

Side Effects

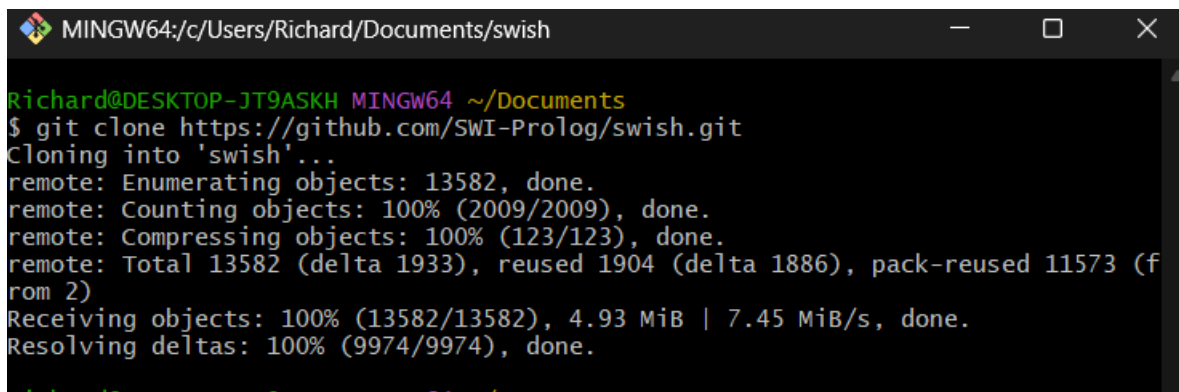
0 Setup

First, you must go to the following link to download and install SWI-Prolog:

<https://www.swi-prolog.org/download/stable>

Open Git Bash in the desired location and enter the following command:

```
git clone https://github.com/SWI-Prolog/swish.git
```

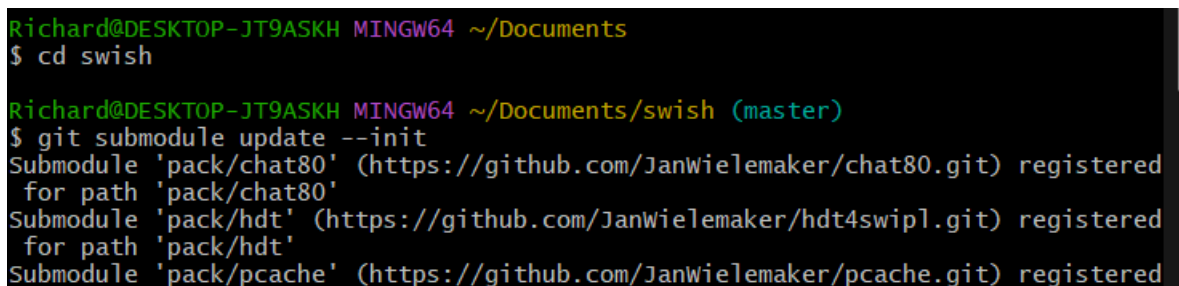


```
MINGW64:/c/Users/Richard/Documents/swish
Richard@DESKTOP-JT9ASKH MINGW64 ~/Documents
$ git clone https://github.com/SWI-Prolog/swish.git
Cloning into 'swish'...
remote: Enumerating objects: 13582, done.
remote: Counting objects: 100% (2009/2009), done.
remote: Compressing objects: 100% (123/123), done.
remote: Total 13582 (delta 1933), reused 1904 (delta 1886), pack-reused 11573 (from 2)
Receiving objects: 100% (13582/13582), 4.93 MiB | 7.45 MiB/s, done.
Resolving deltas: 100% (9974/9974), done.
```

Go to your swish root directory, and then run:

```
cd swish
```

```
git submodule update --init
```








```
Richard@DESKTOP-JT9ASKH MINGW64 ~/Documents
$ cd swish
Richard@DESKTOP-JT9ASKH MINGW64 ~/Documents/swish (master)
$ git submodule update --init
Submodule 'pack/chat80' (https://github.com/JanWielemaker/chat80.git) registered for path 'pack/chat80'
Submodule 'pack/hdt' (https://github.com/JanWielemaker/hdt4swipl.git) registered for path 'pack/hdt'
Submodule 'pack/pcache' (https://github.com/JanWielemaker/pcache.git) registered
```

You must go to the following link and download 'swish-node-modules.zip':

<https://www.swi-prolog.org/download/swish/>

Index of /download/swish

Name	Last modified	Size
 Up	-	-
 README.txt	Thu Nov 6 13:18:18 2014	294
 swish-bower-components.zip	Fri Dec 14 08:05:51 2018	4,185,289
 swish-js-components.zip	Thu Apr 25 12:16:54 2019	4,185,289
 swish-node-modules.zip	Mon Dec 16 16:07:59 2019	10,759,756

Copy this archive to the swish folder and extract it.

