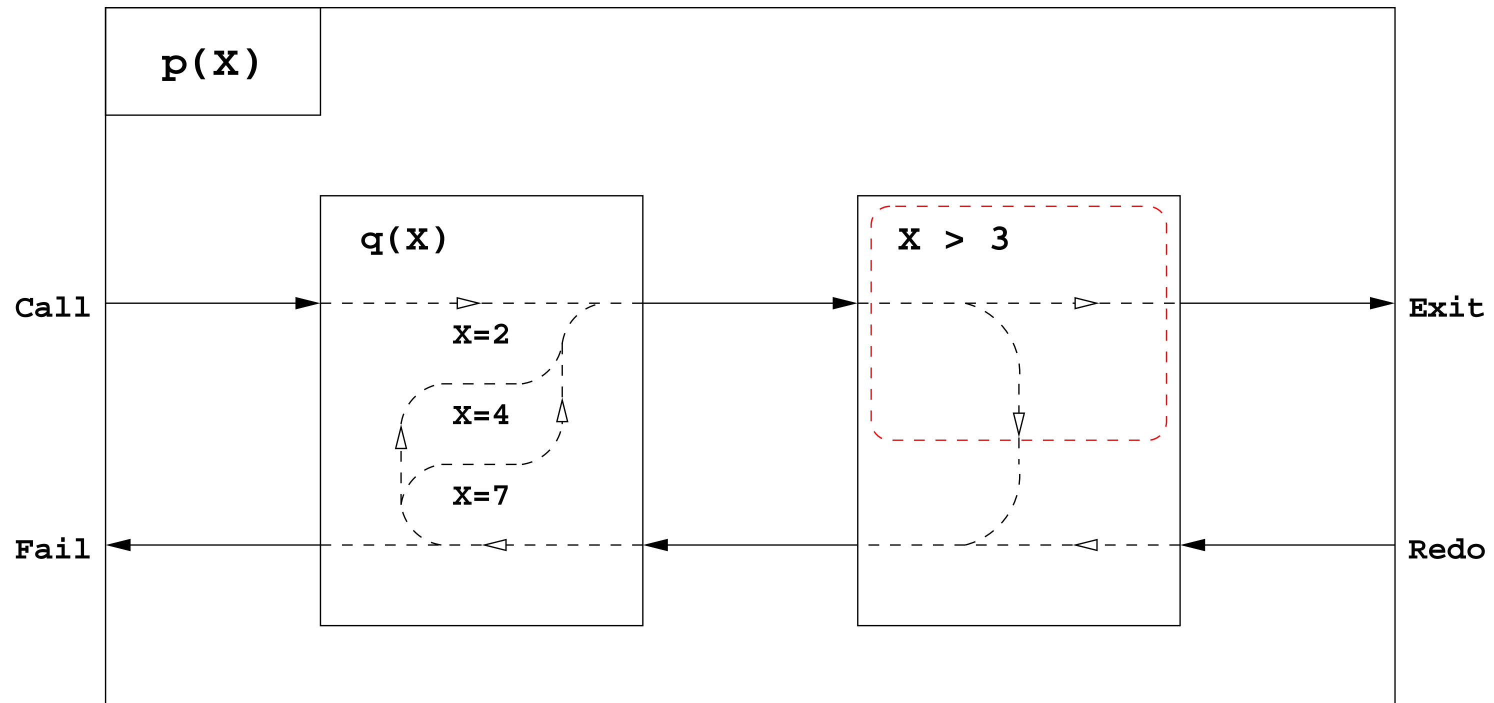
- The two phases of the Prolog procedure-execution
  - forward execution: nested procedure entries an exits
  - backward execution: requesting a new solution from an exited procedure
- A simple example:

```
q(2).  q(4).  q(7).                    p(X) :- q(X), X > 3.
```

  - Enter the `p/1` predicate (Call port)
  - Enter the `q/1` predicate (Call)
  - The `q/1` predicate exits successfully with the `q(2)` result (Exit port)
  - The `> /2` built-in predicate is entered with a `2>3` call (Call)
  - The `> /2` predicate fails (Fail port)
  - (backward execution): backtrack into the (already exited) `q/1`, asking for a new solution (Redo Port)
  - The `q/1` predicate exits with the sttq(4) result (Exit)
  - Predicate `> /2` is entered with a `4>3` call and exits successfully (Call, Exit)
  - The `p/1` predicate exits successfully with the `p(4)` result (Exit)

Procedure-Box model —graphical representation

```
q(2).  q(4). q(7).                  p(X) :- q(X), X > 3.
```



