Hamoon Mousavi
March 9, 2015

Walnut
User Manual

Installation

You need to have Java installed on your machine. The rest is easy. Unzip the folder named Walnut.zip. Then open the terminal, if you are on OS X or Linux, or command prompt, if you are on Windows. Change the directory to …/Walnut/bin. To run the program write:

>java Main.prover

To exit the program write:

>exit;

Commands

Every command ends in either semicolon(;) or colon (:). If you want to see all the steps of the computation use colon otherwise use semicolon. Whitespace is ignored. You can also span one single command into multiple lines to improve readability. So for example you can write the followings interchangeably:

>eval test “a=b+1”;

>eval test

“a=b+1”;

>eval test “a

=b+1”;

Here is the full list of commands:

eval <name> <predicate>

def <name> <predicate>

reg <name> <number system> <regular expression>

reg <name> <alphabet> <regular expression>

load <file name>

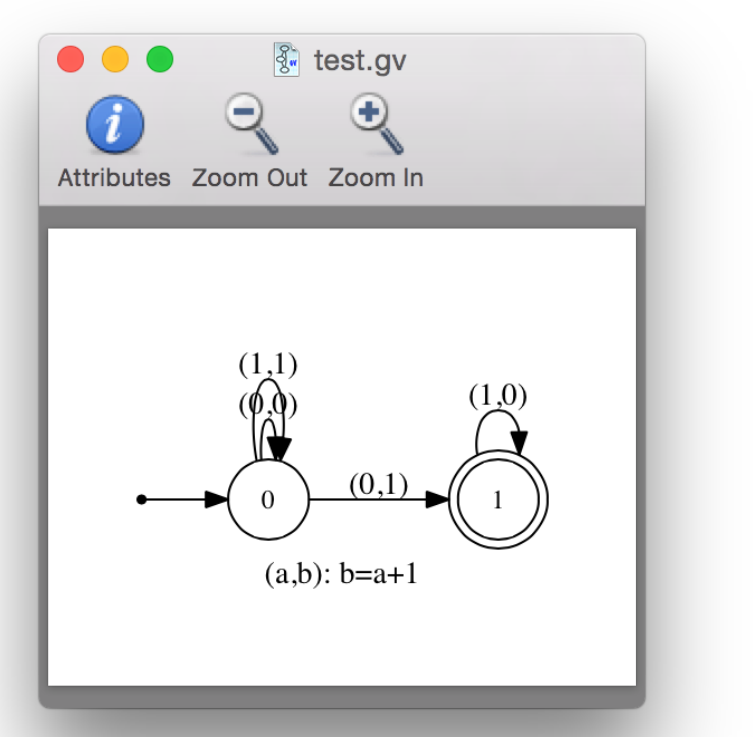
exit

eval command

This is the most important command. This command has two arguments. The first argument is a name for the evaluation. Name of the evaluation starts with a letter and can contain alphanumerics and underscore. The second argument is a predicate that we want to evaluate. Predicates are always place between quotation marks. Let's see an example.

>eval test "b=a+1";

This evaluates to an automaton with 2 binary inputs labeled a and b. The automaton accepts only if b = a+1. This automaton will be drawn and saved in the "Result" directory under the file named test.gv:

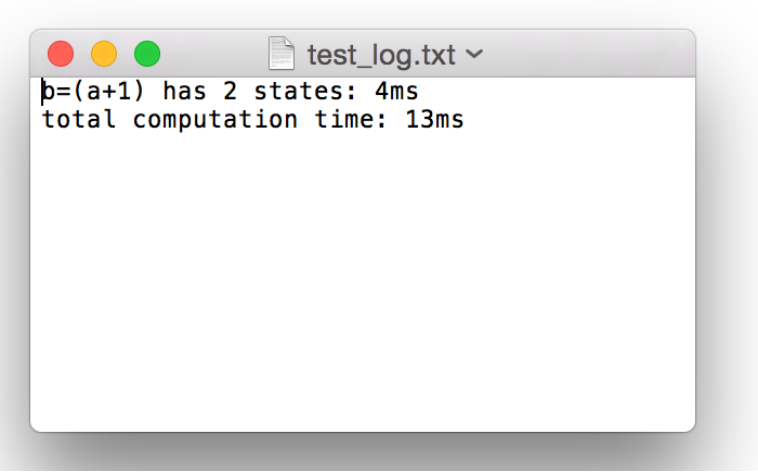


As can be seen, inputs are ordered lexicographically, i.e., a is the first input and b is the second input. To see this graph drawing of the automaton, you need to have the graph visualization software called Graphviz installed on your system. Graphviz is free and can be installed on all platforms.