The following commands must be entered:

```
mkdir -p config-enabled
```

```
cd config-enabled && ln -s ../config-available/auth_http_always.pl
```

Go to environment variables and add the following to PATH:

```
C:\Program Files\swipl\bin\
```

Return to the main swish folder.

```
cd ..
```

You can now run swish prolog locally with the following command (you must be in the swish folder):

```
swipl run.pl
```

```
Richard@DESKTOP-JT9ASKH MINGW64 ~/Documents/swish (master)
$ swipl run.pl
% Created data directory "c:/users/richard/documents/swish/data"
% Updating GIT version stamps in the background.
% Started server at http://localhost:3050/
1 ?- |
```

Run the following command and create your account:

```
swish_add_user.
```

Add all your credentials and press enter.

```
MINGW64:/c/Users/Richard/Documents/swish
Richard@DESKTOP-JT9ASKH MINGW64 ~/Documents/swish (master)
$ swipl run.pl
% Updating GIT version stamps in the background.
% Started server at http://localhost:3050/
1 ?- swish_add_user.
% Password file: c:/users/richard/documents/swish/passwd (create)
User name: Richard
Real name: Richard
Group:      test
E-Mail:     test@gmail.com
Password:
(again):
true.
2 ?- |
```

You can access your local SWISH instance at:

<http://localhost:3050/>

1 Theoretical Considerations

In this lesson, we will learn about side effects which allow us to dynamically manipulate the predicate base.

1.1 Side Effects

Side effects refer to a series of Prolog predicates which allow the dynamic manipulation of the predicate base:

- *assert/1* (= *assertz/1*) → adds the predicate clause given as argument as last clause
- *asserta/1* → adds the predicate clause given as argument as first clause
- *retract/1* → tries to unify the argument with the first unifying fact or clause in the database. The matching fact or clause is then removed from the database
- *retractall/1* → deletes all clauses that unify with the argument (in SWI it will always succeed, even if nothing is deleted)

Predicates defined and/or called in the „.pl” file are called static predicates. Predicates which can be manipulated via *assert/retract* statements at run-time are called dynamic predicates. As opposed to the static predicates that we have seen so far, dynamic predicates should be declared as such.

In Prolog, a predicate is either static or dynamic. A static predicate is one whose facts/rules are predefined at the start of execution, and do not change during execution. Normally, the facts/rules will be in a file of Prolog code which will be loaded during the Prolog session. Sometimes, you might want to add extra facts (or maybe even extra rules) to the predicate, during execution of a Prolog query, using "*assert/asserta/assertz*", or maybe remove facts/rules using "*retract/retractall*". To do this, the predicate must be declared as dynamic.

However, when the interpreter sees an *assert* statement on a new predicate, it implicitly declares it as dynamic. If we want to manipulate a predicate that appears in the „.pl”, then we need to set it as a dynamic predicate by adding the following line to the start of the file:

```
:-dynamic <predicate_name>/<arity>.
```

The important aspects you need to know when working with side effects:

- Their **effect is maintained in the presence of backtracking**, i.e. once a clause has been asserted, it remains on the predicate base until it is explicitly retracted, even if the node corresponding to the *assert* call is deleted from the execution tree (e.g. because of backtracking).
- *assert* - always succeeds; doesn't backtrack.
- ***retract* - may fail and backtrack**, which invalidates temporarily the deletion for predicates within the same body of the clause with the retract call that were called before retract; retract respects the *logical update view* - it succeeds for all clauses that match the argument when the predicate was *called*.

To better understand how it works, try to execute the following query:

Query

```
?-    assert(insect(ant)),  
    assert(insect(bee)),  
    retract(insect(A)),  
    writeln(A),  
    retract(insect(B)),  
    fail.
```

You have probably observed that this query will output:

Answer

```
ant  
bee  
false
```

This is because even if the second call to retract will also delete the fact *insect(bee)*, when backtracking reaches the first retract call, the clause is still present in its logical view - it doesn't see that the clause has been deleted by the second retract call. As such, *insect(A)* can still unify with „bee”.

Prolog also has a ***retractall/1*** predicate, with the following behavior: it deletes all predicate clauses matching the argument. In some versions of Prolog, *retractall* may fail if there is nothing to retract. To get around this, you may choose to assert a dummy clause of the right type. In SWI Prolog, however, *retractall* succeeds even for a call with no matching facts/rules.