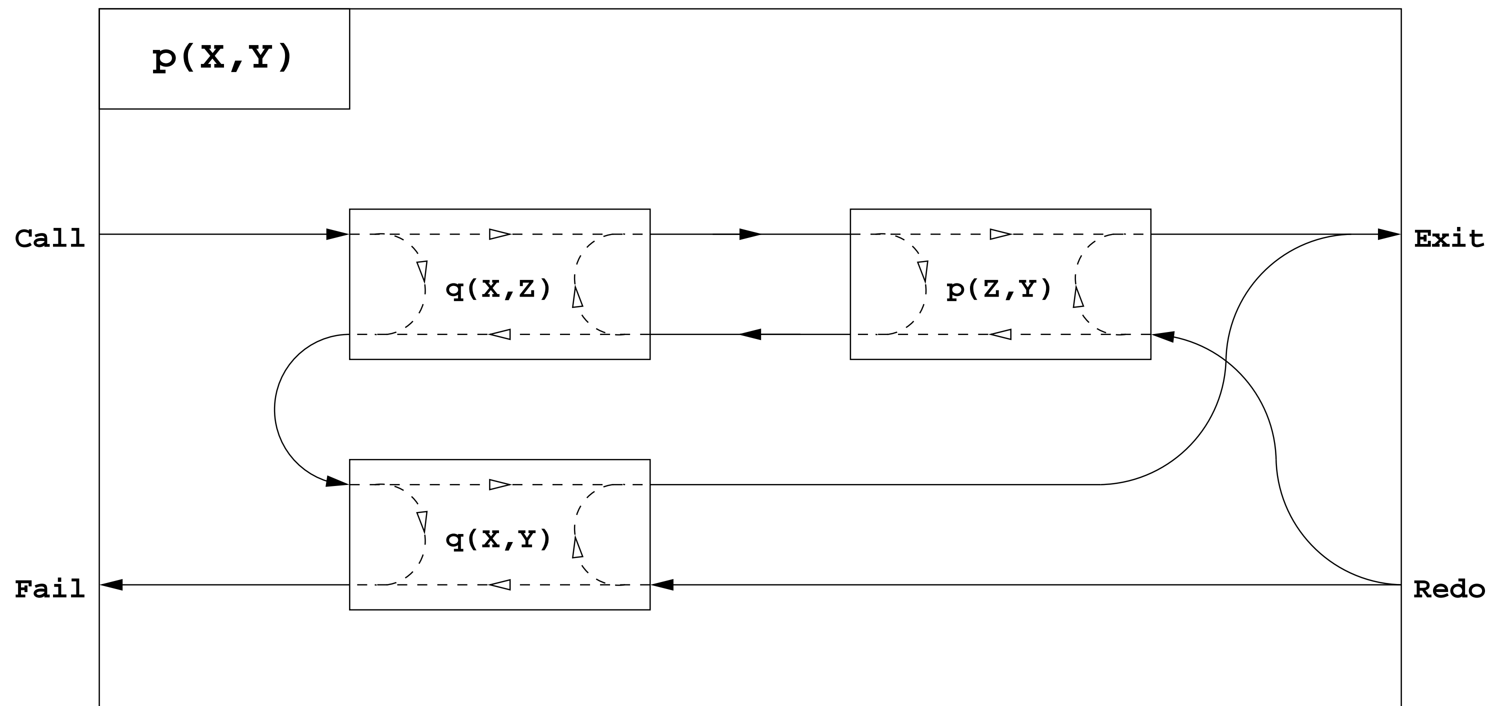
Procedure-Box model —the trace of a simple example

- The previous example tracking in SICStus Prolog

```
q(2).  q(4). q(7).

p(X) :- q(X), X > 3.


| ?- trace, p(X).
        1      1 Call: p(_463) ?
        2      2 Call: q(_463) ?
?       2      2 Exit: q(2) ?         % ? ≡ non-deterministic exit
        3      2 Call: 2>3 ?
        3      2 Fail: 2>3 ?
        2      2 Redo: q(2) ?         % backward execution
?       2      2 Exit: q(4) ?
        4      2 Call: 4>3 ?
        4      2 Exit: 4>3 ?
?       1      1 Exit: p(4) ? X = 4 ? ;
        1      1 Redo: p(4) ?         % backward execution
        2      2 Redo: q(4) ?         % backward execution
        2      2 Exit: q(7) ?
        5      2 Call: 7>3 ?
        5      2 Exit: 7>3 ?
        1      1 Exit: p(7) ?
X = 7 ? ; no
```

