Prolog

Prolog means programmation en logique.

Language was developed by Alain Colmerauer and Philippe Roussel. (In Marseille, France.)

Three Families of Higher Programming Languages

Programming languages fall into three groups: imperative, functional, and declarative.

(These are a repetition of earlier slides.)

Declarative Programming

A program is a logical formula that specifies how the result of the computation is related to the input of the computation.

Logic programming is also called declarative, because the programmer specifies what he wants without explaining how to obtain it.

Since in general, computing from a specification alone is not possible, the form of the specification is limited to Horn Clauses, and all kinds of ugly tricks are necessary to control the search (Cuts and negation as failure).

I think that logic programming is a nice idea, but it has not kept up with the progress in imperative and functional programming. That's a pity. It is very useful for search.

Main example is Prolog.

Factorial, using the three paradigms

We want to compute $n! = 1.2. \cdots .(n-1).n$.

Imperative:

```
unsigned int fact( unsigned int n )
{
   unsigned int res = 1;
   while( n )
   {
      res = res * n;
      n = n - 1;
   }
   return res;
}
```

There are assignments, and a return statement.

Factorial (Functional)

```
define unsigned int fact( unsigned int n )
{
   if n == 0
   then 1
   else n * fact( n - 1 )
}
```

The program is the definition:

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n.(n-1)! & \text{if} \quad n > 0 \end{cases}$$

One can compute by repeatedly expanding the definition:

```
fact( 3 )
if 3 == 0 then 1 else 3 * fact( 3 - 1 )
3 * fact( 2 )
3 * ( if 2 == 0 then 1 else 2 * fact( 2 - 1 ) )
3 * ( 2 * fact( 1 ) )
3 * ( 2 * ( if 1 == 0 then 1 else 1 * fact( 1 - 1 ) ))
3 * ( 2 * ( 1 * fact( 0 ) ))
3 * ( 2 * ( 1 * ( if 0 == 0 then 1 else 1 * fact( 0 - 1 ) )))
3 * ( 2 * ( 1 * 1 ))
6
```

Factorial (Declarative)

Define a predicate fact(N, M) with meaning 'M is the factorial of N'.

```
fact(0,1).

fact( N, M ) if
   N > 0, N1 is N - 1, fact(N1,M1), M is N * M1.
```

The meaning of ',' is 'and'.

Computing works by proving. In order to find 3!, start with `fact(3,M)`, computer will find a value for M.

```
fact( 3, N )
3 > 0, N1 is 3 - 1, fact(N1,M1), M is 3 * M1.
   // Remember that ‘,’ means ‘and’
fact( 2, M1 ), M is 3 * M1.
( 2 > 0, N1a is 2 - 1, fact( N1a, M1a ), M1 is 2 * M1a ),
         M is 3 * M1.
fact( 1, M1a ), M1 is 2 * M1a, M is 3 * M1.
( 1 > 0, N1b is 1 - 1, fact( N1b, M1b ), M1a is 1 * M1b ),
         M1 is 2 * M1a, M is 3 * M1.
fact( 0, M1b ), M1a is 1 * M1b, M1 is 2 * M1a, M is 3 * M1

// Since we have fact(0,1), we get
M1a is 1 * 1, M1 is 2 * M1a, M is 3 * M1.
```

Prolog

The language is untyped.

I think that one could design a type system for Prolog, but as far as I know, nobody has done it.

In functional programming, there exist algorithms that automatically assign types to a program, (e.g. the Hindley-Milner type checking algorithm), but for Prolog, this is not possible.

For example, the `flatten` predicate is hard to type automatically.

Propositional Horn Clauses

A propositional variable is called atom.

A Horn clause is a logical formula of form

$$a_1 \wedge \cdots \wedge a_p \rightarrow b.$$

We will write $b \leftarrow a_1, \ldots, a_p$, so that the Horn clause already looks like a Prolog clause.

When $p = 0$, we will not write the $\leftarrow$.