Walnut generates two other files as the outcomes of this evaluation: test.txt and test_log.txt. They both can be found in the "Result" directory. The first file is test.txt which is the description of the automaton depicted in test.gv:

The first line of test.txt indicates that the first and second inputs are both in most significant digits first and that they are in base 2, i.e., binary. The second line has two numbers. The first number is the state label and the second number is the state output. In this case, state 0 has output 0 which means it is non accepting. Lines 3-5 are the transitions from state 0. So for example, on state 0 and input (0,0) the automaton remains in state 0. The automaton transits to state 1, if it sees (0,1). Line 6 is another state declaration. It says that state 1 is accepting. States with zero as their output are non-accepting, whereas any other number as state output indicates an accepting state.

The file test_log.txt contains the details of the computation including the intermediate steps and the time each of those steps took to complete. In our example, there are not many intermediate steps involved:



Let's see a more interesting example:

>eval squares_in_thue_morse_word "Ak k < n => T[i+k]=T[i+n+k]"

The result of this evaluation is an automaton with two inputs (i,n). The automaton accepts (i,n) if there is a square starting at position i of order n in the Thue-Morse word, i.e., T[i..i+n-1]=T[i+n..i+2n-1]. The A in the beginning of the predicate is the for all quantification. You can read the predicate as follows:

"Give me the automaton that accepts (i,n)'s for which for all k < n the symbols T[i+k] and T[i+n+k] are equal."

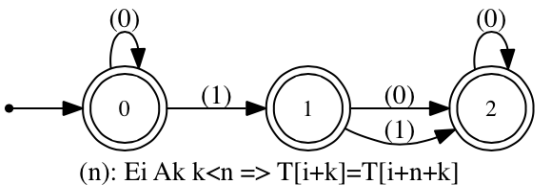
Now if we want to know all length n's for which there exists a square of length n in the Thue-Morse sequence, we simply use the existential quantifier E:

>eval order_of_squares_in_thue_morse_word "Ei Ak k < n => T[i+k]=T[i+n+k]"

So you can read the predicate as follows:

"Give me the automaton that accepts n's for which there exists an index i for which for all k's less that n the symbols T[i+k] and T[i+n+k] are equal."

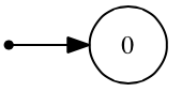
Here is the order_of_squares_in_thue_morse_word.gv:



We know that the Thue-Morse word avoids overlaps. How do we make sure of that with Walnut?

>eval overlaps_in_thue_morse “n > 0 & Ak k <= n => T[i+k]=T[i+n+k]”

The result is



(i,n): n > 0 & Ak k<=n => T[i+k]=T[i+n+k]

Which means that

> “There is no (i,n) for which T[i..i+n] = T[i+n..i+2n].”

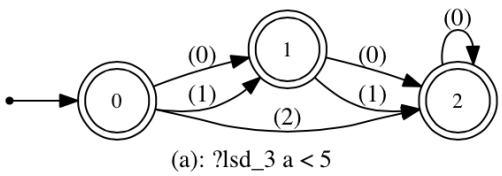
The definition for T can be found in the file T.txt in the directory “Word Automata Library”. You can add your own words to this directory and use them as part of the eval command. The other words that are defined for you includes fibonacci word and paperfolding words.

Is binary the only base we can use with Walnut? The answer is no. You can use base n for all n’s. In addition, you can use Fibonacci and tribonacci bases. The good news is that you can even define your own exotic number systems. Just follow the examples in the directory “Custom Bases”. You can find more information in the section Custom Bases.

How to use bases other than binary?

>eval ternary_example “?lsd_3 a < 5”;

This results in



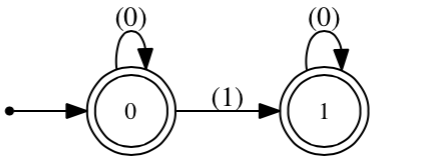
(a): ?lsd_3 a < 5

The automaton accepts exactly those words representing numbers 0,1,2,3, and 4 in the least significant digit ternary base: 0*,10*,20*,10*,110*. Note the trailing zeros in the representations!

