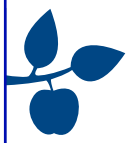

SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

DM509: Programming Languages

An Introduction to PROLOG

Torben Nielsen
tkn@imada.sdu.dk

Rolf Fagerberg
rolf@imada.sdu.dk

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Contents

Chapter 1

Introduction

1.1. About Prolog

Essentially Prolog is a system that performs automated deduction (within a restricted area of logic) made into a programming language. Languages in this vein are called *declarative* programming languages, and programming in such a language is called *declarative* or *logic*[1] programming.

Prolog is a high level language that is especially suited for specialized applications within specific areas, such as expert systems, language parsing and processing, symbolic computation, AI, and exhaustive searches.

Prolog (and other declarative programming languages) is different to "conventional" programming languages (such as C and Java) in several ways. While imperative programs usually consist of a sequence of statements, a Prolog script consists a list of facts (often referred to as the "database") and deductive rules (collectively referred to as *clauses*).

An execution of a Prolog script is an systematic attempt (performed by the Prolog system) to logically deduce a statement (a *goal*) given by the programmer using the facts and rules of the script.

In addition to these striking differences, Prolog is basically untyped (actually, a very simple form of dynamic typing is used).

1.2. GNU Prolog

The Prolog system installed on IMADA's system is GNU Prolog[2].

GNU Prolog is an interpreter, although compilation of a Prolog script to a standalone program is possible.

GNU Prolog can be invoked from a terminal with the command `gprolog`. Once `gprolog` is running, a script can be (re)loaded by issuing the command `['script.pl'].`[3].

[1] This, despite the fact that many programmers used to imperative or object oriented programming tend to find declarative programming anything but logical!

[2] See `http://pauillac.inria.fr/~diaz/gnu-prolog/` for the official manual and further information.

[3] Prolog scripts conventionally have the extension `.pl`.

Note that invoking `gprolog` with a script file as an argument (e.g. `gprolog script.pl`) will (rather annoyingly) not load the script. If you are using the `tcsh` shell, this can be remedied by inserting the following line into your `.tcshrc` file.

```
alias gp 'gprolog --query-goal "['\\\'\!\*\\\'']"'
```

This allows you to start `gprolog` and load a script by invoking the command `gp script.pl`.

Chapter 2

Basic PROLOG Syntax

This chapter covers the basic syntax of PROLOG from the bottom up. Examples of complete PROLOG scripts are given in Chapter 4.

The basic syntax of PROLOG consists of *terms*. Terms can be either *constants*, *variables* or *structures*. These are combined into clauses and facts which make up a PROLOG program.

2.1. Constants

Constants are either alphanumeric strings starting with a lowercase letter (underscores are allowed)[1] or a number (in GNU PROLOG[2] these are divided into *integers* and *floating point numbers* or *floats* – both written with conventional syntax).

Constants (even if they are written as numbers) have no meaning except as identifiers. Because of this, we will consider constants written as numbers as having *number syntax*, rather than being numbers. A constant with number syntax may, if requested by the programmer, be interpreted as an actual number for certain purposes.

The stock data types of strings and characters may be represented as constants in PROLOG (with characters being constants of length one). Constants without number syntax are called atoms and constants in general are called atomic.

Some examples of allowed constants in GNU PROLOG are shown in Figure 2.1.1.

Strings	`george`, `a`, `you_may_use_underscore`, `'4youEye$only'`
Integers	`234`, `-42`
Floats	`34.67`, `-3.8`, `34.2e-4`

Figure 2.1.1: Examples of valid constants in GNU PROLOG.

[1] If surrounded by single quotes, any symbols are allowed in the string.
[2] See section 1.2 for more about GNU PROLOG.

2.2. Variables

Variables are alphanumeric strings starting with an uppercase letter (underscores are allowed).

Variables can be bound to other terms through *unification*[3] during execution. Initially in an execution, they are unbound.

The following strings are examples of allowed variables in: `X`, `Result`, `NumberOfReports`, `My_tax`.

