A Prolog Compendium

Marc Bezem
Department of Informatics, University of Bergen
P.O. Box 7800, N-5020 Bergen, Norway
bezem@ii.uib.no

November 1, 2010

1 Introduction

1.1 Algorithm = Logic + Control

Logic programming (LP) and functional programming (FP) are based on a clear separation between *what* has to be computed and *how* this should be done. LP and FP are examples of the so-called declarative style of programming. Ideally, a program in a declarative language is just a specification of *what* has to be computed. The actual implementation is left implicit and depends on the execution model of the language. These execution models are built-in in the interpreter or compiler of the language. For LP this has been formulated most clearly by Kowalski [1]:

$$\begin{aligned} \textit{algorithm} &= \textit{logic} + \textit{control} \\ &= \textit{what} + \textit{how} \end{aligned}$$

For FP the execution model is the evaluation strategy such as innermost (eager) evaluation. Another strategy in FP is outermost (lazy) evaluation. Most common is a combination of different strategies. For LP we will describe the execution model in detail in Section 3.1.

Examples of imperative languages are C, Pascal, C++ and Java. Imperative programming (IP) is based on a different paradigm. In an imperative language the focus is on *how* something is computed. The *what* is often left implicit, or in the best case expressed in the comments. The execution model of an imperative language is based on the machine state. (The state of a machine can be defined by the values in all its memory. This includes the program counter and may even include the part of the memory where the program is stored.) An imperative program is a sequence of instructions successively changing the state of the machine. Assignment is therefore the most fundamental instruction of imperative

languages. Assignment is absent in pure LP and FP, because they have other ways of dealing with values.

Let us illustrate the difference between declarative and imperative by an example. Assume we want to test whether a boolean function `p` takes value true on `0..99`. The following imperative piece of code implements this test:

```
bool test(){
  int i = 0;
// missing comment 1.
  while (i < 100 && !p(i)) i = i+1;
// missing comment 2.
  return i < 100;
// missing comment 3.
}
```

This piece of code only describes *how* 'something' is tested. *What* actually is tested is not made explicit. Of course we can blame the programmer for not writing comments. The following comments would help, one before and one after the loop:

```
//1. invariant: i <= 100 and not p(j) for all 0 <= j < i
//2. postcondition: invariant and (i >= 100 or p(i))
```

The postcondition implies that the conditional returns the desired boolean value, something which should be made explicit in another comment:

```
//3. if i < 100 then p(i), else not p(j) for all 0 <= j < i = 100
```

However, all this logic is not part of the code. In contrast, in a declarative setting one would like to write:

```
bool test = exists i in 0..99 {p(i)}; // in Haskell: any p [0..99]
```

This makes clear *what* actually should be tested and leaves the *how* to the execution model of the language.

Declarative style should provide language support for constructs like `exists`. A language with a primitive `EXISTS` is the database query languages SQL. Database query languages are good examples of declarative languages. But also certain webforms can be viewed as programs in a declarative language. People shopping on the web specify an order, and do not care about how the order actually is processed.

In IP, a program is essentially a sequence of state-changing instructions. In pure FP, a program is a set of definitions of functions. In pure LP, a program is a set of definitions of predicates. The LP execution model is answering queries about these predicates in a specific way, explained in Section 3.1. An important difference between FP and LP is that evaluating an expression in FP leads to at

most one value, whereas there can be more than one answer to a query. Some queries do have a unique answer, but other questions have no answer at all ...

In the following table we sum up the differences between IP, LP and FP.

	program	execution model	result
IP	instructions	changing machine states	final state
FP	functions	evaluating expressions	$\leq$1 value
LP	predicates	answering queries	$\geq$0 answers

Object orientation (OO) is a different aspect of programming. OO can be applied to all three of IP (e.g., Java), FP (Ocaml), LP (LOOP).

1.2 Getting Started

We propose to use SWI-Prolog `www.swi-prolog.org`, which is available for most platforms. On the Linux computers of the UiB, SWI-Prolog is launched by the command `pl`. This command launches an interactive interpreter:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.7.11)
Copyright (c) 1990-2009 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Here `?-` is the Prolog-prompt. After this prompt we can type our query. However, as we did not load a program yet, we can only ask questions about built-in predicates. Let's try some and see what happens. We enter queries after the prompt and show Prolog's response on the next line(s).

```
?- true.
true.

?-
```

Here we asked Prolog whether `true` holds, and Prolog confirms on the next line that this is indeed the case. We say that the query *succeeds*. After having answered the query, Prolog is willing to answer a new one. This is expressed by the new prompt showing up, but also by the full stop after the answer. From now on we do not show the new prompt in our examples.

Dually can we ask whether `false` holds, which Prolog denies on the next line. We say that the query *fails*:

```
?- false.
false.
```

So Prolog knows the two boolean values by their names `true` and `false` (also: `fail`). Since Prolog knows conjunction “`,`” (logical and) as well as disjunction “`;`” (logical or), we can actually query boolean expressions such as:

```
?- true,(false;true).
true.
```

and

```
?- (false;true),false.
false.
```

Some built-ins have side-effects, for example:

```
?- write(hello).
hello
true.
```

The unary predicate `write` holds for any argument and has as side-effect that it writes its argument to standard output. Here the argument `hello` is written. The fact that the query succeeds is again expressed by `true.` Writing a newline is done by `nl`. Here comes something more complicated:

```
?- write(hello),false,nl,write(great);nl,write(world).
hello
world
true.
```

How come? First, “`,`” and “`;`” are executed from left to right and “`;`” has lower precedence than “`,`”. This means that first `write(hello)` is executed to true writing `hello`. Then `false` is executed, which means that the left-hand side of the disjunction fails, skipping `nl,write(great)`. However, the right-hand of the disjunction succeeds, writing `world` on a new line. The query as a whole succeeds. It is possible that both sides of a disjunction are true. Let's see what happens then:

```
?- write(hello);write(world).
hello
true
```

Note that the Prolog prompt is not showing up here and that `true` is not followed by a full stop. This indicates that the query has succeeded, but that Prolog has not explored all possibilities for the query to succeed. If you are not interested in possible alternatives, press the return key and a full stop is written after which the Prolog prompt will appear. It may be interesting, however, to ask for possible alternatives. To see the next alternative, type “`;`” and it will show up, if it exists. Here this would lead to writing `world` and succeeding definitively:

```
?- write(hello);write(world).
hello
true ;
world
true.
```

Exercise 1.1 Try the following queries and explain what happens. For understanding the side-effects it is important to know that (sub)expressions are executed up to the point where their truth value is determined, but not any further.

```
?- false,halt.
?- write(hello);write(great);write(world).
?- (write(hello);write(great)),false,write(world).
?- (write(hello);write(great)),false;write(world).
```

1.3 Getting Help and Stopping

To open a help-window during a Prolog session, enter the query `help.` To get help on a specific topic, predicate or operator `xyz`, enter the query `help(xyz).` For example, `help(trace)` gives tracing possibilities. To stop a Prolog execution without leaving the interpreter, press (repeatedly) `Ctrl-C`:

```
?-
[forced] Action (h for help) ?
```

Hereafter one has several options, such as:

`h` for help, which shows all options;

`a` for abort, which interrupts the answering of the current query;

`e` for exit, which terminates the whole Prolog session definitively.