Dynamic database manipulation via *assert/retract* can be used for storing computation results (memorisation/caching), such that they are not destroyed by backtracking. Therefore, if the same question is asked in the future, the answer is retrieved without having to recompute it. This technique is called memorisation, or caching, and in some applications it can greatly increase efficiency. However, side effects can also be used to change the behaviour of

predicates at run-time (meta-programming). This generally leads to dirty, difficult to understand code. In the presence of heavy backtracking, it gets even worse. Therefore, this non-declarative feature of Prolog should be used with caution.

1.2 Fibonacci

1.2.1 Memorization

An example of memorisation of partial results, using side effects, is the following predicate which computes the *nth* number in the *fibonacci sequence* (you have already seen the less efficient version in the second lab session):

Code

```
:-dynamic memo_fib/2.

fib(N,F):- memo_fib(N,F), !.
fib(N,F):-
    N>1,
    N1 is N-1,
    N2 is N-2,
    fib(N1,F1),
    fib(N2,F2),
    F is F1+F2,
    assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Follow the execution of (run the queries sequentially):

Query

```
?- listing(memo_fib/2).
% lists all definitions of the predicate memo_fib with 2 arguments

?- fib(4,F), listing(memo_fib/2).
?- fib(10,F), listing(memo_fib/2).
```

1.2.2 Printing memorised results

Whenever you want to collect all the answers that you have stored on your predicate base via assert statements, you can use **failure driven loops** which *force Prolog to backtrack* until there are no more possibilities left. The pattern for a failure driven loop which reads all stored clauses - say for predicate *memo_fib/2* above - and prints all the *fibonacci numbers* already computed:

Code

```
print_all:-
```

```
memo_fib(N,F),  
write(N),  
write(' - '),  
write(F),  
nl,  
fail.  
print_all.
```

We will make use of the backtracking technique (by forcing *fail*) to traverse all clauses of the *memo_fib/2* predicate added to the knowledge base through *assert*. **Due to the fact that backtracking is used, the *memo_fib/2* facts will not repeat.**

Follow the execution of:

```
Query  
?-print_all.  
?-retractall(memo_fib(_,_)).  
?-print_all.
```

1.2.3 Collecting memorised results

To collect results in a list, we can use the built-in predicate: *findall*.

Follow the execution of:

```
Query  
?- findall(X, append(X,_,[1,2,3,4]), List).  
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).  
?- findall(X, member(X,[1,2,3]), List).
```

Let's now see an example of using *side effects* to get all the possible answers to a query: let's write a predicate which computes all the permutations of a list, and returns them in a separate list. We want the query on the predicate to behave like this:

```
Query  
?- trace, all_perm([1,2,3],L).  
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];  
no.
```

Assuming that you already know how to implement the *perm/2* predicate - the predicate which generates a permutation of the input list (see the document on Sorting Methods if not) - the *all_perm/2* predicate specification is:

```
Code  
all_perm(L,_):-  
    perm(L,L1),  
    assertz(p(L1)),
```

```

fail.
all_perm(_,R):- collect_perms(R).

collect_perms([L1|R]):- retract(p(L1)),!, collect_perms(R).
collect_perms([]).

perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).

```

Follow the execution of:

```

Query
?- retractall(p(_)), all_perm([1,2],R), listing(p/1).
?- retractall(p(_)), all_perm([1,2,3],R), listing(p/1).

```

Questions:

1. Why do I need a **retractall** call before calling **all_perm/2**?
2. Why do I need a **!** after the **retract** call in the first clause of **collect_perms/1**?
3. What kind of recursion is used on **collect_perms/1**? Can you do the collect using the other type of recursion? Which is the order of the permutations in that case?
4. Does **collect_perms/1** destroy the results stored on the predicate base, or does it only read them?

1.2.4 Two-stage / Two-step process of failure driven loop collection

Collecting results from failure driven loops, such as the *all_perm/2* predicate, can be seen as a two stage processes. They include a wrapper, *all_preds/2* in our example, that has 2 clauses. The first part is the processing and storing of results (through *asserts*) which is done by the first clause. This first clause must fail, such that the predicate reaches the second clause, where the collection of results beings (through *retracts*).