2.3. Structures

Structures consist of an atom (referred to as the structure's *functor*) and zero or more subterms (referred to as the structure's *arguments* or *components*) in parentheses.

Structures have no intrinsic meaning beyond structural. The may be used both as Boolean functions (called *predicates*) and as data structures (called *records*).

The number of subterms in a structure is called the arity of the structure. Nullary structures (structures that have arity 0) use no parentheses, i.e. they are just atoms.

The following are examples of valid structures in GNU Prolog.

```
peter
owns(peter,volvo_S60)
owns(peter,X)
owns(X,volvo_S60)
```

2.4. Clauses

Clauses define Boolean functions (called *predicates*) by specifying what makes the function true. Clauses are written as follows.

`S :- S1, S2, S3, S4, ..., Sn.` *(note the trailing period!)*,

where $n \geq 0$, and S and all the S_i's are either structures or variables that are instantiated to structures at runtime.

For $i \geq 1$, a clause is called a *rule*. The left hand side is called the *head* of the clause, and right hand side is called the *body*.

If i = 0, a clause may be written simply `S.` and is then called a *fact*.

The commas used in a rule, denote the logical conjunction of the structures on the right hand. Thus, in our example above, S is true if all the S_i's are true.

A single predicate may have several clauses. The predicate is then considered true if at least one of the clauses for it is true, i.e. the Boolean value of a predicate is the logical disjunction of all its clauses.

The following are examples of valid clauses in GNU Prolog:

[3] The process by which Prolog seeks to fulfill a goal. See section 3.1.

```
alice.
owns(peter,volvo_S60).
owns(alice,X).
owns(peter,X) :- car(X),smart(X).
```

Here, `alice` is a nullary predicate and `owns` is a binary predicate. This is expressed by the notation `alice/0` and `owns/2`. Note that `predname/2` and `predname/3` are different predicates!

2.5. Operators

Operators in PROLOG are binary structures where the functor is written using infix notation. Unary operators with prefix notation (but no parentheses) exist too.

Noteworthy built-in operators are the standard arithmetic operators (`+`, `-`, `*`, etc.) and the *list constructor* `.` (a period).

The identifier `[]` is a special constant used to denote the empty list. Syntactic sugar/notational convenience is available for working with lists, e.g. `.(3,.(4,[]))` is the same as `3.(4.[])` which can be written as `[3,4]`, and expressions such as `.(Head,Tail)` may be written as `[Head|Tail]`.

In addition it is possible to use the conventional infix notation of arithmetic operators, e.g. `*(6,+(8,9))` can be written as `6*(8+9)`. Note that the last structure is different from the constant 102!

2.5.1. Comparing Terms

The usual comparison operators (<,>,=, etc.) are also available in PROLOG, albeit with a slightly different syntax. The following table shows all the built-in comparison operators and their meaning.

`X == Y`	`X` is equal to `Y`
`X \== Y`	`X` is not equal to `Y`
`X @< Y`	`X` is less than `Y`
`X @=< Y`	`X` is less than or equal to `Y`
`X @> Y`	`X` is greater than `Y`
`X @>= Y`	`X` is greater than or equal to `Y`

Note that although the comparison operator `==` may (for some uses) seem interchangeable with the unification operator `=`, the actual meaning is vastly different!

Terms in `gprolog` are completely ordered (i.e. any two terms may be compared!) in the following manner (least to greatest):

- Variables (oldest first).
- Floating point numbers in numeric order.
- Integers in numeric order.

- Atoms in alphabetical (i.e. character code) order.
- Compound terms ordered first by arity, then by the name of the principal functor and by the arguments in left-to-right order.

2.6. Comments

Comments in PROLOG are written between `/*` and `*/` for comments spanning multiple lines or after `%` for single line comments.

Chapter 3

Basic PROLOG Concepts

3.1. Unification

Unification is the process by which the PROLOG system tests (by means of pattern matching) if two structures can possibly be equal. If parts of the structures compared are variables, successful unification *instantiates* (assigns a value to) the variables in question.