In UNIX systems like Linux, when you run a program from the command line, `Ctrl-C` is (almost always) the interrupt key. Other platforms have other ways of interrupting processes, typically an interrupt-button in a GUI.

Special predicates are `listing` and `halt`. The former lists your program (may be different from your intention). Executing `halt` will terminate the Prolog session immediately:

```
?- halt.
[nmimb@ii122127 pl]$
```

In the above case we are back at a (customized) Linux-prompt.

1.4 Program Example

In the previous section we have used Prolog without loading a program and our queries involved only built-ins. In Prolog, like in LP in general, a program is a definition of a set of predicates. Loading a program can be seen as extending the knowledge of the Prolog interpreter with the predicates defined in the program. For example, a program `royalty.pl`, an ordinary text-file in the current directory can be loaded in this way:[1]

```
?- consult('./royalty.pl'). % Loading a program
true.
```

The program in the text-file `royalty.pl` reads as follows:

```
man('Haakon VII').
man('Olav V').
man('Harald V').
man('Haakon').
woman('Martha').
woman('Mette-Marit').
woman('Maud').
woman('Sonja').
parent('Haakon VII','Olav V').
parent('Maud','Olav V').
parent('Olav V','Harald V').
parent('Martha','Harald V').
parent('Harald V','Haakon').
parent('Sonja','Haakon').

father(X,Y):- parent(X,Y), man(X).
mother(X,Y):- parent(X,Y), woman(X).
```

The above program defines five predicates, two unary predicates `man/1` and `woman/1` and three binary ones: `parent/2`, `father/2` and `mother/2`.[2] The unary predicates describe part of the Norwegian royal family by gender. The binary predicate `parent` describes the parent-child relationships. The predicates `man`, `woman` and `parent` are defined by listing so-called *facts*, like tuples in a database. The predicates `father`, `mother`, are defined by the last two so-called *rules* in the program. The first of these rules can be phrased as: for every `X` and `Y`, `X` is the father of `Y` if `X` is a parent of `Y` and `X` is a man. The rules can be viewed

[1] Any string is turned into a Prolog constant by putting it between apostrophes: `'...'`. Commenting out the remaining part of a line is done by `%...` In-line comments as in C: `/*...*/`

[2] The extension `/n` behind a predicate name expresses the arity of the predicate. The same name can be used with different arities for different predicates.

as defining new facts from old ones. For example, the first rule combines the facts `man('Harald V')` and `parent('Harald V','Haakon')` into the new fact `father('Harald V','Haakon')`. Note that the rule has been applied taking `'Harald V'` for `X` and `'Haakon'` for `Y`. Here `X` and `Y` are variables and `'Harald V'` and `'Haakon'` are constants. In Prolog, variables start with a capital. This is precisely the reason why the constants `'Harald V'` and `'Haakon'` are quoted: if not, they would be taken as variables.

The facts and rules are called *program clauses*, or *clauses* for short. Queries are also called *goal clauses* or *goals* for short. Answering a query is also called *solving a goal*. Let's query the program above:

```
?- woman(W).
W = 'Maud'
```

In program clauses, variables are universally quantified ("for every"). In goal clauses, variables are existentially quantified ("there exists").[3] The above query asks: "is there a `W` such that `woman(W)` holds?" Prolog answers this question on the basis of the program. The answer `W = 'Maud'` means "yes, `woman(W)` holds for `W = 'Maud'`". So the answer is more informative than just `true`, we also get information about the variable. A more precise way of paraphrasing the query above is therefore: "for which `W` do we have `woman(W)`?". Note that we do not get a full stop after the answer, nor a Prolog prompt, which means that there may be alternative answers. These show up by typing ";", which results in all four expected answers:

```
?- woman(W).
W = 'Maud' ;
W = 'Martha' ;
W = 'Sonja' ;
W = 'Mette-Marit'.
```

The last answer ends by a full stop and the Prolog prompt appears. One could wish these answers to be written without any interaction. This can be done in various ways in Prolog. One way is by a so-called *failure driven loop*:

```
?- woman(W),write(W),nl,fail.
Maud
Martha
Sonja
Mette-Marit
false.
```

In order to understand what happens we must use what we learned in the previous section. The whole query is a conjunction, which is executed from left to

[3] Universal and existential quantification have been explained in MNF130.

right. First, `woman(W)` is executed, binding `W` to `'Maud'`, while three alternative answers have to wait. Since `woman(W),write(W)` share the same variable `W`, the next term to be executed is `write('Maud')`, which succeeds writing `Maud` (without the quotes). The next term is `nl`, which succeeds with the side-effect of a new line. Finally, the fourth term `fails`. The query as a whole hasn't failed yet, as there are still alternative answers to `woman(W)`. These altenatives are processed in the same way as the first alternative, leading to the output as shown above. Thereafter the query finally fails, as signalled by `false.` Of course Prolog has other ways of generating all answers to a query, for example, by collecting them in a list.

A familiar mistake is illusterated by the following example:

```
?- parent(W,Sonja),write(W),nl,fail.
Haakon VII
Maud
Olav V
Martha
Harald V
Sonja
false.
```

If the intention was to ask for the parents of Sonja, the query is wrong. The query actually writes all persons who occur in the first argument of the relation `parent/2`. The reason is that Prolog takes unquoted words beginning with a capital letter as variable names, and we forgot to quote Sonja. Even quotes cannot prevent the next surprise:

```
?- parent(W,'Sonja').  % Use quotes!
false.
```

Why should we have expected this negative answer? Of course everyone has parents, but not everyone has royal parents. The answer is given with respect to the program and the program did not specify any parent for `'Sonja'`.

More complicated queries can easily be conceived. The following query asks for Haakon's royal grandmother:

```
?- parent(X,'Haakon'),mother(Y,X).
X = 'Harald V',
Y = 'Martha'
```

This answer is correct, but the Prolog prompt is missing indicating there may be alternatives. Could this be the other grandmother? No, typing “;” results in `false.` How come? There are two answers to `parent(X,'Haakon')`: `X = 'Harald V'` and `X = 'Sonja'`. The first answer is used above, with `Y = 'Martha'` for the first grandmother. At that moment Prolog is aware of a possible

alternative for X and offers the user to further explore this alternative. However, Prolog is unaware of the fact that the alternative X = 'Sonja' will lead to a subquery mother(Y,'Sonja') which fails. Prolog only discovers this after being asked to explore the second alternative, which results in false.