A (propositional) logic program is a set of Horn clauses.

$$\text{abeliangroup} \leftarrow \text{abelian},\ \text{group}$$

$$\text{abelian} \leftarrow \text{commutative}$$

$$\text{group} \leftarrow \text{monoid},\ \text{hasinverse}$$

$$\text{monoid} \leftarrow \text{binary},\ \text{associative},\ \text{hasneutralelement}$$

For example, for $(+, 0)$ on the reals, we have
$\text{binary},\ \text{commutative},\ \text{associative},\ \text{hasinverse},\ \text{hasneutralelement}.$

For $+$ on `std::string`, we have
$\text{binary},\ \text{associative},\ \text{hasneutralelement}.$

$$a \leftarrow b,\ c$$

$$b \leftarrow d$$

$$d$$

$$c$$

Example of Infinite Program

$$0 < 1,$$

$$1 < 2,$$

$$2 < 3,$$

$$\ldots$$

$$N < N + 1,$$

$$\ldots$$

$$\text{all instances of} \;\; X < Z \leftarrow X < Y, \;\; Y < Z.$$

Bottom-Up Search

Let P be a logic program (a set of Horn clauses).

Start with empty set $S_0 = \emptyset$. Define

$$S_{i+1} = S_i \cup \{b \mid (b \leftarrow a_1, \ldots, a_p) \in P \text{ and } a_1, \ldots, a_p \subseteq S_i\}.$$

Let $S = \bigcup_i S_i$.

theorem: S is the set of logical consequences of P.

If P is finite, then S is finite, and for a finite i, we have $S = S_i$.

If you didn't like the formal description on the previous slide, it can be described as an algorithm too:

```
$S = \emptyset$
As long as there is a Horn clause $(b \leftarrow a_1, \ldots, a_p) \in P$,
s.t. all $a_i \in S$, but $b \notin S$, do
        $S = S \cup \{b\}$.
```

Top Down Search

Prolog uses another algorithm, namely top down search:

$\mathcal{G}$ is the goal set, the set of atoms that we need to prove.

1. If $\mathcal{G}$ is empty, then the original goal is proven.
2. Otherwise, pick an atom $b \in \mathcal{G}$.
3. Find a Horn clause $(b \leftarrow a_1, \ldots, a_p) \in P$. If more than one Horn clause exists (with b on left side), backtrack through all of them.

   If no such Horn clause exists, then give up.
4. Replace $\mathcal{G}$ by

   $$(\ \mathcal{G} \backslash \{b\}\ ) \cup \{a_1, \ldots, a_p\}.$$

   Continue at the first step.

Choices

The algorithm has two points where it has to choose:

2 Pick an atom $b \in \mathcal{G}$.

3 Select a Horn clause with left hand side b, in case there is more than one.

We are interested in the following property: If a goal g can be proven, will the algorithm prove $\mathcal{G} = \{g\}$? Does it depend on the choices that it makes?

The property is called completeness. (If a proof exists, then we will find one.)

Choice **(2)** is unimportant for completeness, choice **(3)** is important.

Choices (2)

$$a \leftarrow b,\ c$$

$$b \leftarrow d$$

$$d \leftarrow e$$

$$e \leftarrow d$$

$$b \leftarrow f$$

$$c$$

$$f$$

It does not matter if the algorithm tries to prove b or c first.

If the algorithm tries to prove b by means of $b \leftarrow d$, it will never find a proof. If it chooses $b \leftarrow f$, it will find a proof.

The problem is caused by infinite paths.

Prolog (SLD-Resolution)

- The goal set $\mathcal{G}$ is represented by a sequence $g_1, \dots, g_n$. The algorithm always picks the first goal g_1.
- It is assumed that the program is a sequence of Horn clauses. In case more than one clause matches, the clauses are tried in the order of the program.

This strategy is called SLD-Resolution, and it is the algorithm used by Prolog. The algorithm backtracks when no clause matches g_1, or the user types ; after a solution.

We will later see how to add variables.

If one puts the clause $b \leftarrow f$ first in the example, Prolog will find a proof.

Variables

We have now seen how propositional (= without variables) Prolog works.

In reality, it works with variables. Both the goal $\mathcal{G} = (g_1, \ldots, g_n)$ and the program clauses can contain variables.

Prolog tries to instantiate (substitute) variables during the search.

For the moment, I assume that atoms have the following form:

Definition: Let p be a predicate name, let $t_1, \ldots, t_m$ be a sequence of variables or constants, then $p(t_1, \ldots, t_m)$ is an atom.