This is perhaps easier to understand by considering the unification of two structures:

`[1,peter,X,f(Y,20)]` and `[Z,peter,alice,f(2,20)]`.

These will unify successfully, since the variable `Z` can be equal to the constant `1`; the constant `peter` is of course equal to itself; the variable `X` can be equal to the constant `alice`; and the structure `f(Y,20)` can be equal to the structure `f(2,20)`, since the variable `Y` can be equal to the constant `2`.

As a result of this unification, the following instantiations will occur: `X = alice`, `Y = 2` and `Z = 1`, in accordance with the above reasoning.

It is important to note that **assignments in PROLOG are made by instantiation through successful unification.**

An unsuccessful unification attempt, for instance between the structures `f(X,X)` and `f(1,2)`, does not instantiate any variables.

It is possible to unify two variables, either directly or as a result of unification between more complex structures. These will then *co-refer*, meaning that they must have the same value at all times. Thus, if one of the variables is instantiated, the other will be as well.

A final noteworthy example is unification of lists formed from the `.` or `|` operators. Consider for instance the lists `[Head|Tail]` (shorthand for `.(Head,Tail)`). These will unify successfully with `Head = 1` and `Tail = [2,3,4]`. Thus, the structure `[Head|Tail]` allows us to split a list into its head and tail.

3.2. Goals, Satisfaction and Backtracking

Goals are Boolean questions asked by the user (e.g. through the prompt in `gprolog`). More precisely, goals are Boolean statements of the same form as the right hand side of a rule, which the user would like to know if can be derived as true, using the facts and rules of a given PROLOG script.

The PROLOG system will try to *satisfy* a goal by attempting to unify the goal with the clauses of the script. The clauses are tested for unification with the goal one by one, starting with the topmost clause.

If the goal is successfully unified with a fact then the goal is satisfied. If the goal unifies with a rule, the goal can be satisfied if the body of the rule can be. The goal is therefore substituted by the body of the rule. The terms of body are now subgoals, which the system will try to satisfy one by one from left to right in the same manner as the original goal.

In general, the current goal will be the logical conjunction of several subgoals. The PROLOG system attempts unification between the left-most of these and the head of a clause in the script. If unification is possible, the body of this clause is substituted for the subgoal – in particular, if the clause in question is a fact, the list of subgoals is shortened, since the body of a fact is empty.

If the list of subgoals vanishes, the original goal has been satisfied, i.e. a derivation of it has been found – it has been proven true from the facts and rules given by the script.

If a subgoal cannot be satisfied *backtracking* will take place. Backtracking is the process by which the PROLOG system undoes previous unifications, thereby unsatisfying previously satisfied subgoals, such that they be attempted satisfied in a different way. Instantiations that occurred as a result of unifications that are undone during backtracking are also undone, i.e. variables there were instantiated as a result of these unifications become uninstantiated during backtracking. Naturally, variables can be reinstantiated during the further search process.

The search procedure performed by the PROLOG system is equivalent to a depth first search in a tree defined by the clauses of the script and the instantiations of variables during the search.

As an example of this process, consider the following script.

```
1  a(X,Y,Z) :- b(X,Y),d(Z).
2
3  b(X,X) :- c(X).
4  b(1,2).
5
6  c(3).
7  c(4).
8
9  d(5).
10 d(6).
```

Assume that we ask the PROLOG system to resolve the goal `a(X,Y,Z)` using the above script. The system will then perform the search shown in figure 3.2.1

In the figure, nodes represent the current goal and edges represent successful unifications of the first subgoal of the current goal with the head of a clause. On each edge, the equations

of the solved form corresponding to the unification are listed.