Exercise 1.2 Extend the program royalty.pl with definitions for predicates son/2 and daughter/2. Test your program with a query asking for the grandson of Haakon VII.

2 Prolog Syntax

In UNIX systems, when you run a program from the command line you can interrupt it by pressing a key. You can find out which keys have special functions like that by entering stty -a at the shell prompt. The one you want here is called "intr" and it is almost always Ctrl-C.

2.1 Basic Syntax

Here is a basic syntax of a subset of Prolog (with examples after %):

```
<const> ::= <number> | <name>                   % -3  12  c  'Maud'
<term>  ::= <const> | <var> | <name>(<terms>) % Maud  cons(Y,nil)
<terms> ::= <term> | <terms>,<term>
<atom>  ::= <name> | <name>(<terms>).     %  true   son('Maud',X)
<atoms> ::= <atom> | <atoms>,<atom>
<fact>  ::= <atom>.                                  %  help(help).
<goal>  ::= <atoms>.
<rule>  ::= <atom>:- <atoms>.                      % p(X):- q,r(X,Y).
```

The lexical categories <number>, <name> and <var> are assumed to be known. An atom in the above sense is an atomic formula in first-order logic. Terminology may vary a bit, the category <name> is often defined as 'atom in the syntactic sense'. Note that a fact is just an atom followed by a full stop and that the body of a rule is the same as a goal, that is, a sequence of one or more atoms followed by a full stop. We'll try not to be pedantic in our notation and terminology.

Exercise 2.1 Extend the syntax of Prolog above with a left-associative operator ";" for <atoms>, with lower precedence than "," (include parentheses).

Exercise 2.2 Parse some rules and goals from Section 1.2–1.4 with the extended syntax from the previous exercise.

Exercise 2.3 Extend the extended syntax of Prolog from the previous exercise with a unary operator "\+" with highest priority.

2.2 Unification

Unification means making terms identical by substitution. In LP, unification is used to determine how a program clause could be used to find a solution to a goal clause. Unification is not always possible. For example, the terms `X` and `f(X)` cannot be unified: any substitution for `X` in `f(X)` will result in a longer term than `X` itself. A *unification algorithm* decides whether unification is possible and, if so, returns a unifying substitution (a *unifier*). We start by defining what a substitution is. We use the usual notation $x, y, z, x_i, \ldots$ for variables, a, b, c for constants, f, g, h for functions, and $t, s, t_i, \ldots$ for first-order terms. First-order terms are constructed from constants and variables by (iterated) function application. This construction is described by the non-terminal `<term>` in the grammar in Section 2.1.

Definition 2.1 A *substitution* is a finite set of equations of the form $x_i = t_i$ where the variables on the left are distinct. Notation: $\sigma = \{x_1 = t_1, \ldots, x_n = t_n\}$. The equations $x_i = t_i$ in the substitution are called the *bindings* of the variables. Application of the substitution σ on a term t is the simultaneous replacement of all occurrences of x_i in t by t_i. The resulting term will be denoted by $\sigma(t)$.

As an example of substitution, consider $\sigma = \{x{=}f(x, y), y{=}c\}$, $t = g(h(x, y), x)$, then $\sigma(t) = g(h(f(x, y), c), f(x, y))$, $\sigma(\sigma(t)) = g(h(f(f(x, y), c), c), f(f(x, y), c))$.

Definition 2.2 A *unification problem* is a finite set of equations of the form $s_i = t_i$. Notation: $E = \{s_1 = t_1, \ldots, s_n = t_n\}$. A *solution* of a unification problem E is a substitution σ such that $\sigma(s_i)$ and $\sigma(t_i)$ are identical, for all $1 \leq i \leq n$. In that case σ is called a *unifier* of E. A set of equations $E = \{x_1 = t_1, \ldots, x_n = t_n\}$ is said to be in *solved form* if each variable x_i occurs exactly once in E. In that case E is a substitution itself and moreover a unifier of itself. It is even a so-called *most general unifier* (mgu): any other unifier can be obtained by further substitution of the variables possibly occurring in the t_i.

We follow the (very readable) treatment of unification by Martelli and Montanari [4, Algorithm 1 on p.251].[4] This is their simplest algorithm, but not the most efficient. It proceeds by transforming the unification problem, in a way not changing the set of unifiers, until a solved form is reached. If there is no unifier, the algorithm detects this and halts.

There are two important transformation steps in the algorithm. The first breaks down compound terms and the second substitutes a term for a variable:

1. Replace the equation $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ by $s_1 = t_1, \ldots, s_n = t_n$. The special case $n = 0$ amounts to deleting $c = c$ for a constant c.

[4] Notations vary. In [4], the application of a substitution σ to a term t is denoted by t_σ.

2. Let $x = t$ be an equation with x a variable and t any term not containing x. Let σ be the substitution $\{x = t\}$ The set of equations is transformed by replacing all equations $s = s'$ other than $x = t$ by $\sigma(s) = \sigma(s')$. Note that it allowed that t is a variable different from x. After this step the only occurrence of x in the set of equations is on the left of the equation $x = t$.

The transformations 1 and 2 transform a set of equations into an *equivalent* set of equations [4, Theorem 2.1 and 2.2]. In other words, they do not change the set of unifiers.

We now describe a non-deterministic unification algorithm in detail. Given a set of equations, perform any of the following steps until none of them applies (halt with success) or non-unifiability is detected (halt with failure):

(ee) Erase any equation of the form $x = x$.

(sw) Replace any equation of the form $t = x$ with t not a variable by $x = t$.

(tr) Apply term reduction to any equation of the form $s = t$ with s and t starting with the same function symbol.

(hf) Halt with failure if there is an equation of the form $s = t$ with s and t starting with different function symbols.

(su) Apply substitution with any equation of the form $x = t$ where x does not occur in t but does occur in another equation in the set.

(oc) Halt with failure if there is an equation of the form $x = t$ where x occurs in t but $x \neq t$.

The step in point (oc) above is called the "occur-check". If none of the steps (ee)–(oc) apply, then the set of equations must be in solved form and hence its own unifier. Since the steps (ee), (sw), (tr) and (su) transform the set of equations into an equivalent one, the solved form is a unifier of the original set of equations. For the same reason, the original unification problem is not solvable if the algorithm halts in step (hf) or (oc). Hence the algorithm is correct when it terminates, and termination is proved in [4, Theorem 2.3].

Example 2.3 We present six runs of the unification algorithm, with all steps named. As the algorithm is non-deterministic, alternative steps may be possible. The last two runs show that even the solved forms may be different. However, the solved forms are harmless variants in such cases.