Examples:

```
parent( elizabeth, charles ).   % No variables.
sibling(Y1,Y2).   % variables start with capital or
                  % underscore _
sibling( _Y1, _Y2 ).
```

Definition A substitution Θ is a finite set of assignments to variables. It is written as a set

$$\Theta = \{V_1 := t_1, \ldots, V_m := t_m\}.$$

Each V_i is a variable, and each t_i is a constant, or another variable.

It is not allowed that $V_i = V_j$ and $t_i \neq t_j$.

Later, when our atoms become more complicated, we will allow more complicated substitutions.

Unification

In order to decide if the clause `grandson(X,Y) :- grandchild(X,Y), male(X)` is useful, when proving `grandson(Z,elizabeth)`, the interpreter has to compare the terms.

By comparing, one sees that `{ X := Z, Y := elizabeth }` makes the goal equal to the head of the clause. The clause becomes `grandson(Z,elizabeth) :- grandchild(Z,elizabeth), male(Z)`.

SLD-resolution replaces the goal `grandson(Z,elizabeth)` by two goals `grandchild(Z,elizabeth), male(Z)`.

From the previous slide, we have the goal
`grandchild(Z,elizabeth), male(Z)`.

The interpreter will find the clauses

```
   grandchild(X,Y) :- grandmother(Y,X).
   grandchild(X,Y) :- grandfather(Y,X).
```

SLD-resolution tries the first, so that we compare `grandchild(Z,elizabeth)` with `grandchild(X,Y)`. The atoms fit with substitution `{ X := Z, Y := elizabeth }`. The new goal set becomes `grandmother(elizabeth,Z), male(Z).`

At this point, SLD-resolution finds the clause
`grandmother(X,Z) :- mother(X,Y), parent(Y,Z).`

Our goal from the previous slide is
`grandmother(elizabeth,Z), male(Z).`

The clause `grandmother(X,Z) :- mother(X,Y), parent(Y,Z)` fits with substitution `{ X := elizabeth }`. The result is `mother(elizabeth,Y), parent(Y,Z), male(Z),` which gets replaced by
`parent(elizabeth,Y),female(elizabeth),parent(Y,Z),male(Z).`

After that, SLD-resolution will try all of

```
parent( elizabeth, charles ).
parent( elizabeth, anne ).
parent( elizabeth, andrew ).
parent( elizabeth, edward ).
```

etc. etc.

Tracing

If you want to see the Prolog interpreter at work, type `trace.` at the command line.

Type `nodebug.` to turn it off.

Type `h` if you want to see the commands of the trace mode.

Renaming Variables

Special care needs to be taken when a variable occurs in the goal and in a clause.

For example, the goal `p(a,X)` can be proven from the program

```
    p(X,b) :- q(X).
    q(a).
```

The two variables named X in the goal and in the clause, are different variables.

In order to do this, the interpreter has to rename variables.

One can rename the goal into `p(a,X')` and substitute `{ X' := b, X := a }`.

If you use Prolog and turn on the tracer, you see that it introduces variables of form `_N`, with N a number all the time.

Renaming Variables (2)

In the example on slide 24, we ignored the problem with variable renaming. We do it correct now:

The goal from the previous slide was

```
grandmother(elizabeth,Z), male(Z).
```

The clause was

```
grandmother(X,Z) :- mother(X,Y), parent(Y,Z).
```

We should have renamed the goal into

```
grandmother(elizabeth,Z'), male(Z').
```

The substitution should have been

```
{ X := elizabeth, Z' := Z }.
```

The result would have been the same.

(This is not always the case)

Unification

The process of trying to make two atoms equal is called unification. It was invented by J.A. Robinson in 1965. The unrestricted form is one of the great achievements of logic in computer science.

I give a few examples on simple atoms: `p(X,b)` with `q(a,Y)`: Not possible.

`p(X,b)` with `p(a,Y)`. Possible with `{ X := a, Y := b }`.

`p(X,X)` with `p(Y,Z)`. Possible with `{ X := Y, Z := Y }`. There are more possibilities, e.g. `{ Y := X, Z := X }`. It doesn't matter which one is selected.

`p(X,X)` with `p(a,a)`. Possible with `{ X := a }`.

`p(X,X)` with `p(a,b)`. Not possible.

`p(X,X)` with `p(a,Y)`. Possible with `{ X := a, Y := a }`.