Procedure-Box: A more complex example

```
p(X,Y) :- q(X,Z), p(Z,Y). p(X,Y) :- q(X,Y).

q(1,2). q(2,3). q(2,4).
```



Procedure-Box model —principles of connection

- How is the box of a "parent" predicate built from boxes of predicates called in it?
- We can assume that all clause heads contain distinct variables, as the head unifications can be transformed to calls of the = /2 built-in predicate.
- Forward execution:
  - The Call port of the parent is connected to the call port of the first call of the first clause.
  - The Exit port of a predicate call is connected to
    - the Call port of the following call,
    - the Exit port of the parent if there are no following calls
- Backward Execution:
  - The Fail port of a predicate call is connected to
    - the Redo port of the preceding call, or
    - to the Call port of the first call of the following clause if there are no preceding calls
    - to the Fail port of the parent if there are no following clauses
  - The Redo port of the parent is connected to the Redo port of the last call of each clause
    - always go back to the clause from which the control exited previously

Procedure-Box Model —Object Oriented View

- Each predicate is transformed to a class which has a constructor function (to get the call parameters) and has a method to give the “(next) solution”.
- The class registers the number of the clause in which the control is.
- At the first call of the method we pass the control to the first Call port of the first clause.
- When we arrive at a Call port of a body goal an instance is **created** of the predicate called, then
- the ”next solution” method is called of this predicate instance (*)
  - If this call returns with success then the control jumps to Call port of the next call or to the Exit port of the parent
  - If this call fails then the predicate instance is destroyed and we jump to Redo port of the previous call, or to the beginning of the following clause, etc.
- When we arrive at the Redo port then we continue at step (*)
- The Redo port of the parent (which corresponds to the non first call of the “next solution” method) gives the control to the last Redo port of the clause whose clause number is stored in the instance.

Box of OO approach: p/2 C++ code of method of ”next solution”

```
boolean p::next()
{ switch(clno) {
  case 0:                     // entry point for the Call port
    clno = 1;                 // enter clause 1:                          p(X,Y) :- q(X,Z), p(Z,Y).
    qaptr = new q(x, &z);     // create a new instance of subgoal q(X,Z)
  redo11:
    if(!qaptr->next()) {      // if q(X,Z) fails
      delete qaptr;           // destroy it,
      goto cl2;               // and continue with clause 2 of p/2
    }
    pptr = new p(z, py);      // otherwise, create a new instance of subgoal p(Z,Y)
  case 1:                     // (enter here for Redo port if clno==1)
    /* redo12: */
    if(!pptr->next()) {       // if p(Z,Y) fails
      delete pptr;            // destroy it,
      goto redo11;            // and continue at redo port of q(X,Z)
    }
    return TRUE;              // otherwise, exit via the Exit port
  cl2:
    clno = 2;                 // enter clause 2:                          p(X,Y) :- q(X,Y).
    qbptr = new q(x, py);     // create a new instance of subgoal q(X,Y)
  case 2:                     // (enter here for Redo port if clno==1)
    /* redo21: */
    if(!qbptr->next()) {      // if q(X,Y) fails
      delete qbptr;           // destroy it,
      return FALSE;           // and exit via the Fail port
    }
    return TRUE;              // otherwise, exit via the Exit port
  } }
```