1. $\{f(a,x) = f(y,b)\} \overset{tr}{\rightsquigarrow} \{a = y, x = b\} \overset{sw}{\rightsquigarrow} \{x = b, y = a\}$ solved form

2. $\{f(a,x) = f(x,b)\} \overset{tr}{\rightsquigarrow} \{a = x, x = b\} \overset{sw}{\rightsquigarrow} \{x = a, x = b\} \overset{su}{\rightsquigarrow}$
$\{x = a, a = b\} \overset{hf}{\rightsquigarrow}$ failure

3. $\{f(a, f(b, x)) = f(x, y)\} \overset{tr}{\rightsquigarrow} \{a = x, f(b, x) = y\} \overset{sw}{\rightsquigarrow} \{x = a, f(b, x) = y\} \overset{su}{\rightsquigarrow}$ $\{x = a, f(b, a) = y\} \overset{sw}{\rightsquigarrow} \{x = a, y = f(b, a)\}$ solved form

4. $\{f(x, x) = f(y, g(y))\} \overset{tr}{\rightsquigarrow} \{x = y, x = g(y)\} \overset{su}{\rightsquigarrow} \{x = y, y = g(y)\} \overset{oc}{\rightsquigarrow}$ failure

5. $\{x = y, y = z, z = x\} \overset{su}{\rightsquigarrow} \{x = y, y = z, z = y\} \overset{su}{\rightsquigarrow} \{x = z, y = z, z = z\} \overset{ee}{\rightsquigarrow}$ $\{x = z, y = z\}$ solved form

6. $\{x = y, y = z, z = x\} \overset{su}{\rightsquigarrow} \{x = y, y = x, z = x\} \overset{su}{\rightsquigarrow} \{x = y, y = y, z = y\} \overset{ee}{\rightsquigarrow}$ $\{x = y, z = y\}$ solved form

It should be said the occur-check can make the unification algorithm inefficient. For this reason many Prolog systems leave out the occur-check by default. This rarely leads to errors, but care has to be taken in using unification for theorem proving and type inference. There is usually a correct unification algorithm in the library:

```
?- unify_with_occurs_check(X,f(X)). % correct unification in SWIPL
false.
```

Exercise 2.4 Apply the unification algorithm to the following problems. In the last two, `H,T` are variables, `[]` is a constant and `[.|.]` a binary function.

1. $\{f(x, x) = f(y, a)\}$
2. $\{f(z) = y,\ g(y) = x, h(x) = z\}$
3. $\{f(z, z) = y,\ g(y, y) = x\}$
4. $\{f(x, h(x), y) = f(g(z), u, z),\ g(y) = x\}$
5. {`member(a,[a|[b|[]]]) = member(H,[H|T])`}
6. {`member(b,[a|[b|[]]]) = member(H,[H|T])`}

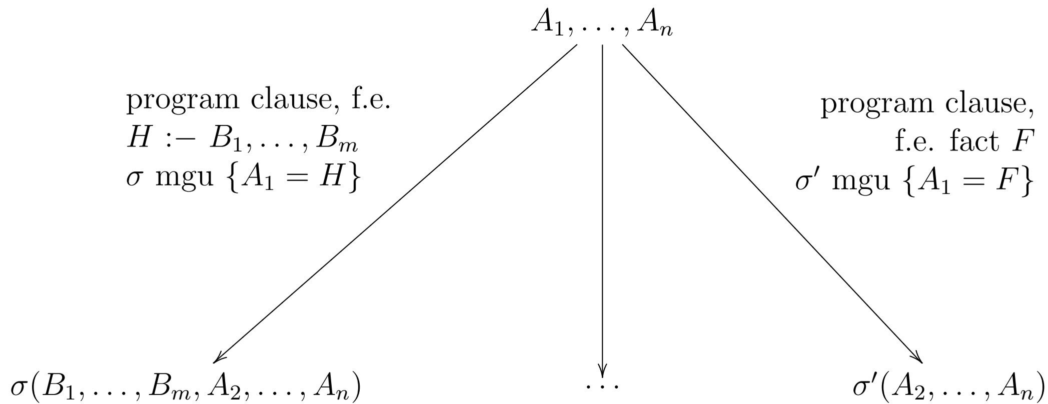
3 Prolog Semantics

Generally speaking, an operational semantics ('execution model') describes *how* a program is executed. A declarative semantics describes *what* is computed. In terms of the slogan Algorithm = Logic + Control, the declarative semantics is the logic and the operational semantics the control.

3.1 Operational Semantics

In order to formally describe the execution model we first define a tree for every logic program P and goal G. This tree is a labeled, ordered tree with a root. The nodes of the tree are labelled with goals and the label of the root is G. The outgoing edges of each inner node correspond to program clauses whose head can be unified with the leftmost atom in the goal. These outgoing edges are annotated by the program clause and the unifier applied. Application means here that the first atom in the goal is replaced by the atoms in the body of the program clause, after which the unifier is applied to the whole new goal. In the special case that the program clause is a fact the first atom in the goal is deleted before applying the unifier. Note that the atoms in a goal may share variables. This is an important form of data flow, and is the reason that the unifier must be applied to all atoms in the goal. Outgoing edges are ordered from left to right in the order of appearance of the corresponding program clause in P.

The situation of an inner node can thus be depicted in this way:



In the rightmost branch we have exemplified the possibility that a program clause is a fact, that is, there is no body $B_1, \ldots, B_m$. This is the only way a child ('subgoal') can be smaller (in terms of number of atoms) than the parent. This is important, since for solving a goal we need to solve all atoms in the goal.

There are two kinds of leaves in the tree. First, it is possible that in the picture above no program clause can be applied to A_1. In that case the goal $A_1, \ldots, A_n$ fails. The node has no children and is called a *failure leaf*. The other possibility is that there is no A_1 because the goal is empty. A node labeled with a empty goal (*notation* : $\Box$) is called a *success leaf*.

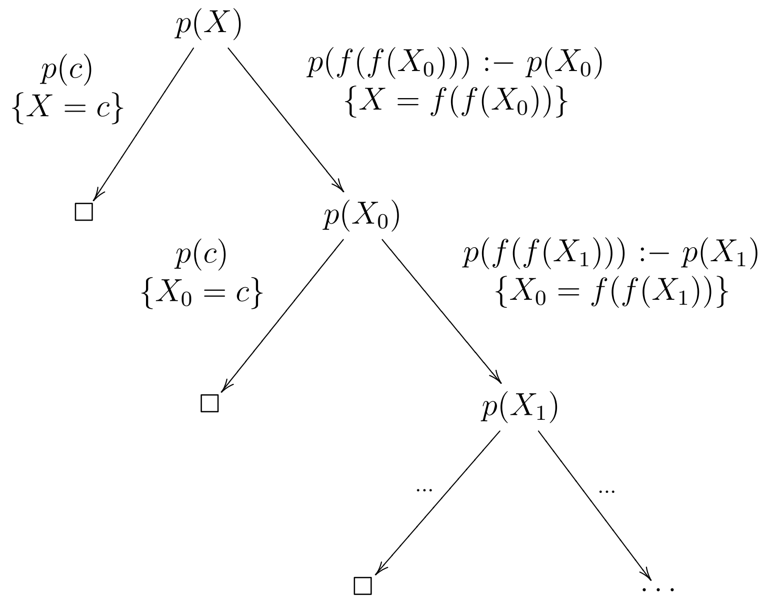
In this way the whole search tree is defined for any program P and goal G. Prolog traverses this tree depth-first from left to right ('backtracking') until a success leaf is found. If so, the composition of all the substitutions on the path to the root, restricted to the variables occurring in the original goal, is returned. If the whole tree has been traversed without success, failure is reported. Note that there is a clear possibility for non-termination if the tree has an infinite path.