Prolog

We have now seen atoms of form $p(t_1, \ldots, t_m)$ where each t_i is either a constant or a variable. In reality, the t_i can arbitrary expressions, e.g.

```
p( s(a), f(X,b)),
member( s(a), [ t(a), s(X), b(X) ] ),
sin(x+y) / cos(x+y) = tan(x+y).
```

What you need to understand is that

All expressions are trees

This applies to logic, functional languages, Prolog, C^{++}, Java, etc.

Prolog (2)

definition: terms are recursively defined as follows:

1. A variable (starting with capital or '_') is a term.
2. A number (floating point or integer) is a term.
3. A constant (starting with lower case or quoted) is a term.
4. If $t_1, \ldots, t_n$ with $n > 0$ are terms, and f is a function name, then $f(t_1, \ldots, t_n)$ is a term. (f can be a function symbol, an operator `+,-,*,/`, or anything quoted.)

B.t.w. what does 'recursively defined' mean?

Recursively Defined

It usually means two things:

1. There are no terms that can be obtained in another way.
2. Every term can be obtained at most one way.

Do you know other examples of recursive definitions?

Natural Numbers

The natural numbers are recursively defined as follows:

1. 0 is a natural number.
2. If n is a natural number, then $(n+1)$ is also a natural number.

1. No natural number can be obtained in another way. Because of this we have the proof principle of complete induction. If $E(0)$ s true, and $E(n) \Rightarrow E(n+1)$ then E is true for all natural numbers.
2. No natural number can be obtained in more than one way. Because of this, we can have recursive function definitions:

$$n! = \begin{cases} n = 0 & 1 \\ n > 0 & n \times (n-1)! \end{cases}$$

(otherwise, two cases might apply at the same time.)

Atoms

Definition If $t_1, \ldots, t_n$ with $n \geq 0$ are terms, and p is a predicate name, then $p(t_1, \ldots, t_n)$ is an atom.

Lists

If you have done AI, you probably have encountered Lisp. Prolog has lists too.

Lists are built from the constant `[]` with the binary operator (.) .

```
[1] ==> .( 1, [] )
[ a, b ] ==> .( a, .( b, [] ))
[ [], [b] ] ==> .( [], .( .( b, [] ), [] ))
[ [a], [b] ] => .( .(a, [] ), .( .( b, [] ), [] ))
[ A | B ] => .( A, B )
```

Factorial

```
fact(0,1).

fact( N, M ) :-
   N > 0, N1 is N - 1, fact(N1,M1), M is N * M1.
```

The logical meaning is:

$$\begin{aligned} &\mathrm{fact}(0,1), \\ &\forall N M N_1 M_1 \;\; \mathrm{fact}(N,M) \quad \leftarrow \quad N > 0 \wedge N_1 = N - 1 \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{fact}(N_1, M_1) \wedge M = N \times M_1. \end{aligned}$$

Member Predicate

```
member( X, [ X | R ] ).
member( X, [ Y | R ] ) :- member( X, R ).
```

The logical meaning is:

$$
\begin{array}{l}
\mathrm{member}(X,\ .(X,R)\ ),\\
\forall XYR\ \ \mathrm{member}(X,\ .(Y,R)\ ) \quad \leftarrow \quad \mathrm{member}(X,R).
\end{array}
$$

```
member( 1, [ 2, 1, 3 ] ).   // Succeeds.
member( X, [ 1, 2, 3 ] ).   // Enumerates Members
member( f(X), [ f(a), g(a), f(b), g(b) ]
   // Succeeds with X = a, X = b.
```

.

Substitutions/Unification

Definition: A substitution (usually called Θ) is a finite set of assignments to variables. It is written as

$$\Theta = \{ \ V_1 := t_1, \ldots, V_m := t_m \ \}.$$

The V_i are variables, the t_i are terms (which may be variables).

It is not allowed that $V_i = V_j$, and $t_i \neq t_j$.

The application $t\Theta$ of a substitution $\Theta\{\ V_1 := t_1, \ldots, V_m := t_m\ \}$ on a term t is recursively defined as follows:

1. If t is a variable for which there is an i with $t = V_i$, then $t\Theta = t_i$.

   If t is a variable that is not equal to a t_i, then $t\Theta = t$.