Let's see another example, this time on the fibonacci word:

>eval order_of_squares_in_fibonacci_word "?msd_fib Ei Ak k < n => F[i+k]=F[i+n+k]";

This evaluates to an automaton that accepts only n's, in most significant fibonacci, for which there exists a square of order n starting at position i in the fibonacci word. The result is



(n): ?msd_fib Ei Ak k<n => F[i+k]=F[i+n+k]

The automaton accepts 0*10*. These are the fibonacci representations of 0,1,2,3,5,8,13, … .

def command

This command has the exact same syntax as eval. The only difference is that the automaton outcome of the evaluation is saved in an automata library for later use. As an example consider the following command:

>eval nonsense "a = 3*((b+3)/2)";

You can break it into the following commands:

>def core "x =(y+3)/2";

>eval nonsense "Ec a = 3*c & $core(c,b)";

Now suppose the nonsense we want to compute is the following:

>eval nonsense "a = 3*((b+4)/2)";

If instead we want to make use of our library automaton, we could write

>eval nonsense “Ec a = 3*c & $core(c,b+1)”;

So we can pass any proper predicate as an argument to a library automaton. The only limitation is that the automaton result of the predicate passed as an argument must have only one input.

So where do we save core? We save it as core.txt in the directory “Automata Library”. You can write your own automata in text files and save them in this directory for later use. Just make sure that the text files have UTF-16 encoding. Don’t forget the dollar sign when you are referring to your library automaton.

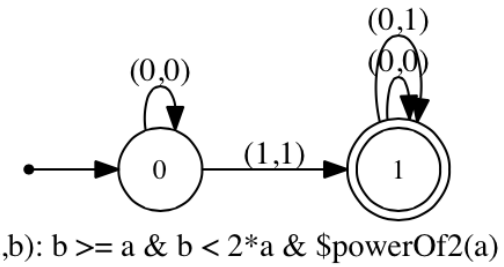
reg command (and paperfolding sequence)

This command provides yet another way of defining an automaton. Just as an example, suppose we want to have an automaton accepting (a,b) where a is the largest power of 2 not greater than b. First we can define an automaton accepting only powers of 2:

>reg powerOf2 msd_2 “10*”;

The result is saved in “Automata Library” directory for late use. Now we can use our powerOf2 automaton as follows:

>eval largestPowerOf2 “b >= a & b < 2*a & $powerOf2(a)”;



(a,b): b >= a & b < 2*a & $powerOf2(a)

To see the syntax of regular expressions we use in Walnut please refer to

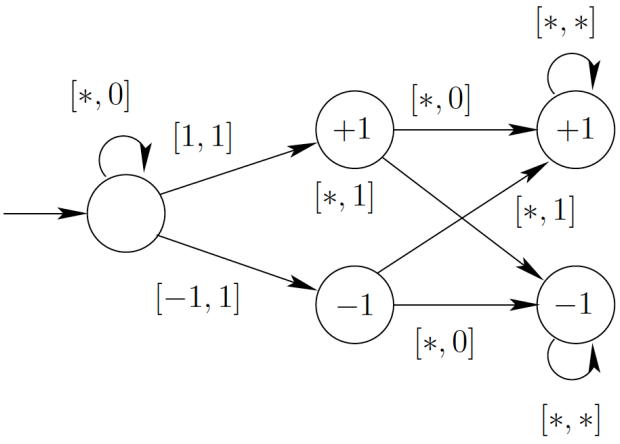
http://www.brics.dk/automaton/doc/index.html

We use this very neat automata library for automata minimization and regular expressions.

Sometimes some or all inputs of an automaton are not numbers. For example if you are familiar with the paperfolding sequences, you'll know that we need a folding instruction and an index: PF[f][i]. The folding instruction f is just a combination of 1 and -1 which tells us which way to fold. So it does not belong to any number system. We can compute the symbol at position i of the paperfolding sequence with folding instruction f using the following automaton, provided that

$$i < 2^{|f|}$$

in other words provided that enough folding instructions are provided (for more information refer to our paper "A New Approach to Paperfolding Sequences"):



The * symbol is our wildcard matching symbol. The first input is the folding instruction an is defined over {-1,1}, and the second input is the index in least significant binary representation. How do we define this automaton in Walnut?

PF.txt

```
{-1,1} lsd_2

0 0
* 0 -> 0
1 1 -> 1
-1 1 -> 2

1 1
* 1 -> 4
* 0 -> 3

2 -1
* 0 -> 4
* 1 -> 3

3 1
* * -> 3

4 -1
* * -> 4
```

We create PF.txt as above and put it in “Word Automata Library”. Look at the first line of this file. It says that the first input is defined over {-1,1} and the second input is in least significant binary. The rest should be easy to understand. Note that all automata defined in “Word Automata Library” are automata with output, whereas all automata defined in “Automata Library” are ordinary automata.