Example 3.1 Consider program `royalty.pl` and goal `father(X,'Haakon')`. The search tree is depicted in Figure 1 on page 26, with `p` abbreviating `parent`. The binding `F='Harald V'` is returned.

Example 3.2 Consider the goal `p(X)` and the following program P:

```
p(c).
p(f(f(X))):- p(X).
```

We must first deal with a subtlety that hitherto has been ignored. Note that the variable `X` occurs both in the goal and in the second program clause. This would prevent unification of the goal with the head of the second program clause. However, the two occurrences of the variable `X` are in different scopes. The goal is asking the question: for which `X` can we infer `p(X)`? The second program clause represents the knowledge: for all `X` we have `p(f(f(X)))` if we have `p(X)`. In both cases the variable could be renamed without changing the question nor the knowledge represented. Since the goal is given by the user, Prolog silently renames variables in program clauses. Here is (part of) the tree:



In this case the tree is infinite. The success leaves represent all positive answers to the query `p(X)`. The answers are given as substitutions for `X` computed by composing all mgu's along the path from the success leaf to the root. In the tree above this means $X = c$ for the first success leaf found, $X = f(f(c))$ for the second, and so on. In general, the answers are of the form $X = f^n(c)$ for any even natural number n.

Exercise 3.1 Consider the following program and the goal `q(X)`.

```
p(c).
p(f(f(X))):- p(X).
q(X):- p(f(X)).
```

Draw the search tree and verify the answer(s) with Prolog Hint: `trace,q(X).`

Exercise 3.2 Consider the program `royalty.pl` and the following goal:

```
parent(X,'Haakon'),mother(Y,X).
```

Draw the search tree and verify the answer(s) with Prolog.

3.2 Declarative Semantics

For the practice of Prolog programming, the operational semantics from the previous section is most important.[5] However, given the theoretical ideal of LP we will describe in this section the declarative semantics of a logic program P. This takes two steps: first we define variable-free terms and variable-free atoms, then we define the variable-free atoms implied by the program. We restrict attention to programs consisting of clauses defined by the basic syntax in Section 2.1 and not involving predicates from the library ('built-ins'). We assume that goals do not introduce new names.

A *variable-free term* in the context of P is a special case of a first-order term as described by the non-terminal `<term>` in the grammar in Section 2.1. The extra requirements are that the term does not contain variables and that all names (constants and functions) occur in P. A *variable-free atom* in the context of P is as described by the non-terminal `<atom>` in the grammar in Section 2.1, with the extra requirement that all terms in `<terms>` are variable-free and all names (constants, functions and predicates) occur in P.

Example 3.3 In the context of the program in Example 3.2, the variable-free terms are $c, f(c), f(f(c)), \ldots$ and the variable-free atoms are $p(c), p(f(c)), p(f(f(c))), \ldots$. Note that there are infinitely many variable-free terms and atoms in this case.

Programs are finite, so without functions there can only be finitely many variable-free atoms, as is the case in the following example.

Example 3.4 In the context of `royalty.pl`, the variable-free terms are:

```
   'Martha'   'Mette-Marit'   'Maud'     'Sonja'
'Haakon VII'   'Olav V'    'Harald V'  'Haakon'
```

[5]Studying the declarative semantics could be postponed till after the next chapter on the pragmatics of Prolog.

There are eight variable-free atoms with each of the unary predicates `man/1` and `woman/1`. There are 64 variable-free atoms with each of the binary predicates `parent/2, father/2, mother/2`.

The last example shows that not all variable-free atoms are true. We are in particular interested in those variable-free atoms that are implied by the program P. These are collected in an iterative way. We first give an informal description. Start by collecting all variable-free atoms that are instances of facts in P. Then apply the rules of P to see if this extends the set of variable-free atoms collected so far. Iterate this process until no more variable-free atoms can be obtained. It it possible that we need infinitely many iterations, but also in that case the set of variable-free atoms becomes saturated in the sense that no more variable-free atoms can be obtained by applying the rules.

Now we give a more formal description. By induction we define an increasing sequence of sets P_i of variable-free atoms that are implied by P. Let P_1 be the set of all variable-free instances of facts in P. Assume P_i has been constructed ($i \geq 1$). Let P_{i+1} consist of all variable-free atoms in P_i plus all atoms A such that there exists a variable-free instance $A :- B_1, \ldots, B_n$ of a rule in P such that $B_1, \ldots, B_n$ are all in P_i. Define $P_\omega = \bigcup_{i \geq 1} P_i$. Then P_ω is saturated in the sense that no more variable-free atoms can be obtained by applying rules from P. This can be seen as follows. Let $A :- B_1, \ldots, B_n$ be a variable-free instance of a rule in P such that $B_1, \ldots, B_n$ are in P_ω. Then there exist $i_1, \ldots, i_n \geq 1$ such that $B_j \in P_{i_j}$ for all $1 \leq j \leq n$. Let $m = \max(i_1, \ldots, i_n)$, then $B_j \in P_m$ for all $1 \leq j \leq n$. Hence $A \in P_{m+1}$ and hence $A \in P_\omega$.

The set P_ω defined above is the declarative semantics of P. It is called the *minimal Herbrand model* and gives a precise description of the predicates defined by P. It can be shown that P_ω consists exactly of all variable-free atoms implied by P (implied in the sense of logical consequence). Ideally, one has that the operational semantics in the previous section and the declarative semantics in this section agree completely. This can be shown to be the case for LP in general with breadth-first search for success leaves in the trees in Section 3.1. However, for efficiency reasons Prolog searches depth-first, and leaves out the occur-check in the unification algorithm. Consequently, the operational semantics of Prolog is not always in perfect agreement with the declarative semantics, see the next example. We do have perfect agreement, however, for the program `royalty.pl` and for the program in Example 3.2. In general, it is important to be aware of what the predicates in your program mean declaratively.

