

Chapter 5

Simple Interpreters

5.1 NFA

Our first task is to write an interpreter for non-deterministic finite automaton (NFA; see, e.g., [1]).

One question we must ask ourselves is, how do we represent the NFA to be interpreted. One solution is to represent the NFA by Prolog facts. E.g., we could use Prolog predicate `init/1` to represent the initial states, `final/1` to represent the final states, and `trans/3` to represent the transitions between the states. To represent the NFA from Figure 12.2 we would thus write:

Program 5.1.1

```
init(1).  
trans(1,a,2).  
trans(2,b,3).  
trans(2,b,2).  
final(3).
```

Let us now write an interpreter, checking whether a string can be generated by the automaton. The interpreter needs to know the current state `St` as well as the string to be checked. The empty string can only be generated in final states.

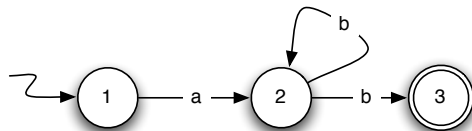


Figure 5.1: A simple NFA

A non-empty string can be generated by taking an outgoing transition from the current state to another state $S2$, thereby generating the first character H of the string; from $S2$ we must be able to generate the rest T of the string.

```
accept(S, []) :- final(S).
accept(S, [H|T]) :- trans(S,H,S2), accept(S2,T).
```

We have not yet encoded that initially we must start from a valid initial state. This can be encoded by the following:

```
check(Trace) :- init(S), accept(S,Trace).
```

We can now use our interpreter to check whether the NFA can generate various strings:

```
| ?- check([a,b]).
yes

| ?- check([a]).
no
```

We can even only provide a partially instantiated string and ask for solutions:

```
| ?- check([X,Y,Z]).
X = a,
Y = b,
Z = b ? ;
no
```

Finally, we can generate strings accepted by the NFA:

```
| ?- check(X).
X = [a,b] ? ;
X = [a,b,b] ? ;
X = [a,b,b,b] ? ;
```

Exercise 5.1.2 Extend the above interpreter for epsilon transitions.

Are there problems with loops on epsilon transitions? If so, what are the solutions?

5.2 Regular Expressions

Our next task is to write an interpreter for regular expressions (see, e.g., [1]). We will first use operator declarations so that we can use the standard regular expression operators “.” for concatenation, “+” for alternation, and “*” for repetition. For this we write the following operator declarations (see Section 4.1).

```
:- op(450,xfy,'.'). /* + already defined; has 500 as priority */
:- op(400,xf,'*').
```

Let us first develop an interpreter using `append/3` :

Program 5.2.1

```
:- use_module(library(lists), [append/3]).
gen(X, [X]) :- atomic(X).
gen(X+Y,S) :- gen(X,S) ; gen(Y,S).
gen(X.Y,S) :- gen(X,SX), append(SX,SY,S), gen(Y,SY).
gen('*'(_X), []).
gen('*'(X),S) :- gen(X,SX), append(SX,SSX,S), gen('*'(X),SSX).
```

Observe how to compute the meaning/effect of `X.Y` we compute the effect of `X` and `Y` by recursive calls to `gen`. This is a common pattern, that will appear time and time again in interpreters. We call this “*compositional style*” (or *denotational style*).

The above interpreter can be used as follows:

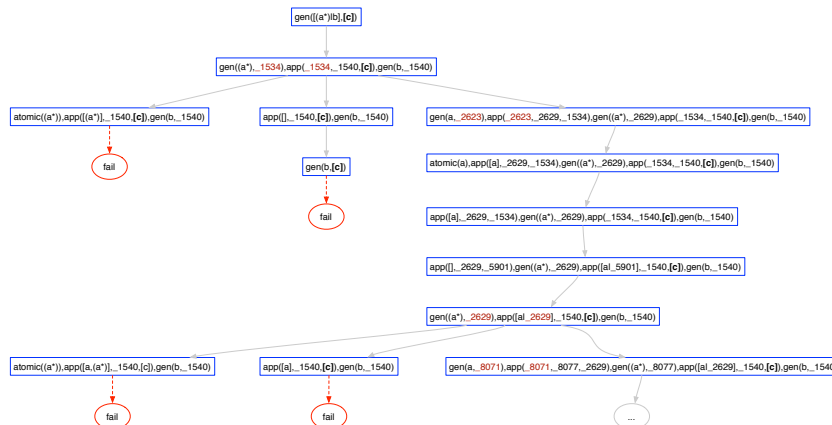
```
| ?- gen(a*, [X,Y,Z]).
X = a,
Y = a,
Z = a ?
yes

| ?- gen((a+b)*, [X,Y]).
X = a,
Y = a ? ;
X = a,
Y = b ? ;
X = b,
Y = a ? ;
X = b,
Y = b ? ;
no
```