2. If t is a number, then $t\Theta = t$.

3. If t is a constant, then $t\Theta = t$.

4. If t has form $f(t_1, \ldots, t_n)$ with $n > 0$, then

$$f(t_1, \ldots, t_n)\Theta = f(t_1\Theta, \ldots, t_n\Theta).$$

Unification with Terms

We are still interested in the problem of determining whether a clause can be applied on a given goal (See slide 28).

To be precise, we have a goal

$$\mathcal{G} = (g_1, \ldots, g_n),$$

and a Horn clause

$$B \leftarrow A_1, \ldots, A_m,$$

and we want to find a substitution Θ, s.t. $g_1\Theta = B\Theta$.

If such substitution exists, then SLD-resolution continues search with

$$(A_1\Theta, \ldots, A_m\Theta, g_2\Theta, \ldots, g_n\Theta).$$

It happens very often, that more than one substitution is possible.

For example, when we try to use the clause
`grandchild(X,Y) :- grandmother(Y,X)` for proving
`grandchild(Z,elizabeth)`, we might use
`{ X := Z, Y := elizabeth }`, but also
`{ Z := X, Y := elizabeth }`,
`{ X := harry, Y := elizabeth }`, or
`{ X := andrew, Y := elizabeth }`.

Only with the first two substitutions, we will find all solutions.

What do we do? SLD resolution (slide 19) already uses backtracking in the case where more than one Horn clause can be applied on a given goal.

We don't want backtracking on substitutions too.

Finding the Best Substitutions

Fortunately, one can show that among all substitutions, there always is a best substitution.

If one chooses this substitution, it is guaranteed that the choice of the substitution will not result in any backtracking.

This means that SLD-resolution needs to backtrack on choice of applicable Horn clauses, but nowhere else.

On the previous slide, the first two substitutions are optimal.

The other two are too special: There is no need to decide now that `X:= harry` or `andrew`.

The general rule is: Don't assign anything when you can avoid it.

Most General Unifiers

In order to find the matchings, we need a way of solving systems of equations.

In the simplest form, one starts with $G_1 \overset{?}{=} H$, and checks if there is a solution.

Note that one also needs to take the existing substitution Θ into account, and that during equation solving, Θ may get extended.

Later, we will see that systems of equations can be obtained in a smarter way, by doing a structural comparison first.

Solving Systems of Equations (3)

The algorithm assume that for every equation $t \stackrel{?}{=} u$, we always have $t\Theta = t$ and $u\Theta = u$. (Both sides are top level normalized.)

As long as there are equations left, select an equation $t \stackrel{?}{=} u$ and remove it from the set of equations.

- If $t = u$, then nothing further needs to be done for this equation.
- If t is a variable, then add $t \Rightarrow u$ to Θ.
- If u is a variable, then add $u \Rightarrow t$ to Θ.
- If t has form $f(t_1, \ldots, t_n)$, and u has form $f(u_1, \ldots, u_n)$, (with the same function symbol), then add new equations

  $$t_1\Theta \stackrel{?}{=} u_1\Theta, \cdots, t_n\Theta \stackrel{?}{=} u_n\Theta$$

  to the system of equations.

- In all remaining cases, there is no solution, and we **fail**.

The algorithm continues until there are no equations left, or we failed.

Extracting Equations in Smarter Way

compare(t, u, Θ) assumes that t and u are top level normalized. It returns either a set of equations that need to be made true in order to obtain $t = u$, or $\perp$, when that is that not possible.

compare(t, u, Θ) is defined as follows:

- If $t = u$, then return $\{\ \}$.
- If either of t or u is a variable, then return $\{t \overset{?}{=} u\}$.
- If t has form $f(t_1, \ldots, t_n)$, and u has form $f(u_1, \ldots, u_n)$, then let

  $$e_1 = \textbf{compare}(t_1\Theta, u_1\Theta, \Theta), \ldots, e_n = \textbf{compare}(t_n\Theta, u_n\Theta, \Theta).$$

  If one of $e_i = \perp$, then return $\perp$. Otherwise, return $e_1 \cup \cdots \cup e_n$.
- In the remaining cases, return $\perp$.

SLD-Resolution

prove$(G_1, \ldots, G_m, \Theta)$ tries to prove $G_1, \ldots, G_m$ from the program, possibly extending Θ.