Example 3.5 Recall the program in Example 3.3 and interchange the two program clauses to get the following program P:

```
p(f(f(X))):- p(X).
p(c).
```

Changing the order of the program clauses has no effect on the declarative semantics, which is for both programs $P_\omega = \{p(f^n(c)) \mid \text{even } n \geq 0\}$. (By convention, $f^0(c)$ is c.) However, the query `p(X)` leads to an ugly error here:

```
?- p(X).
ERROR: Out of local stack
   Exception: (246,060) p(_G984214) ?
```

What has happened is that, by interchanging the two program clauses, the left-right ordering of the search tree in Example 3.2 is reversed. This means that all success leaves now are on the right of the infinite path in the tree. As the tree is traversed depth-first, from left to right, no solutions are found. At the same time internal nodes are stacked and stacked, leading to stack overflow.

Exercise 3.3 Determine the declarative semantics of the program `royalty.pl` and verify your result with the following Prolog queries:

```
?- man(X).
?- woman(X).
?- parent(X,Y).
?- father(X,Y).
?- mother(X,Y).
```

4 Prolog Pragmatics

4.1 Programming Techniques

4.1.1 Generate-and-test

The built-in predicate `between/3` on integers is defined by:

```
between(N1,N2,N3) iff N1 <= N3 <= N2
```

It can also be used to generate integers in the third argument:

```
?- between(-1,2,X).
X = -1 ;
X = 0 ;
X = 1 ;
X = 2.
```

This makes it possible to implement the test from Section 1, whether or not a boolean function `p` takes value true on `0..99`:

```
exists:- between(0,99,X), p(X).
```

This is an example of a technique called *generate-and-test*. In its general form it works as follows. Assume we want to test whether there exists an element in some given domain that has some given property. One solution is to split this problem in two parts, one for generating elements of the domain and one for testing whether such an element has the desired property. In order to do this we need two predicates, which we call `domain/1` and `property/1` and we write:

```
test:- domain(X), property(X).
```

The technique becomes more interesting when the predicates have more parameters. For example, with two parameters it takes this form:

```
test(Par1,Par2):- domain(Par1,Par2,X), property(Par1,Par2,X).
```

Many problems can be cast in this form. Before we can give more examples we introduce lists and recursion in Prolog in the next section.

4.1.2 Recursion

Like in Haskell, a list in Prolog is either empty or is constructed from a head element and a tail list. The empty list in Prolog is denoted by `[]` and for extending a list with a head ("`:`" in Haskell) Prolog uses the infix binary function `[.|.]`. This leads to rather cumbersome notations for lists, such as `[a|[b|[c|[]]]]` (cf. `a:b:c:nil` in Haskell). Fortunately there are convenient shorthands defined by

```
[H1,...,Hn|Tail]=[H1|[H2,...,Hn|Tail]], [H1,...,Hn]=[H1,...,Hn|[]]
```

This makes it possible to write the above example as `[a,b,c]`. It makes it also possible to write `[a,b,c|D]` as a pattern matching any list starting with `a,b,c`.

An element occurs in a list if and only if it is the head or occurs in the tail. This leads to the following recursive definition of a predicate `member/2`:

```
member(Head,[Head|Tail]).
member(Head,[ X  |Tail]):- member(Head,Tail).
```

Logically these clauses are fine, but Prolog gives a useful warning when loading the above two clauses (warnings can be ignored, errors cannot):

```
Warning: ....pl:1: Singleton variables: [Tail]
Warning: ....pl:2: Singleton variables: [X]
```

This warns us that that the program contains variables that occur *only once in their clause*.[6] The fact that the second clause contains the variable `Tail` even twice does not change this fact, since variables in different clauses have different scopes. When a variable occurs only once in a clause, the only thing it does is matching a corresponding term in a goal, without doing something with the binding. This means that the name of the variable does not play a role. Prolog encourages in such cases the use of anonymous variables "_" ('wildcards'):

[6]Such a warning is extremely useful in cases where a variable name has been mistyped.

```
member(Head,[Head|_]).
member(Head,[_|Tail]):- member(Head,Tail).
```

The fact that member occurs in the body of the second clause makes the definition of this predicate recursive. Like in Haskell [2], recursion is the basic mechanism for looping.

Exercise 4.1 Draw the search trees for the following goals and verify the answers with Prolog.

```
member(c,[a,b,c]).
member(d,[a,b,c]).
member(X,[a,b,c]).
```

Using the predicate member/2 we can apply the generate-and-test technique for testing whether two lists have overlap:

```
overlap(List1,List2):- member(Elt,List1), member(Elt,List2).
```

This example demonstrates that the same predicate member/2 can be used both for generating candidates (members of the first list) and for testing (whether these candidates occur in the second list).

Exercise 4.2 Draw the search trees for the following goals and verify the answers with Prolog.

```
overlap([a,b,c],[d,c]).
overlap([a,b,c],[d,e]).
overlap([a,b,c],[d,E]).
```

Appending two lists yields a uniquely defined third list. If the first list is empty, the third list equals the second. Otherwise the first list has a head and a tail, and appending is done at the tail after which the head can be added. This leads to the following recursive predicate append:

```
append([],L,L).
append([H|T],L,[H|TL]):- append(T,L,TL).
```

Clearly, append/3 is functional in the sense that the third argument is a function of the first two arguments. However, the formulation of append as a predicate has the unexpected use illustrated in the second goal in the next exercise. In a predicate, the roles of arguments as input or output need not be fixed.

Exercise 4.3 Find all solutions to append([a,b],[c],L) and append(L1,L2,[a,b,c]). Draw the search trees and verify the answers with Prolog.

The last exercise shows that append can be used to compute prefixes and suffixes. This is made explicit by:

```
prefix(Prefix,L):- append(Prefix,_,L).
suffix(Suffix,L):- append(_,Suffix,L).
```

4.1.3 Accumulators

Reversing a list can be done recursively as follows. If the list is empty, there is nothing to be done. If the list consists of a head and a tail, reverse the tail and append the head as a the one-element list. The following Prolog program does the job:

```
naive_rev([],[]).
naive_rev([H|T],RevTH):- naive_rev(T,RevT), append(RevT,[H],RevTH).
```

However, this program is very inefficient, see the following exercise.