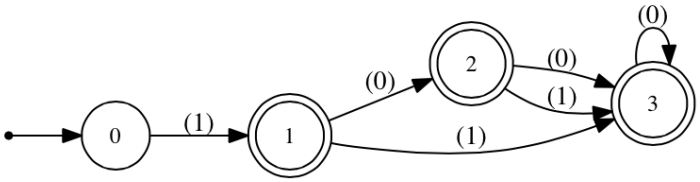
Let’s see an example of paperfolding sequence evaluation in Walnut. We would like to find the order of squares in all paperfolding sequences:

>reg endsIn2Zeros lsd_2 “(0|1)*00”;

This automaton accepts only binaries ending in 2 zeros when represented in least significant digit first representation. We use this automaton to make sure indices to papferfolding sequences are always within the limits of the folding instructions provided.

>eval orderOfSquaresInPaperfoldingSequences “?lsd_2 n > 0 & Ef,i i >= 1 & $endsIn2Zeros(i) & $endsIn2Zeros(n) & (Ak k<n => PF[f][i+k] = PF[f][i+k+n])”;

The result is the following:

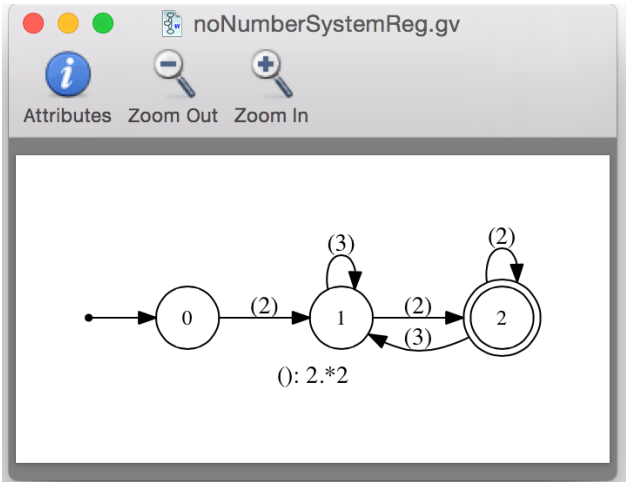


(n): ?lsd_2 n > 0 & Ef,i i >= 1 & $endsIn2Zeros(i) & $endsIn2Zeros(n) & (Ak k < n => PF[f][i+k] = PF[f][i+k+n])

Which means that the only squares appearing in any paperfolding sequence are of order 1,3, and 5.

Let’s get back to reg command. You can use reg command without any number system:
>reg noNumberSystemReg {2,3} “2.*2”;

So noNumberSystemReg will be the following automaton (. means any character):



noNumberSystemReg.txt

```
{2,3}
0 0
2 -> 1
1 0
2 -> 2
3 -> 1
2 1
2 -> 2
3 -> 1
```

One last thing you should know about reg command is that your regular expressions are always defined over {0,1,2,…,9}. So whatever regular expression your write, Walnut computes its intersection with {0,1,2,…,9}*.

List of operators

So you might wonder what are all of the operators you can use in a predicate and what are their precedences. Here is the list of all operators, starting with the most precedent and ending with the least precedent:

```
 *: multiplication by a constant
 /: division by a constant
 +: addition
 -: subtraction
 =: equal
!=: not equal
 <: less than
 >: greater than
<=: less than or equal
>=: greater than or equal
 ~: negation of an automaton
 `: reverse of an automaton
 &: logical and
 |: logical or
 ^: exclusive or
=>: implies
<=>: equivalence
 E: there exits (can be used with multiple variables Ea,b,c)
 A: for all (can be used with multiple variables Aa,b,c)
```

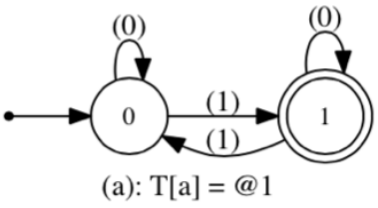
Parentheses overrides any precedence.

Constants

There are two types of constants in a predicate: numbers and alphabet symbols. To indicate an alphabet symbol use @. So for example, we can write

>eval onesInThueMorse “T[a] = @1”;

This gives us:



(a): T[a] = @1