Note that in each clause of the code, the variables are local variables, hence new unique variable names are introduced at each unification with the head of a clause of the code.[1]

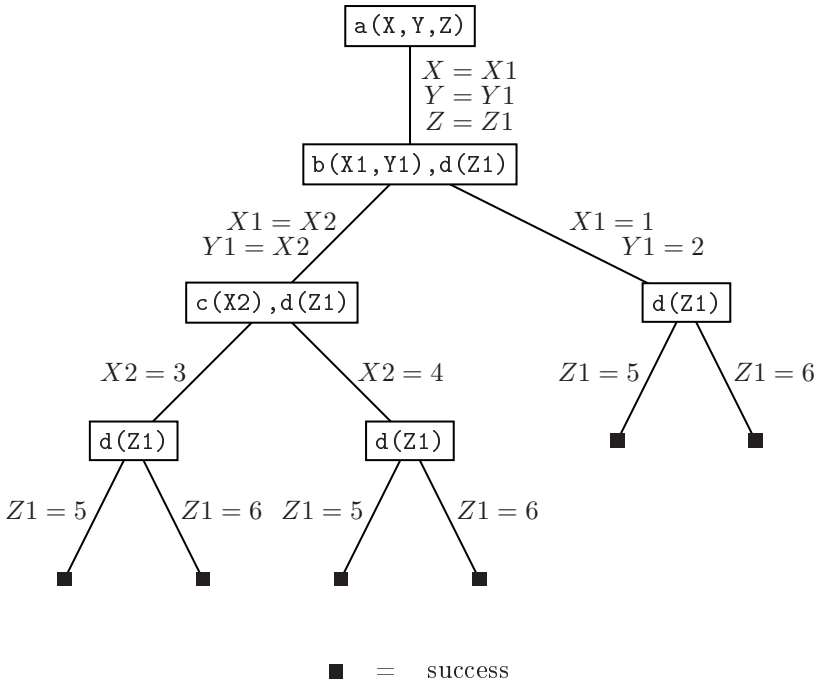

Figure 3.2.1: The search tree produced by the PROLOG system when trying to resolve the goal `a(X,Y,Z)` using the above script. The nodes represent the current goal and the edges represent successful unifications, the results of which are listed on the corresponding edge.

When the search reaches a successful leaf, the calculated answer is the solved form of the set of equations on the path from the leaf to the root. However, only the equations for the variables occuring in the original goal are reported. Hence, the six possible reported answers are:

X:	3	3	4	4	1	1
Y:	3	3	4	4	2	2
Z:	5	6	5	6	5	6

Note, GNU PROLOG stops at first solution found and display the instatianted variables (if any). The user may request another solution by entering `;` at the prompt. In addition, all solutions may be requested by entering `a`. Further answers are produced by backtracking and continuing the search.

[1] The need for local variables is clear if we consider an initial goal of e.g. `a((X,Y),Z,Q)`, where the variables `X` and `Y` in the goal and in the head of the first clause clearly are independent.

Backtracking is discussed in detail in Chapter 5. For now tho, we will proceed to study some small PROLOG scripts that make use of what we have seen so far.

Chapter 4

Learning by Doing

This chapter contains several examples of PROLOG scripts. Some of the examples illustrate useful programming techniques. It is recommended that you read the scripts and the provided output, and try to follow the derivation procedure.

4.1. A Simple Script

```
1  /* A simple script about (heterosexual) couples: */
2
3  male(peter).
4  male(paul).
5  female(beatrice).
6  female(alice).
7  female(clepatra).
8
9  diffSex(X,Y) :- male(X),female(Y).
10
11 possibleCouple(X,Y) :- diffSex(X,Y), likes(X,Y), likes(Y,X).
12
13 likes(peter,alice).
14 likes(alice,peter).
15 likes(paul,alice).
16 likes(paul,beatrice).
17 likes(paul,clepatra).
```

The following is a transcript of a `gprolog` session using the above script. See if you can follow the derivation procedure performed by `gprolog` for each query.

```
Q:

| ?- female(alice).

A:

yes

Q:

| ?- female(Y).

A:

Y = beatrice ? ;

Y = alice ? ;

Y = clepatra

yes

Q:

| ?- likes(paul,X).

A:

X = alice ? ;

X = beatrice ? ;

X = clepatra

yes

Q:

| ?- likes(cleopatra,X).

A:

no

Q:

| ?- diffSex(X,Y).

A:

X = peter
Y = beatrice ? ;

X = peter
Y = alice ? ;

X = peter
Y = clepatra ? ;

X = paul
Y = beatrice ? ;

X = paul
Y = alice ? ;

X = paul
Y = clepatra

yes


Q:

| ?- possibleCouple(X,Y).

A:

X = peter
Y = alice ? ;

no
```