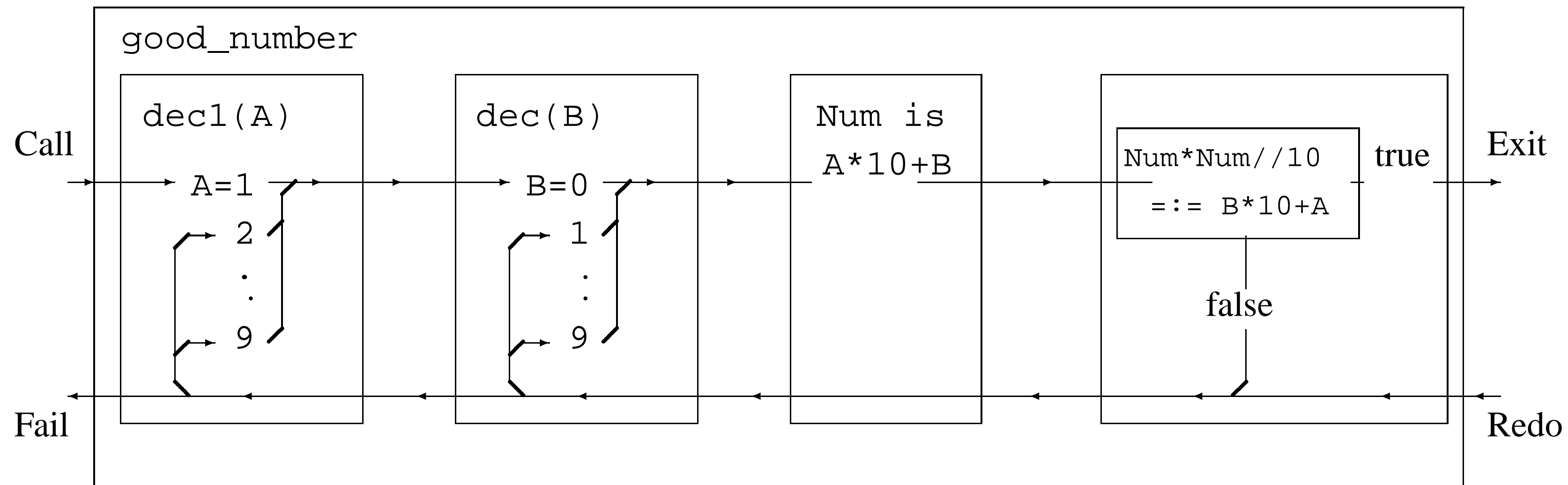
Backtracking —an arithmetic example

- Example: search of ”good” number
- The task: Find a two digit number whose square has three digits and the first two digits of the square are the same as those of the original number, but in reverse order.
- The program:

```
% dec1(J): J is a positive decimal digit.
dec1(1). dec1(2). dec1(3). dec1(4). dec1(5). dec1(6). dec1(7).
dec1(8). dec1(9).

% dec(J): J is a decimal digit.
dec(0). dec(J) :- dec1(J).

% Square of Num is a 3 digit number and begins with Num reversed.
good_number(Num):-
        dec1(A), dec(B),
        Num is A * 10 + B, Num * Num // 10 =:= B * 10 + A.
```

Prolog Execution —the 4-port procedure box model

```
good_number(Num):-
        dec1(A), dec(B),
        Num is A * 10 + B, Num * Num // 10 =:= B * 10 + A.
```



Backtracking Search —Enumeration of an Interval

- `dec(J)` enumerated the integers between 0 and 9
- Generalization: let's enumerate the integers between `N` and `M` (`N` and `M` are integers)

```
% between(M, N, I): M =< I =< N, I integer.
between(M, N, M) :-
        M =< N.
between(M, N, I) :-
        M < N,
        M1 is M+1,
        between(M1, N, I).

% dec(X): X is a decimal digit
dec(X) :- between(0, 9, X).

| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ;
Z = 14 ? ;
Z = 23 ? ;
Z = 24 ? ;
no
```

The SICStus Procedure-Box Model Debugger —the Most Important Commands

- Baisc tracing commands
  - `h <RET>` (help) — displays the list of commands
  - `c <RET>` (creep) or `<RET>` — continue tracing, stopping at every port
  - `l <RET>` (leap) — just stop at breakpoints, but keep building boxes at all ports
  - `z <RET>` (zip) — just stop at breakpoints, do not build boxes
  - `+ <RET>` resp. `- <RET>` — put/remove a spypoint on the current predicate
  - `s <RET>` (skip) — skips the body of the predicate (Call/Redo ⇒ Exit/Fail)
  - `o <RET>` (out) — exits from the body of the predicate
- The commands which influence the Prolog execution
  - `u <RET>` (unify) — unifies the current goal with a user-supplied term, instead of execution.
  - `r <RET>` (retry) — retries the execution of the current call (jumps to the Call port)
- Other commands
  - `w <RET>` (write) — writes out the call without observing the depth-limit
  - `b <RET>` (break) — enters a new, embedded Prolog interaction level
  - `n <RET>` (notrace) — switches off tracing
  - `a <RET>` (abort) — aborts the current run

FURTHER CONTROL STRUCTURES

Disjunction, example: the „ancestor" predicate

- The „ancestor" relation is the transitive closure of the „parent" relation: a parent is an ancestor (1), and an ancestor of an ancestor is also an ancestor (2), thus:

```
% ancestor0(E, Anc): Anc is ancestor of E.
ancestor0(E, P) :- parent(E, P).                               %  (1)
ancestor0(E, Anc) :- ancestor0(E, Anc0), ancestor0(Anc0, Anc).  %  (2)
```

- The definition of `ancestor0` is mathematically correct, but gives an infinite search space:

```
parent(child,father). parent(child,mother). parent(mother,grandfather).
```

```
| ?- ancestor0(child, Anc).
   Anc = father ? ; Anc = mother ? ;  {later:} ! Error: insufficient memory
```

- The cause of the infinite recursion is that the goal `:- ancestor0(father, X).` fails at clause (1), and (2) leads to a `:- ancestor0(father, Y), ancestor0(Y, X).` goal and so on ...
- Let us eliminate the left recursion:

```
ancestor1(E, P) :- parent(E, P).                               %  (3)
ancestor1(E, Anc) :- parent(E, P), ancestor1(P, Anc).          %  (4)
```

```
| ?- ancestor1(child, Anc).
Anc = father ? ; Anc = mother ? ; Anc = grandfather ? ; no
```

- This executes all `parent(X,Y)` subgoals twice: in (3) and in (4).