Exercise 4.4 Draw the search trees for the goal `reverse([a,b,c],L)`. Test the following goals with Prolog and compare the timings:

```
time(naive_rev([a,a,a,a],L)).
time(naive_rev([a,a,a,a,a,a,a,a],L)).
```

The reason for the inefficiency of `reverse` is that appending an element to the end of a list is linear in the length of the list, and `reverse` does this a linear number of times. As a consequence, `reverse` takes time quadratic in the length of the list, whereas linear time should be sufficient. Clearly, a more efficient way of reversing a list is desirable. It seems natural to go through the list and stack the elements we meet on our way. When meeting the end of the list, the stack can be emtied in the reverse order. This is the way it would be done in IP, but that would require a stack in memory, changing state at each step. (In IP, reversing a doubly linked list can be done in constant time by interchanging the pointers to the begin and the end of the list!) In LP and FP we do not have state, but we have recursion (implemented by using a stack) and parameters. A first attempt to use these would result in:

```
stack([],Stack):- write('stack: '), write(Stack),nl.
stack([H|T],SoFar):- stack(T,[H|SoFar]).
```

The search trees for a goal such as `stack([a,b,c],[])` is simple and linear, with subsequent subgoals `stack([b,c],[a])`, `stack([c],[b,a])`, `stack([],[c,b,a])`. Building a stack in the second argument seems fine:

```
stack: [c, b, a]
true.
```

But how to get out the reversed list? For this we need to give the predicate an extra argument which is simply passed to the recursive call and which returns the result in the base case. This leads to:

```
reverse(L,RevL):- revapp(L,[],RevL).
revapp([],Output,Output).
revapp([H|T],Accumulator,Output):- revapp(T,[H|Accumulator],Output).
```

The extra (middle) argument of `revapp` is called an *accumulator*, as it so to say accumulates the result. Note that `revapp` actually appends the second list to the reversed first list. It is instructive to closely compare `revapp` with `append` by using the same variable names:

```
append([],L,L)
revapp([],L,L).
append([H|T],L,[H|TL]) :- append(T,L,TL).
revapp([H|T],L,RevTHL) :- revapp(T,[H|L],RevTHL).
```

The predicate `append` has to remember `H` until the recursive call returns and the list `[H|TL]` can be constructed. The predicate `revapp` doesn't have to do anything after the recursive call. The latter type of recursion is called *tail recursion* and can be implemented more efficiently without stacking recursive calls.

Exercise 4.5 Test the following goals with Prolog and compare the timings:

```
time(naive_rev([a,a,a,a,a,a,a,a],L)).
time(  reverse([a,a,a,a,a,a,a,a],L)).
```

4.1.4 Arithmetic

Prolog has the usual arithmetical operators `+, -, *, /`, besides `//, mod` for integer division and remainder. Arithmetical equality and disequality are denoted by `=:=` and `=\=`, respectively. Inequalities in Prolog can be expressed by `>, <, =<, >=`. All these can only be evaluated if all operands evaluate to a number. None of these operators/relations gives the possibility to evaluate an arithmic expression and bind a variable to the result. It is exactly this what the operator `is/2` does. For example, the query `X=1+1, Y is 2*X` binds `X` to `1+1` and `Y` to `4`. Note that `=` denotes unifiability and not equality.

Exercise 4.6 Test the following five goals with Prolog:

```
 1+1 = 2        X = 1+1        X is 1+1        0*X = 0       0*X =:= 0
```

4.1.5 Cut, Negation-as-Failure

The "cut" is an operational device in Prolog that prunes the search tree. The cut is denoted by `!` and is true, but passing it has as side-effect that specific parts of the search tree are cut out.

For example, try the query `member(a,[a,b,a])`. This query succeeds two times, and thereafter considers the alternative `member(a,[])` before it finally fails (draw the search-tree!). Assume we are interested in a test whether an element occurs in a list, but not in alternative occurrences. Consider this program:

```
membership(H,[H|_]):- !.
membership(H,[_|T]):- membership(H,T).
```

The effect of the cut in the first clause is that the second clause is not used for searching for other occurrences when the cut has been passed. We get the following behaviour:

```
?- membership(a,[a,b,a]).
true.
```

In general, the side-effect of passing a cut is that all remaining alternatives to atoms left of it, including the head of the clause, are discarded from the search tree. When backtracking arrives at the cut, control jumps to the caller of the head of the clause (difficult!). A so-called *green* cut prunes parts of the search tree in which no solutions are found. In contrast, a *red* cut possibly discards solutions. The cut above is a red cut, discarding solutions that we are not interested in. The following example is the use of a red cut to approximate the negation of a predicate:

```
notp(X):- p(X), !, fail.
notp(_).
```

This example of a red cut is called *negation-as-failure.* The query `notp(t)` fails if `p(t)` succeeds. Furthermore, `notp(t)` succeeds if `p(t)` terminates with failure. Note that the query `notp(t)` does not terminate if `p(t)` does not terminate. This form of negation based on finite failure is Prolog's built-in negation operator `\+`. An example of the use of a green cut can be found at the end of the next section.

4.2 Data structures

4.2.1 Terms as data structures

One way of viewing predicates is as tables of a relational database (without explicit names of the attributes). The name of the table is the name of the predicate. A tuple in a table corresponds to a variable-free atom:

```
parent('Sonja','Haakon').
```

In general, tuples can be represented as fixed-length argument lists. Functions make it possible to represent data structures whose size is not known beforehand. The best known such data structure, both in LP and FP, is the list. Lists are represented by means of a binary function and a constant. In Prolog, standard lists use `[.|.]` and `[]` and list notation is right-associative (e.g., `[a|[b|[c|[]]]]`). It would be possible to define lists in a left-associative way (e.g., `[[[[]|c]|b]|a]`). The right-associative standard clearly favours left-to-right reading and supports easy extension of lists on the left.

One may argue that lists do not exploit the full potential of a binary function, which makes it possible to represent binary trees in a natural way (e.g., `[[]|[[[]|[]]|[]]]`). If we only use the constant `[]`, such representations have only form and no place for content. Using different constants, we can at least have content in the leaves of the binary tree (e.g., `[a|[[b|c]|d]]`). It is therefore natural to look at ternary functions to represent binary labelled trees, inspired by the Haskell type:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Here `Nil` represents the empty tree and `Node` is a ternary function for combining two subtrees below a new labelled root. For example, we have the following values of type `Tree Int`

```
Node(2,Nil,Nil)
Node(1,Node(2,Nil,Nil),Node(3,Node(4,Nil,Nil),Nil))
```

Of course, the verbose representation `Node(2,Nil,Nil)` of a leaf could be prevented by a constructor `Leaf a` instead of `Nil`, in particular if no representation of the empty tree is needed.

Exercise 4.7 Draw the abstract syntax trees of:

```
[a|[b|[c|[]]]]
[[[[]|c]|b]|a]
[[]|[[[]|[]]|[]]]
[a|[[b|c]|d]]
Node(1,Node(2,Nil,Nil),Node(3,Node(4,Nil,Nil),Nil))
```

It is only a little step to change the syntax to Prolog and to write a predicate for testing whether a term occurs in a tree:

```
member(E,node(E,_,_).
member(E,node(_,Left,_):- member(E,Left).
member(E,node(_,_,Right):- member(E,Right)
```

Search trees have the property that in any node the label is larger than any label in the left subtree and smaller that any label in the right subtree. If a search tree is balanced, that is, left and right subtrees have more or less the same number of nodes, binary search is faster:

```
membership(E,node(E,_,_):- !.
membership(E,node(X,Left,_):- E<X, !, membership(E,Left).
membership(E,node(X,_,Right):- E>X, membership(E,Right).
```

The cuts are green cuts here, provided the tree is a search tree.