The above interpreter is unfortunately not very efficient, as the argument `SX` in the clauses for concatenation and repetition are traversed a second time by the call to `append`. There is, however, a more serious problem with the interpreter, illustrated by the following call:

```
| ?- gen((a*).b, [c]).
! Resource error: insufficient memory
```

This is very similar to the problem we have encountered while parsing in Section 4.3. The problem is that we call `gen(X,SX)` with `SX` unbound (see Figure 5.2, e.g., the call `gen((a*),_1534)`). The solution is the same as in

Figure 5.2: Call graph for first version for the call `gen((a*).b, [c])`

Section 4.3, namely the use of difference lists. These avoid having to traverse lists multiple times and also avoid calling `gen(X, SX)` with `SX` being an unbound variable.

Program 5.2.2 Regular expression interpreter with difference lists:

```
generate(X, [X|T], T) :- atomic(X).
generate(X +_Y, H, T) :- generate(X, H, T).
generate(_X + Y, H, T) :- generate(Y, H, T).
generate(X.Y, H, T) :- generate(X, H, T1), generate(Y, T1, T).
generate('*'(_), T, T).
generate('*'(X), H, T) :- generate(X, H, T1), generate('*'(X), T1, T).

gen(RE, S) :- generate(RE, S, []).
```

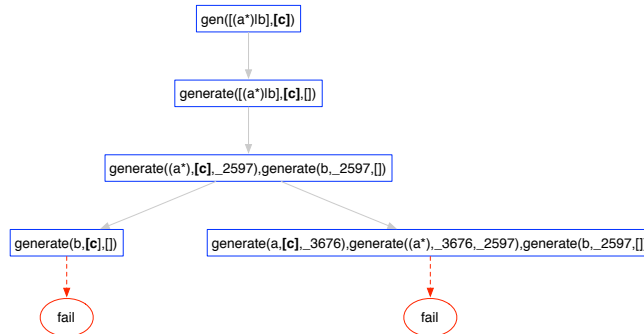
We can now safely use the above call (see Figure 5.3), resulting in the correct reply that the string `[c]` cannot be generated by the regular expression:

```
| ?- gen((a*).b, [c]).
no
```

As already mentioned in Section 4.3, there is an alternate, maybe more intuitive reading of the difference lists. In particular, for a call `generate(RE, InEnv, OutEnv)` we can view the last two arguments as environments, where

- `InEnv` are the characters still to be consumed overall,
- `OutEnv` are the characters remaining after `RE` has been matched.

Our program can equally well be written in DCG style notation. Observe, how the DCG notation helps us to automatically *thread* the environment through the various literals in the clauses.

Figure 5.3: Call graph for second version for the call `gen((a*).b,[c])`**Program 5.2.3**

```

generate(X) --> [X], {atomic(X)}.
generate(X+_Y) --> generate(X).
generate(_X+Y) --> generate(Y).
generate('.'(X,Y)) --> generate(X), generate(Y).
generate('*'(_)) --> [].
generate('*'(X)) --> generate(X), generate('*'(X)).

```

5.2.1 Representation: Prolog terms vs facts

In the NFA program we have represented the automaton by a set of Prolog facts. In the Regular Expression program we have encoded the regular expression to be interpreted as a Prolog term and passed it as an argument.

Which approach is better?