4.2. Predicates as Functions

```
/* The predicate myAppend(L1, L2, L3) is true if and only if
the concatenation of L1 and L2 is equal to L3 */

myAppend([],L,L).
myAppend([X|L1],L2,[X|L3]) :- myAppend(L1,L2,L3).
```

The following is a transcript of a `gprolog` session using the above script.

```
Q:

| ?- myAppend([1,2],[3],[1,2,3]).

A:

yes

Q:

| ?- myAppend([1,2],[3],[1,2,4]).

A:

no

Q:

| ?- myAppend([1,2],[3],L3).

A:

L3 = [1,2,3].

Q:

| ?- myAppend(L1,[3],[1,2,3]).

A: L1 = [1,2]

Q:

| ?- myAppend(L1,L2,[1,2,3]).

A:

L1 = []
L2 = [1,2,3]

L1 = [1]
L2 = [2,3]

L1 = [1,2]
L2 = [3]

L1 = [1,2,3]
L2 = []

Q:

| ?- myAppend(L1,L2,L3).

A:

L1 = []
L3 = L2 ? ;

L1 = [A]
L3 = [A|L2] ? ;

L1 = [A,B]
L3 = [A,B|L2] ? ;

L1 = [A,B,C]
L3 = [A,B,C|L2] ? ;

L1 = [A,B,C,D]
L3 = [A,B,C,D|L2] ? ;

L1 = [A,B,C,D,E]
L3 = [A,B,C,D,E|L2] ? ;

L1 = [A,B,C,D,E,F]
L3 = [A,B,C,D,E,F|L2] ?

etc....
```

As seen in the above transcript, predicates can be used as functions (with results created through unification). The return value of this sort of function is one of the arguments. For many predicates, the situation is as seen in the transcript; all arguments may potentially serve as the return value.

4.3. Performing Arithmetic Operations

As the following GNU PROLOG script shows, it is possible to interpret constants with number syntax as actual numbers and use these in arithmetic operations.

```
/* An example finding N factorial (the predicate factorial(N,F) is true if
(N is instantiated and) F is N factorial). */

factorial(0,1).

factorial(N,F) :-
        N>0,
        N1 is N-1,
        factorial(N1,F1),
        F is N * F1.
```

Here, `>` is a built-in predicate written as an infix operator. The arguments of `<` must be (variables instantiated to) structures which can be interpreted as arithmetic expressions (e.g. 2*3+4). In the case of variables, these must be instantiated at the time satisfaction is attempted. Note that `N-1` is a structure, not a number.

The built-in predicate `is` evaluates such a structure (any structure which can be interpreted as an arithmetic expression) and unifies the result with the left argument.

The following is a transcript of a `gprolog` session using the above script.

```
Q:

| ?- factorial(4,24).

A:

true

Q:

| ?- factorial(4,25).

A:

no

Q:

| ?- factorial(3,F).

A:

F = 6

Q:

| ?- factorial(N,6).

A:

An error message, because
variables taking part in '>'
and 'is' are not instantiated.
```

4.4. Writing Output

```
1  /* Programming example: Towers of Hanoi */
2
3  hanoi(N) :- move(N,left,centre,right).
4
5  /* Arguments: Number of discs, source disc, destination disc, spare disc */
6
7  move(0,_,_,_).
8  move(N,A,B,C) :-
9          N >= 1,
10         M is N-1,
11         move(M,A,C,B),
12         inform(A,B),
13         move(M,C,B,A).
14
15 inform(X,Y) :- write(X),write('->'),write(Y),nl.
```

Here, `write/1` is a built-in predicate which succeeds once, and as a side effect prints a representation of its argument on screen. The predicate `nl/0` similarly prints a newline.