```

Code
all_preds(L,_):- store(L).
all_preds(_,R):- collect(R).

store(L):-
    get(L, L1),
    assertz(p(L1)),
    fail.

collect([L1|R]):- retract(p(L1)),!, collect(R).
collect([]).

```

1.3 Failure driven loop vs recursion

We would like to highlight a comparison between failure driven loops and recursive predicates and which is better in different scenarios. The main issue with recursion when reading the memory base is that each recursive call will read the same fact, resulting in an infinite loop. However, due to the inherent backtracking of fail, in failure driven loops you can easily read the memory base.

One way to implement such a loop with recursion that reads the memory base is through the use of *retract* (the same approach can be made with failure driven loops, however it is not necessary). This guarantees that no infinite loop appears, as the content is removed. This is a **disadvantage, as it destroy the input**. Thus, to keep the input when using recursion, a trick must be used. We must create an additional predicate that keeps track of what was already accessed (in this case, the *seen/1* predicate). However, this approach is inefficient, at each recursion step all predicates that have been accessed are accessed again until an unaccessed one is found.

Code	
<pre>:-dynamic p/1. p(1). p(2). p(3). p(4). p(5).</pre>	
<pre>% does not require the retract failure_driven_loop1:- p(X), assert(q(X)), fail. failure_driven_loop1:-listing(q/1).</pre>	<pre>% but can be implemented with it failure_driven_loop2:- retract(p(X)), assert(q(X)), fail. failure_driven_loop2:-listing(q/1).</pre>
<pre>% must make the retract, % otherwise infinite loop recursion1:- retract(p(X)), assert(q(X)), recursion1. recursion1:-listing(q/1).</pre>	<pre>% to keep p/1 in knowledge base, % requires additional predicate, % highly inefficient recursion2:- p(X), not(seen(X)),!, assert(seen(X)), assert(q(X)), recursion2. recursion2:-listing(q/1). % might be nonsensic here, simple case, yet when q/1 is a process, it is needed</pre>

Follow the execution of:

Query

```
?- failure_driven_loop1.  
?- failure_driven_loop2.  
?- recursion1.  
?- recursion2.
```

1.4 Univ predicate

Also known as the infix predicate:

$X = ..L$

It means that list L contains as first element the functor of X, followed by arguments of X and it has 2 functionalities:

- **if** var(X) **then**
 - L is used to construct the appropriate structure of X; the head of L must be an atom, and becomes the functor of X
- **if** nonvar(X) **then**
 - functor of X matches the first element (head) of L, while the arguments of X matches with the tail of the list (left to right arguments of X, left to right in tail of L)

Query

```
?-X=..[a,b,c,d].  
Yes, X=a(b,c,d)  
  
?-X=..[member,a,[b,c]].  
Yes, X=member(a,[b,c]).  
  
?-f(a,b,c)=..X.  
Yes, X=[f,a,b,c].  
  
?-append([H|T],L,[H|R])=..X.  
Yes, X=[append,[H|T],L,[H|R]].
```

Through the univ/infix predicate, we can create more general functions that allow rules to be applied after our own choice. For example, we want to create a predicate that allows us to apply a function (in Prolog, a rule) to all of its elements, called *map/3*.

Code

```
map(_, [], []).
map(Pred, [H|T], [H1|R]) :-
    P=..[Pred, H, H1],    % create a predicate through univ → Pred(H
    call(P), !,           % call the created predicate
    map(Pred, T, R).

double(X, Y) :- Y is X * 2.
halve(X, Y) :- Y is X / 2.
% ... add any function that you want
```

Follow the execution of:

Query

?- trace, map(double, [1, 2, 3], Result).

Result = [2, 4, 6].

?- trace, map(halve, [2, 4, 6], Result).

Result = [1, 2, 3].

2 Exercises

Before beginning the exercises, add the following facts which define trees (complete and incomplete binary) into the Prolog script:

Code

% Trees:

```
incomplete_tree(t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))).
```

```
complete_tree(t(7, t(5, t(3, nil, nil), t(6, nil, nil)), t(11, nil, nil))).
```

Write a predicate which:

1. Generates a list with all the possible decompositions of a list into 2 lists, without using the built-in predicate *findall*.

Query

```
?- all_decompositions([1,2,3], List).
```

```
List = [ [], [1,2,3] ], [ [1], [2,3] ], [ [1,2], [3] ], [ [1,2,3], [] ] ;
```

```
false
```

2. Filters out elements based on a given function (*suggestion*: univ predicate).

Query

```
?- filter(even, [1, 2, 3, 4], Result).
```

```
Result = [2, 4].
```

3. Returns true if any elements satisfies a given function (*suggestion*: univ predicate).

Query

```
?- any(greater_than_three, [1, 2, 4]).
```

```
true.
```

4. Returns true if all elements satisfy a given function (*suggestion*: univ predicate).

Query

```
?- all(positive, [1, 2, 3]).
```

```
true.
```

5. Collects (using retracts through a 2-step process) in a **difference list** all even elements of a given **incomplete** list.

Query

```
?- even_dl([1,2,3|_], S, E).
```

```
S = [2|E]
```

6. Collects (using retracts through a 2-step process) all internal nodes of an **incomplete** binary tree.

Query

?- incomplete_tree(T), internal_list(T, R).

R = [7, 5|_].