The disjunction

- The `ancestor1` predicate can be made more efficient by merging its clauses:

```
ancestor2(E, Anc) :- parent(E, P), itself_or_ancestor(P, Anc).

itself_or_ancestor(E, E).                            (1)
itself_or_ancestor(E, Anc) :- ancestor2(E, Anc).
```

- The `itself_or_ancestor` predicate can be eliminated with the introduction of a **disjunction**:

```
ancestor3(E, Anc) :-
        parent(E, P),
        (   Anc = P
        ;   ancestor3(P, Anc)
        ).
```

- SICStus Prolog implements the above disjunction by building an auxiliary-predicate equivalent to `itself_or_ancestor` and transforms `ancestor3` to `ancestor2`.
- (Recall:) The `X=Y` in-built predicate unifies its two arguments.
- The `X=Y` procedure could be defined using the fact: `U = U.` ≡ `=(U, U)`, cf. `(1)`.

The disjunction as a syntactical sweetener

- The disjunction can have multiple branches. The ';' operator binds less tightly than ',' therefore the disjunction is parenthesised, while its branches don't have to be bracketed. Example:

```
a(X, Y, Z) :-
        p(X, U), q(Y, V),
        (   r(U, T), s(T, Z)
        ;   t(V, Z)
        ;   t(U, Z)
        ),
        u(X, Z).
```

- The disjunction can always be eliminated with auxiliary-predicates.
  - We look for the variables which can be found both in the disjunction and outside it, as well
  - The auxiliary-predicate will contain these variables as arguments
  - Each clause of the auxiliary-predicate corresponds to a branch of the disjunction

```
auxiliary(U, V, Z) :- r(U, T), s(T, Z).
auxiliary(U, V, Z) :- t(V, Z).
auxiliary(U, V, Z) :- t(U, Z).

a(X, Y, Z) :-
        p(X, U), q(Y, V),
        auxiliary(U, V, Z),
        u(X, Z).
```

- The semantics of the disjunction can be defined with this auxiliary-predicate conversion.

Disjunctions —comments

- Are the clauses in ‘AND’ or ‘OR’ relation?
  - Tne clauses of the database are in **AND** relation, e.g.

    ```
    parent('Imre', 'István').            parent('Imre', 'Gizella').
    ```

    means: Imre has parent István **AND** Imre has parent Gizella.
  - The clauses in AND relationship lead to disjuctive answers (which are in OR relation):

    ```
    :- parent('Imre' Sz). ⇒ Sz = 'István' ? ; Sz = 'Gizella' ? ; no
    ```

    The answer to the question „Who is a parent of Imre” is: István OR Gizella.
- The above predicate with two clauses can be converted to a single clause using a disjunction:

  ```
  parent('Imre', P) :-
          (   P = 'István'          (*)
          ;   P = 'Gizella'         (*)
          ).
  ```

  Thus the conjunction has been changed to a disjunction De Morgan identities.
- In general: all predicates can be converted to have one clause only:
  - The clauses are transformed to have identical heads usingnew variables and equalities:

    ```
    parent('Imre', P) :- P = 'István'.
    parent('Imre', P) :- P = 'Gizella'.
    ```
  - The bodies are collected into a disjunction, which forms the body of the new predicate (`(*)`).

Negation

- Task: Let us find (in the database) a parent who is **not** a grandparent!
- For this we need negation:
  - Negation by failure: the `\+ Call` structure runs the `Call` and succeeds if and only if the `Call` fails.
- A solution to the above task:

```
| ?- parent(_, X), \+ grandparent(_, X).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- An equivalent solution:

```
| ?- parent(_Gy, X), \+ parent(_, _Gy).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- What happens if the two calls are switched?

```
| ?- \+ parent(_, _Gy), parent(_Gy, X).
no
```