There is no clear cut answer; it all depends on the particular circumstances. Advantages of the first approach with Prolog facts are:

- we can use Prolog queries to search the fact database, e.g., `trans(Pre,_,2)` to find predecessors of 2 or `trans(X,_,X)` to find states with self-loops.
- Prolog provides automatic first argument indexing; e.g., looking for the successors of known state `X` using `trans(X,A,Y)` will be very efficient.
- the Prolog facts do not have to be passed as explicit arguments to the interpreter.

The disadvantages of this approach are:

- the interpreter cannot easily modify the program to be interpreted ... modifications via `assert/retract` can be expensive and have to be handled with extreme care inside an interpreter;

Also, when asserting a term, the Prolog system basically makes a copy of the term being asserted: no sharing with other terms is maintained.

e.g., in the regular expression example: sub-expressions $X + Y$ passed to recursive calls

- modifications via `assert/retract` are expensive

Exercise 5.2.4 Write the NFA interpreter by passing the NFA as an argument. Try to measure the performance (using `statistics(runtime, [T1,T2])`).

5.3 What we have learned

- compositional style of writing interpreters
- threading environments very similar to DCG-style
- two different ways to represent programs to be interpreted: as Prolog facts or as argument terms

Chapter 6

Interpreters for Propositional Logic: Implementing Negation

Let us try to write an interpreter for simple propositional logic formulas, using the constants *true* and *false*, as well as the two connectives *or* and *and*.

Program 6.0.1

```
int(const(true)).
int(const(false)) :- fail.
int(and(X,Y)) :- int(X), int(Y).
int(or(X,Y)) :- int(X) ; int(Y).
```

Note that for the logical connectives, we again use the “decompositional” style: to compute the effect or meaning of a formula, we first compute the effect or meaning of the direct subformulas and then derive the overall effect or meaning from these. The interpreter can be used as follows.

```
| ?- int(and(const(true),const(true))).
yes
| ?- int(and(const(true),const(false))).
no
```

One may wonder why we have wrapped true and false into `const/1`. The reason is to separate constants from the other formulas and to be able to ask queries of the following kind:

```
| ?- int(and(const(Y),const(X))).
X = true,
Y = true ? ;
no
```

Without the `const/1` wrapper we would have had to ask the query `?-int(X,Y)` which has infinitely many solutions.

Exercise 6.0.2 Write the above interpreter without the `const/1` functor and explain why you now get infinitely many solutions.

Let us now try to add negation to our interpreter. The obvious solution is to directly employ Prolog negation and to add the following clause to the above program:

Program 6.0.3

```
int(not(X)) :- \+ int(X).
```

At first sight this seems to work:

```
| ?- int(not(const(true))).
no
| ?- int(not(const(false))).
yes
```

However, take a look at the following two queries:

```
| ?- int(not(const(X))).
no
| ?- int(const(X)).
X = true ? ;
no
```

What is going on here? Why does Prolog fail to find the solution `X = false` for the first query, while it is able to find the solution `X=true` for the second one?

The reason is that Prolog's built-in negation is not logical negation but so-called "negation-as-failure". This negation can be given a logical description only when its arguments contains no variables at the moment it is called. To understand this issue let us examine a simpler program:

Program 6.0.4

```
int(0).
int(s(X)) :- int(X).
```

Program 6.0.5

```
?- \+ int(a). /* succeeds */
?- \+ int(X), X=a. /* fails */
?- X=a, \+ int(X). /* succeeds */
```

As you can see in the last two queries, conjunction is not commutative here. The explanation is that the Prolog negation is not declarative, i.e., it cannot be described within logic (where conjunction is commutative).

Logically the program is $int(0) \wedge \forall X.int(s(X)) \leftarrow int(X)$.

Prolog (and logic programming) uses the negation as failure rule: if we cannot prove p then assume that $\neg p$ is true. Note that in classical logic one can have that neither p nor $\neg p$ is a logical consequence. There is, however, a way to view (in some circumstances) Prolog negation as classical negation on a transformed program. This concept is called **Clark completion**.

The idea of Clark completion is to view a predicate definition not as a series of implications but as a big “if-and-only-if” formula. This is achieved by first introducing an equality predicate (say $=/2$ as in Prolog) and translating all clauses so that the head contains only variables (the same for all clauses of the same predicate).