- If $m = 0$, show Θ to the user, and ask for input. If the answer is **;**, then **return**, otherwise end the procedure.
- For every clause $H \leftarrow A_1 \wedge \cdots \wedge A_p$ in the program, do the following:
- Construct an α-variant $H' \leftarrow A_1' \wedge \cdots \wedge A_p'$ that shares no variables with $G_1, \ldots, G_m$.
- Call **compare**$(H'\Theta, G_1\Theta, \Theta)$. If the result is not $\perp$, then
  1. Let $t_1 \overset{?}{=} u_1, \cdots, t_n \overset{?}{=} u_n$ be the equations returned by **compare**.
  2. Try to solve $t_1 \overset{?}{=} u_1, \cdots, t_n \overset{?}{=} u_n$, possibly extending Θ.
  3. If a solution is found, then call then call

prove$(A'_1, \ldots, A'_p,\ G_2, \ldots, G_m, \Theta)$.

4. Restore Θ.

Occurs Check

Careless solving of equations may result in infinite terms, for example when $[\ X, X\ ]$ is matched against $[\ Y,\ f(Y)\ ]$.

There are several solutions:

1. Ignore the problem, and let the interpreter crash happily.
2. Reject such unifcation, by doing the occurs check.
3. Find a smart representation for infinite terms. They are called rational terms, because like rational fractions, they either end, or start repeating themselves.

   SWI Prolog uses this option.

Primitive Predicates

There are a few predicates, that are built-in. The interpreter will not look for a definition in the program.

1. `T1 = T2`, succeeds if system **compare**$(T1, T2, \Theta)$ has solution.
2. `T is T2`, succeeds if `T2` is a numerical expression that can be evaluated, and for the result `R`, **compare**(R, T, Θ) has a solution.
3. `fail`. Never succeeds.
4. `write(X)`. Always succeeds, and prints the value of 'X'.

Printing Solutions

When printing a solution Θ, the interpreter will not print the complete Θ, but only the variables that occurred in the original goal.

Cut

Prolog's backtracking feature is nice, but one does not always want backtracking:

```
      setinsert( X, L, L ) :- member( X, L ).
      setinsert( X, L, [ X | L ] ) :- \+ member( X, L ).
```

The second check is redundant, if the first has succeeded, and it requires administration, even though it is guaranteed to fail.

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

The cut is written as `(!)`, and acts like a commit. Within in the predicate where it occurs `setinsert`, all choices made until now, will become final.

Cut (2)

The predicates marked with (*) will be committed.

```
setinsert*( X, L, L ) :- member*( X, L ), ! .
setinsert*( X, L, [ X | L ] ).
```

1. Predicates before `!` in the same Horn clause are committed.
2. The choice of the Horn clause to the last goal is committed.

Actually, negation `\+` is implemented as

```
    \+ X :- X, !, fail.
    \+ X.
```

Ugly, but useful.

Unification

Definition: Let $\mathcal{E} = \{\ t_1 = u_1, \ldots, t_n = u_n\ \}$ be a set of equations between terms.

Let $\mathcal{R}$ be a rewrite system. A unifier of $\mathcal{E}$ is a substitution Θ s.t. $t_1\Theta = u_1\Theta, \ldots, t_n\Theta = u_n\Theta$.

A most general unifier is a unifier Θ, such that for every unifier Θ', there exists a substitution Σ, s.t. $\Theta' = \Theta \cdot \Sigma$.

A unifier of two terms t_1, t_2 is a unifier of the set $\{t_1 \overset{?}{=} t_2\}$. Similarly, a most general unifier of t_1, t_2 is a most general unifier of $\{t_1 \overset{?}{=} t_2\}$.

Unification (2)

The terms $f(X,1)$ and $f(0,Y)$ have unifier $\{X := 0,\ Y := 0\}$. This is also the most general unifier.

The terms $f(X, s(X))$ and $f(Y, Z)$ have unifier
$\{X := 0,\ Y := 0,\ Z := s(0)\}$. Is this a most general unifier?

Do $f(X, Y)$ and $f(a, Z)$ have a unifier?

And $f(X, Y)$ and $g(X, Y)$?

What about $f(X, Y)$ and $f(a, X)$?

And $f(X, s(X))$ and $f(s(Y), Y)$?

Most General Unifiers

G.A. Robinson showed the following (1965)

- If a system of equations has a unifier, then it has a most general unifier.
- There exists an algorithm that decides if a system of equations has a most general unifier. It also computes the most general unifier.