NF —Negation by Failure

- The `\+ Call` is a built-in meta-predicate (cf. $\not\vdash$ — unprovable)
  - executes the `Call` call,
  - if `Call` completes successfully, then fails
  - else (ie. if `Call` fails) succeeds.
- During the execution of `\+ Call` at most one solution of `Call` is obtained.
- `\+ Call` never binds variables.
- Problems with the Negation by Failure:
  - „closed world assumption” (CWA) — anything that is unprovable is considered false.

```
| ?- \+ parent('Imre', X).                ----> no
| ?- \+ parent('Géza', X).                ----> true ?
```

  - $\backslash + H$ declarative semantics: $\neg\exists X(H)$, where X denotes the variables in H that are unbound *at the moment of call.*

```
| ?- \+ X = 1, X = 2.                      ----> no
| ?- X = 2, \+ X = 1.                      ----> X = 2 ?
```

Example: determining the co-effi cient in a linear expression

- Formula: a number, the ‘x’ name constant or structures built from these via the ‘+’ and ‘*’ operators.
- `% :- type term == {x} \/ number \/ {term+term} \/ {term*term}.`
- Linear formula: a number appears at least on one side of the ‘*’ operator.

```
% coeff(Term, E): The coefficient of x is E in the Term linear formula.
coeff(x, 1).
coeff(Term, E) :-
        number(Term), E = 0.
coeff(T1+T2, E) :-
        coeff(T1, E1),
        coeff(T2, E2),
        E is E1+E2.
coeff(T1*T2, E) :-
        number(T1),
        coeff(T2, E0),
        E is T1*E0.
coeff(T1*T2, E) :-
        number(T2),
        coeff(T1, E0),
        E is T2*E0.
```

```
| ?- coeff(((x+1)*3)+x+2*(x+x+3), E).
E = 8 ? ;
no
```

```
| ?- coeff(2*3+x, E).
E = 1 ? ;
E = 1 ? ; no
```

Co-effi cient problem: eliminating repeated results

- with the use of negation:

```
(...)
coeff(K1*K2, E) :-
        number(K1), coeff(K2, E0), E is K1*E0.
coeff(K1*K2, E) :-
        \+ number(K1),
        number(K2), coeff(K1, E0), E is K2*E0.
```

- with the more efficient conditional construct:

```
(...)
coeff(K1*K2, E) :-
        (   number(K1) -> coeff(K2, E0), E is K1*E0
        ;   number(K2), coeff(K1, E0), E is K2*E0
        ).
```

Conditional constructs

- Syntax (`condition`, `then`, `else` are arbitrary goals):

```
(...) :-
        (...),
        (   condition -> then
        ;   else
        ),
        (...).
```

- Declarative semantics: the above form is equivalent to the following one, if the `condition` is a simple condition (cannot be solved in multiple ways):

```
(...) :-
        (...),
        (   condition, then
        ;   \+ condition, else
        ),
        (...).
```

Conditional constructs (continued)

- Procedural semantics
  A `(condition->then;else),continuation` goal run is:
  - The `condition` call is executed.
  - If `condition` succeeds, then the `then,continuation` subgoal remains, with the substitutions resulting from the *first* solution of the `condition`. The other solutions of the `condition` subgoal are ignored.
  - If `condition` fails, then the `else,continuation` subgoal remains without any substitution.
- Multiple branching with nested conditional constructs:

```
(    cond1 -> then1
;    cond2 -> then2
;    ...
)
```

```
(    cond1 -> then1
;   (cond2 -> then2
 ;     ...
 ...))
```

- The `else` part can be omitted, the default is: `fail`.
- The `\+ cond` negation can be replaced with the `( cond -> fail ; true )` conditional construct.

Conditional constructs —examples

- Factorial:

```
% fact(+N, ?F): N! = F.
fact(N, F) :-
        (   N = 0 -> F = 1                              % N = 0,  F = 1
        ;   N > 0, N1 is N-1, fact(N1, F1), F is N*F1
        ).
```

- The above conditional has the same meaning as the disjuntion obtained by replacing `->` by comma (see comment), but it is more efficient, as it does't leave a choicepoint.
- Sign of number:

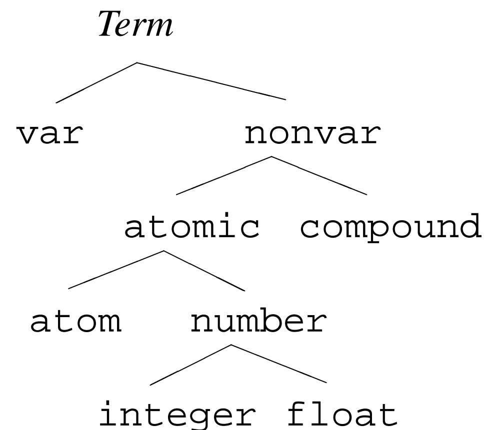
```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```

The Prolog Term: the Data Structure of Prolog

- Simple data objects:
  - Constants
    - Integers (infinity size in practice)
    - Floating point numbers
    - name constants (max 65535 characters in SICStus Prolog)
  - Variables
- compound data objects:
  - structures: $\langle$ structure name $\rangle$ ($\langle$ arg_1 $\rangle$, . . . , $\langle$ arg_n $\rangle$)
  - $\langle$ structure name $\rangle$ is an arbitrary name constant
  - $\langle$ arg_i $\rangle$ is an arbitrary term
  - The number of the arguments of a structure is also called its *arity*.
  - The arity of structures is between 1 and 255 in SICStus Prolog
  - The *functor* of the structure: $\langle$ structure name $\rangle$ / n

Prolog Terms

- Classification of Prolog terms — built-in predicates for classification



`var(X)`	X variable
`nonvar(X)`	X non-variable
`atomic(X)`	X constant
`compound(X)`	X structure
`atom(X)`	X name constant
`number(X)`	X number
`integer(X)`	X integer
`float(X)`	X float

- A classification predicate **checks** the **current** state of its argument, therefore it has no declarative semantics:

```
| ?- X = 1, integer(X).                 ⟹ yes
| ?- integer(X), X = 1.                 ⟹ no
| ?- atom('István'), atom(istvan).      ⟹ yes
| ?- compound(leaf(X)).                 ⟹ yes
| ?- compound(X).                       ⟹ no
```

Unifi cation —the Data Manipulation Mechanism of Prolog

- Unification: making two Prolog terms (eg. a procedure call and a procedure head) the same, by possibly instantiating the variables.
- Examples
  - Input parameter passing — substitutes the head variables:
    call: `grandparent('Imre', GP)`,
    head: `grandparent(C, G)`,
    substitution: `C = 'Imre', G = GP`
  - Output parameter passing — substitutes the variables of the call:
    call: `parent('Imre', P)`,
    head: `parent('Imre', 'István')`,
    substitution: `P = 'István'`
  - Input/Output parameter passing — substitutes the variables of both the call and the head:
    call: `sum_tree(leaf(5), Sum)`
    head: `sum_tree(leaf(V), V)`
    substitution: `V = 5, Sum = 5`

Unifi cation: Substitution of Variables

- The concept of substitution:
  - The substitution is a function which assigns terms to certain variables.
    - Example: $\sigma = \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{s(b,B)}, \texttt{Z}\leftarrow\texttt{C}\}$. Here $Dom(\sigma) = \{X, Y, Z\}$
    - The σ substitution assigns `a` to `X`, s(b,B) to `Y` and `C` to `Z`. Notation: $X\sigma = a$ etc.
  - The substitution function can be naturally extended to all terms:
    - $T\sigma$: σ applied to term T: the substitution σ is applied *simultaneously* to T.
    - Example: $\texttt{f(g(Z,h),A,Y)}\sigma = \texttt{f(g(C,h),A,s(b,B))}$
  - The composition of σ and θ substitutions ($\sigma \otimes \theta$) — applying them one after the other
    - The substitution of $\sigma \otimes \theta$ assigns to $x \in Dom(\sigma)$ variables the $(x\sigma)\theta$ term, and to other $y \in Dom(\theta) \backslash Dom(\sigma)$ variables it assigns $y\theta$ ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

$$\sigma \otimes \theta = \{\, x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \bigcup \{\, y \leftarrow y\theta \mid y \in Dom(\theta) \backslash Dom(\sigma)\,\}$$

    - eg. in case of $\theta = \{\texttt{X}\leftarrow\texttt{b}, \texttt{B}\leftarrow\texttt{d}\}$ $\sigma \otimes \theta = \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{s(b,d)}, \texttt{Z}\leftarrow\texttt{C}, \texttt{B}\leftarrow\texttt{d}\}$
- A term G is **more general** than S, if there exists a substitution ρ, such that $S = G\rho$
  - Example: $G =$`f(A,Y)` is more general than $S =$`f(1,s(Z))`, because in case of $\rho = \{\texttt{A}\leftarrow 1, \texttt{Y}\leftarrow\texttt{s(Z)}\}$, it holds that $S = G\rho$.

Unifi cation: the Most General Unifi er

- Terms A and B can be unified if there exists a substitution σ such that $A\sigma = B\sigma$. This $A\sigma = B\sigma$ is called a common instance of A and B.
- Two terms usually may have more common instances.
  - Example: Common instances of A =`f(X, Y)` and B =`f(s(U), U)` can be
    - C_1 =`f(s(a),a)` with the substitution of $\sigma_1 = \{$`X`←`s(a)`, `Y`←`a`, `U`←`a`$\}$
    - C_2 =`f(s(U),U)` with the substitution of $\sigma_2 = \{$`X`←`s(U)`, `Y`←`U`$\}$
    - C_3 =`f(s(Y),Y)` with the substitution of $\sigma_3 = \{$`X`←`s(Y)`, `U`←`Y`$\}$
- The most general common instance of A and B is a term C which is more general than all common instances of A and B.
  - In the example above C_2 and C_3 are the most general common instances
- **Theorem**: The most general common instance is unique, except for variable renaming.
- The most general unifier of A and B is a substitution $\sigma = mgu(A, B)$ for which $A\sigma$ and $B\sigma$ are the most general common instances of the two terms.
  - In the example above σ_2 and σ_3 are most general unifiers.
- **Theorem**: The most general unifier is unique, except for variable renaming.

The Unifi cation Algorithm

- The Unification Algorithm
  - input: two Prolog terms: A and B
  - the task: determine the unifiability of the two terms
  - the result: in case of success, return the most general unifier ($mgu(A, B)$).
- The unification algorithm, i.e. determining $\sigma = mgu(A, B)$
  1. If A and B are the same variables or constants, then $\sigma = \{\}$ (empty substitution).
  2. Else, if A is a variable, then $\sigma = \{A \leftarrow B\}$.
  3. Else, if B is a variable, then $\sigma = \{B \leftarrow A\}$.
  4. Else, if A and B are compounds with the same name and arity and their argument lists are A_1,...,A_N and B_1,...,B_N resp., and
     a. The most general unifier of A_1 and B_1 is σ_1,
     b. The most general unifier of $A_2\sigma_1$ and $B_2\sigma_1$ is σ_2,
     c. The most general unifier of $A_3\sigma_1\sigma_2$ and $B_3\sigma_1\sigma_2$ is σ_3,
     d. ...

     then $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \ldots$.
  5. In all other case A and B are not unifiable.

Unifi cation examples

- A = `sum_tree(leaf(V), V)`, B = `sum_tree(leaf(5), S)`
  - (4.) The name and arity of A and B is the same
    - (a.) `mgu(leaf(V), leaf(5))` (4th, then 2nd) = $\{\texttt{V}\leftarrow\texttt{5}\} = \sigma_1$
    - (b.) `mgu(V`σ_1`, S)` = `mgu(5, S)` (3rd) = $\{\texttt{S}\leftarrow\texttt{5}\} = \sigma_2$
  - so $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{\texttt{V}\leftarrow\texttt{5}, \texttt{S}\leftarrow\texttt{5}\}$
- A = `node(leaf(X), T)`, B = `node(T, leaf(3))`
  - (4.) The name and arity iof A and B is the same
    - (a.) `mgu(leaf(X), T)` (3rd) = $\{\texttt{T}\leftarrow\texttt{leaf(X)}\} = \sigma_1$
    - (b.) `mgu(T`σ_1`, leaf(3))` = `mgu(leaf(X), leaf(3))` (4th, then 2nd) = $\{\texttt{X}\leftarrow\texttt{3}\} = \sigma_2$
  - so $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{\texttt{T}\leftarrow\texttt{leaf(3)}, \texttt{X}\leftarrow\texttt{3}\}$

Unifi cation Examples in practice

- Built-in predicates related to unification:
  - `X = Y` unifies its two arguments, fails, if this is not possible.
  - `X \= Y` succeeds, if the two arguments are not unifiable, otherwise itfails.
- Examples:

```
| ?- 3+(4+5) = Left+Right.
        Left = 3, Right = 4+5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).
        T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.                 % because 1+2*3 ≡ 1+(2*3)
        no
| ?- X*Y = (1+2)*3.
        X = 1+2, Y = 3 ?
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
        B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
        U = f(3), X = 3, Z = 2*2 ?
```

A Further Issue in Unification: the Occurs Check

- Question: Are `X` and `s(X)` unifiable?
  - The mathematical answer is : *no*, A variable is not unifiable with a structure in which it appears (checking this is called the "occurs check").
  - The occurs check is costly, so it is not used by default, consequently cyclic terms may be created.
  - It is available as a standard predicate: `unify_with_occurs_check/2`
  - Extension (eg. SICStus): The proper handling of cyclic terms created because not performing occurs checks.
- Examples:

```
| ?- X = s(1,X).
        X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
        no
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
```