The following shows a test run of the above script using `gprolog`.

```
1  | ?- hanoi(3).
2  left->centre
3  left->right
4  centre->right
5  left->centre
6  right->left
7  right->centre
8  left->centre
```

Chapter 5

Backtracking and...Cut!

5.1. Backtracking Revisited

We now briefly return to the backtracking process. Consider the following PROLOG script.

```
1  a :- c,b.
2  a :- c.
3
4  b :- e.
5  b :- e,c,f.
6  b :- c.
7
8  c.
9  c.
10
11 e :- fail.
12 e.
13
14 f.
15 f.
16 f.
```

For the sake of clarity, no variables occur in this example, hence no instantiations will occur (unification, however, still occurs).

We will now consider the search tree generated by PROLOG when asked to resolve the goal `a`. This tree is shown in Figure 5.1.1 Note that the predicate `fail/0` will not unify with anything (it always fails), which shows that leaves in the tree do not necessarily indicate successes. Try to follow the satisfaction process.

5.2. Making the Cut

While the ability of the PROLOG system to backtrack may be extremely useful, there are still situations where we wish to avoid (or control) this. For this, PROLOG provides the built in

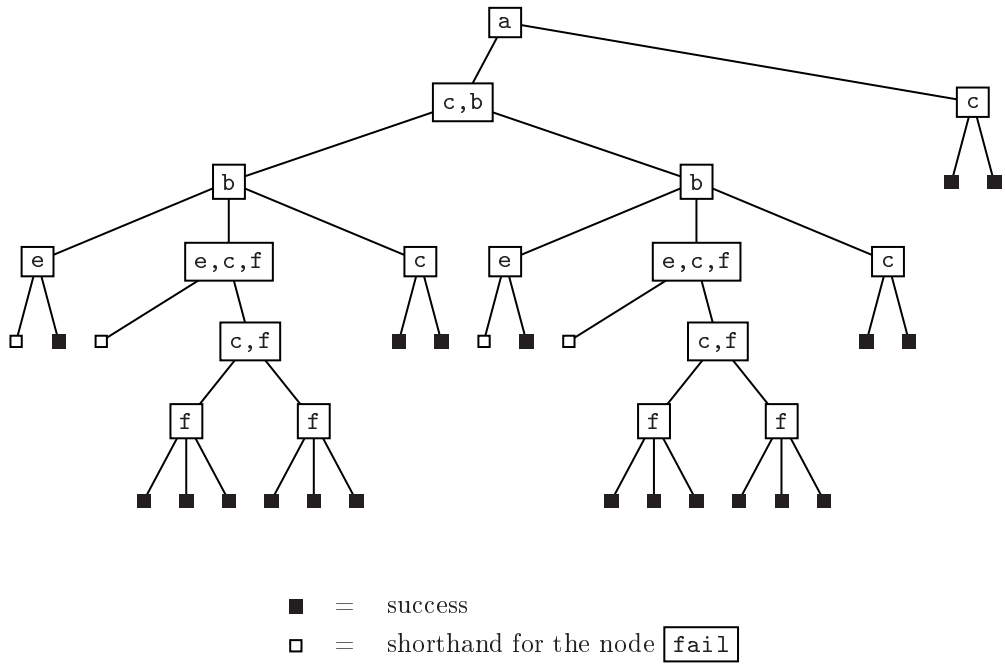

Figure 5.1.1: The search tree considered by PROLOG when asked to resolve the goal `a` using the above script.

predicate `!/0`, called the *cut* predicate. The cut predicate always succeeds, but has the side effect of committing all choices made to the left of it in a rule. That is, how predicates on the right hand side of the rule that come before the cut were satisfied and which rule was chosen for the predicate on the left hand side. Consider for example the rule `b :- e,c,!,f`. The cut will commit the choice of rule for `c`, `e` and `b`.

Thus, the effect of a cut on the backtracking process is that if a cut is attempted resatisfied, the search process continues at the node two levels above the highest node containing this cut predicate in its goal list. From this node, the next non-visited child is searched. Said another way, when a cut is unsatisfied, all choices that the cut has committed are also unsatisfied.