Take the two clauses defining `int` above. Our first clause can be translated into:

- $\forall Y.int(Y) \leftarrow Y = 0$

Second clause can be transformed as follows:

- $\forall X.int(s(X)) \leftarrow int(X)$
- $\forall X.int(Y) \leftarrow Y = s(X), int(X)$
- $\forall Y.int(Y) \leftarrow \exists X.(Y = s(X) \wedge int(X))$

We can now combine both into $IFF(P)$:

- $\forall Y.int(Y) \leftrightarrow (Y = 0 \vee \exists X.(Y = s(X) \wedge int(X)))$.

This formula is combined with axioms defining the equality predicate, called free-equality theory for P , denoted by $FET(P)$:

- $0 = 0$
- $\forall X, Y.s(X) = s(Y) \rightarrow X = Y$ ¹
- $\forall X.\neg(s(X) = 0)$
- $\forall X.\neg(0 = s(X))$

We have that $FET(P) \wedge IFF(P) \models \neg int(a)$. However, Prolog’s implementation of negation is only sound when there are no free variables inside the negated goal !

Hence to use negation (inside an interpreter) there are basically three solutions. The first is to always ensure that there are no variables when we call the negation. This may be difficult to achieve in some circumstances; and generally

¹The other direction, $\forall X, Y.s(X) = s(Y) \leftarrow X = Y$ is implied by the substitution schema.

means we can use the interpreter in only one specific way. But it is still often a practical solution.

The second solution is to delay negated goals until they become ground. This can be achieved using the built-in predicate `when/2` of Prolog. The call `when(Cond, Call)` waits until the condition `Cond` becomes true, at which point `Call` is executed. While `Call` can be any Prolog goal, `Cond` can only use: `nonvar(X)`, `ground(X)`, `?=(X, Y)`, as well as combinations thereof combined with conjunction (`,`) and disjunction (`;`). With this built-in we can implement a safe version of negation, which will ensure that the Prolog negation is only called when no variables are left inside the negated call:

Program 6.0.6

```
safe_not(P) :- when(ground(P), \+(P)).
```

A disadvantage is that this does not work with all Prolog systems (though most now support co-routining). One can also refutations with a floundering goal, where all goals suspend. In that case, one does not know whether the query is a logical consequence of the program or not. The Gödel programming language supported such a safe version of negation.

Another common technique to circumvent this problem is to get rid of the need for the built-in negation, by explicitly writing a predicate for negated formulas:

Program 6.0.7

```
int(const(true)).
int(const(false)) :- fail.
int(and(X,Y)) :- int(X), int(Y).
int(or(X,Y)) :- int(X) ; int(Y).
int(not(X)) :- nint(X).

nint(const(false)).
nint(const(true)) :- fail.
nint(and(X,Y)) :- nint(X) ; nint(Y).
nint(or(X,Y)) :- nint(X), nint(Y).
nint(not(X)) :- int(X).
```

This interpreter now works as expected for negation and partially instantiated queries:

```
| ?- int(not(const(X))).
X = false ? ;
no
```

This interpreter thus actively searches for solutions to the negated formulas.

Exercise 6.0.8 Extend the above interpreter for implication `imp/2` and equivalence `equiv/2`. Use it to print the truth table for an arbitrary formula. Tip: try writing a predicate `truth_table/2` with two arguments, where the first argument is the formula and the second the list of propositional variables in the formula.

Try to extend the interpreter so that it does not give multiple solutions to `T1 = const(X)`, `T2 = or(T1,T1)`, `T3 = and(T2,T2)`, `T4 = and(T3,T3)`, `int(T4)`.

Literature: Clark Completion and Equality Theory, SLDNF, Constructive negation, Gödel Programming language,... SAT solvers

6.1 What have we learned

- wrapping data into functors can be useful, e.g., to avoid infinite backtracking (see, `const/1` functor).
- implementing negation: it is rarely a good idea to use Prolog negation for this. One solution, is the use of co-routines; the other is to write explicit predicates for interpreting the negation of a formula.