4.2.2 Open data structures

Open data structures are data structures with 'holes'. If you keep track of the holes, you can fill them and thereby extending the data structure. If the extension itself has a hole, it can again be extended. In Prolog, a 'hole' is represented by a variable. Open data structures are first-order terms with variables, as opposed to variable-free or *closed* terms. The difference with the is made clear by comparing the following two:

```
closed list: [a,b,c|[]]        open list: [a,b,c|X]
```

Where the closed list has `[]`, the open list has a variable. To extend the closed list, one has to replace `[]` by some list. The only way to reach `[]` is to go through the whole first list. This is basically what `append` does. To extend the open list, one has to replace `X` by some list. This can be done by simply binding `X` to the new list, but it requires having direct access to the variable `X`.

The following nice example is from [3], based on an problem of Dijkstra, called the Dutch national flag problem. Assume we have a list of objects that are either red, white or blue. These objects should be sorted in the following way: first the red objects, then the white objects and finally the blue objects.

Exercise 4.8 Given the predicates `red/1, white/1, blue/1`, write a predicate `dnf1/4` such that a call `dnf1(Objects,Reds, Whites, Blues)` sorts the list `Objects` in the lists `Reds, Whites, Blues`.

The solution to the above exercise allows us to solve the Dutch national flag problem in the following way:

```
dnf1(Objects,RedsWhitesBlues):-
  dnf1(Objects,Reds, Whites, Blues),
  append(Reds,Whites,RedsWhites),
  append(RedsWhites,Blues,RedsWhitesBlues).
```

It is the use of `append` which makes this solution less efficient. Using open lists, [3] gives a solution which is more efficient (and close to Dijkstra's original solution):

```
dnf2(Objects,RedsWhitesBlues):-
  dnf2(Objects,
       RedsWhitesBlues,WhitesBlues,
       WhitesBlues,Blues,
       Blues,[]).

dnf2([],R,R,W,W,B,B).

dnf2([Item|Items],R0,R,W0,W,B0,B):-
  colour(Item,Colour),
```

```
  dnf2(Colour,R0,R,W0,W,B0,B,Item,Items).

dnf2(red,[Item|R1],R,W0,W,B0,B,Item,Items):-
  dnf2(Items,R1,R,W0,W,B0,B).

dnf2(white,R0,R,[Item|W1],W,B0,B,Item,Items):-
  dnf2(Items,R0,R,W1,W,B0,B).

dnf2(blue,R0,R,W0,W,[Item|B1],B,Item,Items):-
  dnf2(Items,R0,R,W0,W,B1,B).
```

Exercise 4.9 Examine carefully what happens when we query `flag` after having added the following clauses:

```
colour(r,red). colour(w,white). colour(b,blue).
flag :- trace, dnf2([r,w,b,r,w,b,b,r,r,w,w],_).
```

Acknowledgements

This text has greatly benefitted from comments and suggestions by John Fisher, Dag Hovland and Richard O'Keefe.

References

[1] R. Kowalski. *Algorithm = Logic + Control.* CACM **22**(7):424–436, 1979.

[2] G. Hutton. *Programming in Haskell.* CUP, 2007.

[3] R.A. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[4] A. Martelli and H. Montanari. *An Efficient Unification Algorithm.* ACM Transactions on Programming Languages and Systems, **4**(2):258–282, 1982.

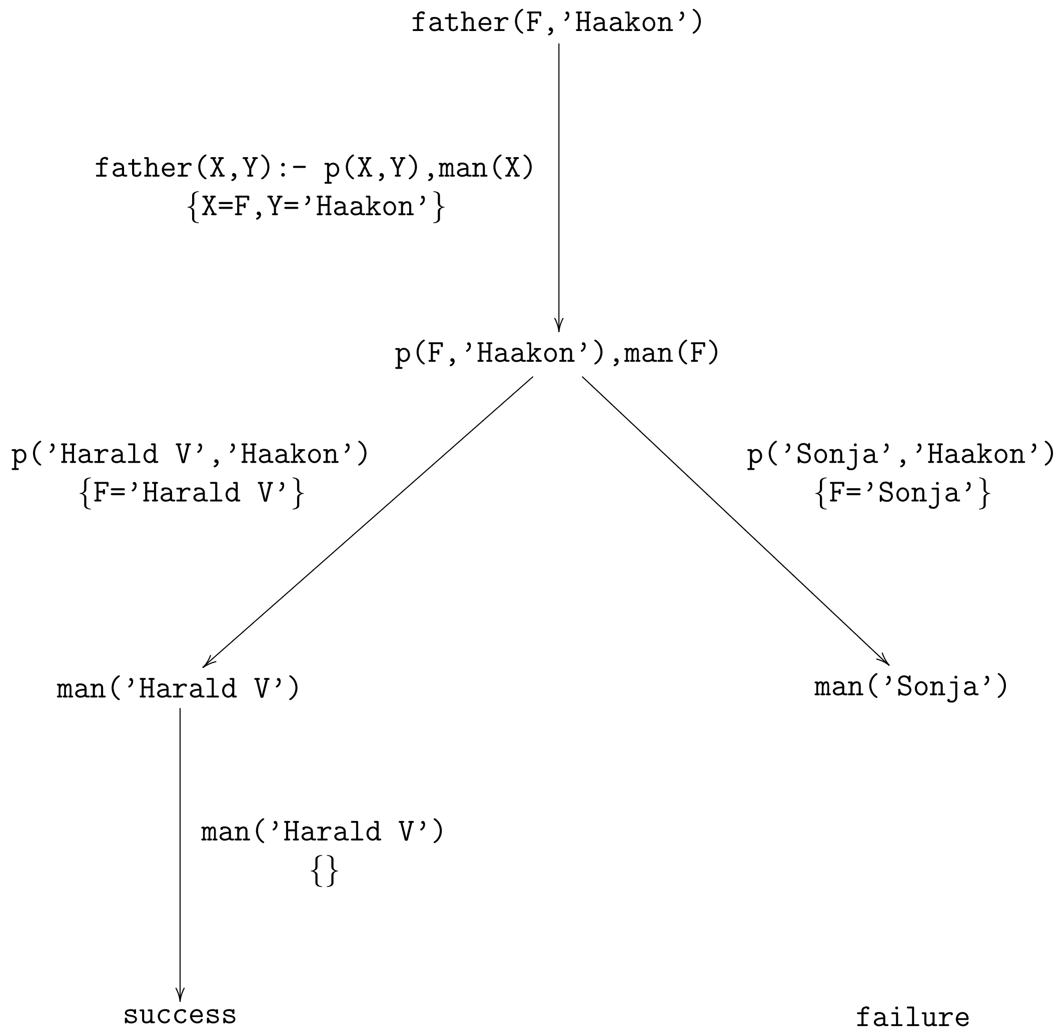


Figure 1: Seach tree for `royalty.pl` and goal `father(X,'Haakon')`.