To illustrate this consider the above PROLOG script with an added cut.

```
1  a :- c,b.
2  a :- c.
3
4  b :- e.
5  b :- e,c,!,f.
6  b :- c.
7
8  c.
9  c.
10
11 e :- fail.
12 e.
13
```

```
14  f.
15  f.
16  f.
```

The search tree for the goal `a` will then be as shown in Figure 5.2.1. Bold arrows denote jumps caused by the added cut, and subtrees no longer visited are shown on a gray background.

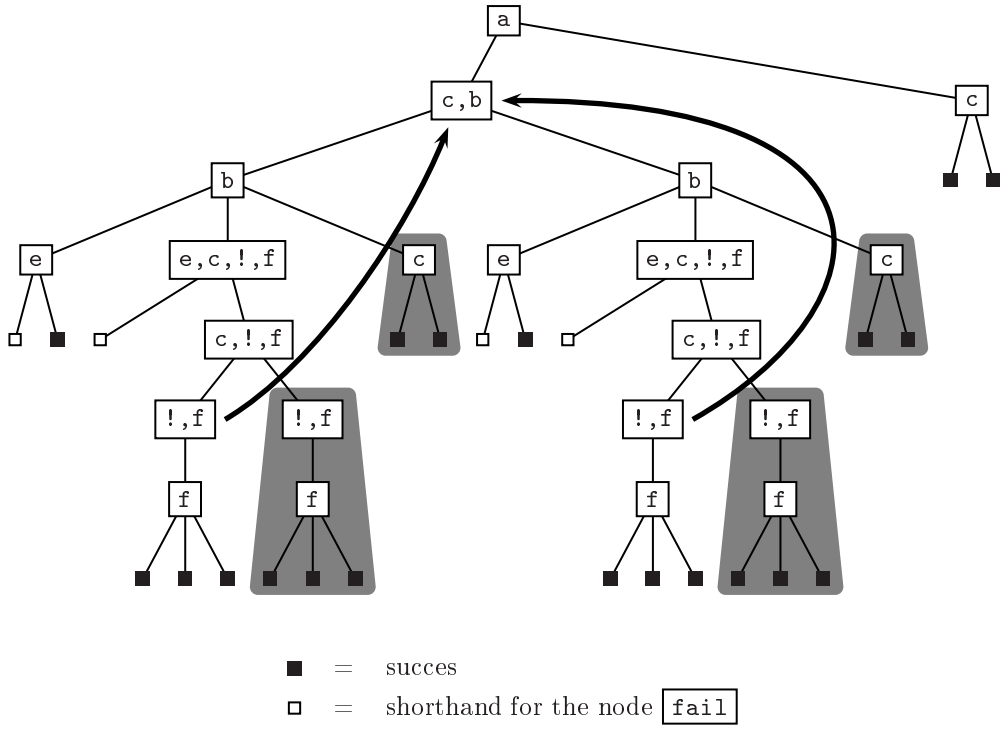

Figure 5.2.1: The search tree considered by PROLOG when asked to resolve the goal `a` using the above script (with the added cut). Bold arrows denote jumps caused by the added cut, and subtrees no longer visited are shown on a gray background.

5.3. Common Uses of Cuts[1]

There are three common uses of cuts:

- Confirming the choice of a rule,
- Forcing failure (using the so-called "cut-fail" combination), and
- Terminating "generate and test" code.

We will examine these uses in the following sections.

[1] This section adapted from [CM94].

5.3.1. Confirming the Choice of a Rule

Imagine that we want to write a PROLOG predicate `sum_to(N,R)` that instantiates `R` to the sum of $1, \ldots, N$. Initially, we may attempt the following definition of `sum_to`.

```
sum_toWrong(N,1) :- N=<1.
sum_toWrong(N,R) :-
        N1 is N-1,
        sum_toWrong(N1,R1),
        R is R1 + N.
```

But as the following output shows us, this definition of the predicate does not provide the correct answer (or rather, it provides wrong answers as well!).

```
| ?- sum_toWrong(2,R).

R = 3 ? ;

R = 4 ? ;

R = 4 ? ;

R = 3 ? ;

R = 1 ? ;

R = -2 ? ;

R = -6 ? ;

R = -11 ?

etc.
```

The reason for this is that, when asked for more solutions PROLOG initiates backtracking. This causes the system to unsatisfy the last satisfied goal and attempt to resatisfy it in a different way. It is easy to see that the last satisfied goal must been satisfied by the base case (`sum_toWrong(N,1)`).

However, once the base case has been applied to satisfy a goal, we do not wish for this goal to be satisfied in a different way. We can achieve this by inserting a cut as shown in the following definition of `sum_to`.

```
sum_toGood(N,1) :- N=<1,!.
sum_toGood(N,R) :-
        N1 is N-1,
        sum_toGood(N1,R1),
        R is R1 + N.
```

As shown in the following test run, this definition produces only one (correct) result.

```
| ?- sum_toGood(2,R).

R = 3

yes
```

5.3.2. The "Cut-Fail" Combination

There may be situations in which we wish to force the PROLOG system to fail when attempting satisfaction of a goal. For example, we may know that if a certain condition is true, a goal should not be satisfiable.

Consider, for instance, the following PROLOG script designed to identify average tax payers.

```
1  income(peter,400000).
2  foreigner(peter).
3
4  average_taxpayerWrong(X) :- foreigner(X), fail.
5  average_taxpayerWrong(X) :- income(X,I), I < 500000.
```

Here the built in predicate `fail/0` is used to force a failure. It is clear that the goal `average_taxpayerWrong(peter)` should fail, since Peter is a foreigner. However, as the following output shows, this is not the case.

```
1  | ?- average_taxpayerWrong(peter).
2
3  yes
```

This is because the `fail/0` predicate initiates backtracking, which causes the PROLOG system to attempt satisfaction of the goal using another rule. And since the goal is satisfiable by another rule (the next one), the system reports that it has satisfied the goal.

Again, this can be remedied by inserting a cut as follows.

```
1  average_taxpayer(X) :- foreigner(X), !, fail.
2  average_taxpayer(X) :- income(X,I), I < 500000.
```

The output shown below shows that this new definition gives the correct answer.

```
1  | ?- average_taxpayer(peter).
2
3  no
```

In this case, it would be more natural to use the built in "not satisfiable" operator `\+`. The `\+` operator is defined such that `\+X` is satisfiable if only if `X` is not satisfiable. This allow for the following definition of the `average_taxpayer` predicate, i.e. an average tax payer is not a foreigner and has an income less than 500000.

```
1  average_taxpayer(X) :- \+foreigner(X), income(X,I), I < 500000.
```

5.3.3. Terminating "Generate and Test" Code

A typical type of programing in PROLOG is so-called "generate and test" code. This type of code involves generating a solution and testing if it is correct. Consider, for example, the following script to perform integer division.

Here, we use the `is_integer/1` predicate to generate a result and the `div_test/3` predicate to test if it is correct.

```
1  /* Divide N1 by N2 */
2  divide(N1, N2, Result) :-
```

```
3           is_integer(Result),
4           div_test(N1, N2, Result),
5           !.
6
7   /* Generate all integers */
8   is_integer(1).
9   is_integer(X) :- is_integer(Y), X is Y+1.
10
11  /* Test for N1/N2 = D */
12  div_test(N1,N2,D) :-
13        P1 is N2*D,
14        P2 is N2*(D+1),
15        P1 =< N1,
16        P2 > N1.
```

We have placed a cut at the end of the `divide` predicate, because we know that there is only one possible choice of `Result` that will satisfy the predicate, and therefore do not wish to consider further values of `Result`. In fact, were we to "allow" backtracking by removing the cut, the script would simply run forever if we did not accept the initial (only) result.

References

[CM94] Clocksin and Mellish: *Programming in Prolog*, 5^{th} ed., Springer, 2003.