Extending DFAs into Extended Hybrid Finite Automata

Table of Contents

Introduction....3
What is an eHFA?....3
Why an eHFA?....3
Tracking pattern group position....4
Extending DFAs with variable matching....6
Extending NFAs with variable matching....6
Extending DFAs to support variable matching....7
Implemention....29
Appendix A – Thompson's construction....30
Appendix B – The subset construction....30
Appendix C – Anchored and Unanchored regular expressions....30
Appendix D: Shortest and Longest match in a DFA....32
References....32

Introduction

What is an eHFA?

The term eHFA may not be standardized, with in this document is used to refer to a finite automata that provides both DFA and NFA characteristics with extensions providing features (variables, and backreferences) that are not possible to support in a pure DFA, or NFA based on regular languages.

Why an eHFA?

The eHFA is used because it is the best choice to provide advanced pattern matching. A pure DFA provides fast linear matching, with a fixed runtime memory bounds but can grow exponentially large. While an NFA provides for a small pattern matching engine that may not provide a linear match and may not have fixed runtime memory bounds.

The AppArmor HFA combines DFAs and NFAs in such a way as to use DFAs for pattern matching with NFA nodes that act as choke points controlling the expansion of the DFA size. The end result is a FA that has known bounds, and performance like a DFA, and smaller sizes closer to that of an NFA.

The AppArmor HFA is biased towards a DFA and can and does produce pure dfas for matching. It will never produce a pure NFA.

The extended part of the name means the HFA has been augmented with scratch memory and extra notations that provide abilities that are either handled poorly or can not be handled by pure NFAs and DFAs (such as variables, back references, counting constraints, and recognition of limited Dyck languages).

So beyond variable match we need

- counting constraints
- variables
  - as constraints on counting constraints, Dyck languages
  - as pattern match constraints
- back references
- filtering (marks start location, etc much like backrefs and counting constraints)
- subdfas
- anchored vs. unanchored matches and recording the start position (similar to backrefs)
  - checking accept information on every input vs at end of the input
- longest left/shortest left matches
-

Extended FAs

Extending finite automata
use syntax similar to back references for variables \@1
transitions that don't match are not shown and transition to the 0 trap state.

Splitting \@ into something like (\.@+1(\.@+2 (\.@+3|)|)|)+ to trade off state explosion/reduced working memory

Tracking pattern group position

To be able to track the start position for an expression we need to augment the DFA with some memory. When the a transition is made from NFA state 1 (the start state) to NFA state 2 (the first match state of the expression) the input position is saved to memory. This simplistic scheme is sufficient for many simple expressions like the expression in Illustration 18 but fails for the expression in Illustration 1.

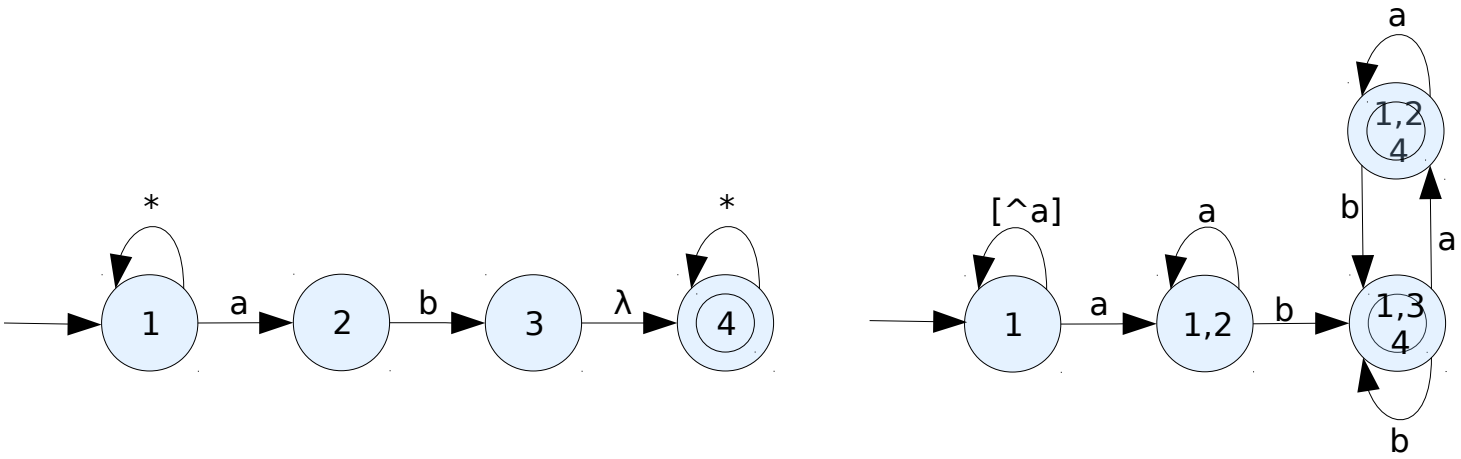


Illustration 1: NFA (left) and DFA (right) for the unanchored expression (ab)

To understand why it fails and how to fix it, consider the expression from Illustration 1 matching against the input 'aab'. Following through the NFA the first 'a' causes a transition from state 1 to state 2, so we save off position 1 as the start position, but the second character is also an 'a', for which State 2 does not have a transition, so the match needs to backtrack and take the alternate transition of state 1 to state 1 for the first 'a' discarding the saved start position value. We can now match the remainder of the input 'ab' to get a match.

Now let us consider the same input of 'aab' against the DFA. The first 'a' causes a transition from state {1} to state {1,2}. This transition contains the NFA state 1 to state 2 transition so the input position is saved. State {1,2} then transitions to itself which causes also contains the NFA state 1 to state 2 transition causing the saved input position to be reset to position 2. The match then continues for the remainder of the input.

This simple reset of the start position is not however enough to handle the example in ???? for input aaab. This occurs because the first 2 characters of the input match the first two characters of the pattern,
and we don't know until the 3rd character whether the first character is a match. In a DFA this means that multiple potential start positions must be tracked simultaneously.

Pattern aab with input aaab

pattern a.*b with input ????

hrmmm ababa
oh when groups start and how to track them

Foo

- conversion to dfa
  - start spin state
  - tail spin state if consuming all input
    - don't have to match all input if looking at accept values

- storing match start positions
-

Shortest left and longest left matches in a DFA (appendix ?)

-right anchored by definition is always a longest match (though not necessarily longest left)

Extending DFAs to match variables
To extend DFAs with variables we need to understand what extension are needed to dfa states and transitions as well as how these affect DFA construction. To do this we will extend NFAs to do variable matching and then extend the rules of the subset construction.

Nfas can have multiple transitions for a single character – try each transition
Nfas can have lambda transitions

Need to cleanup to indicate the difference between anchored and unanchored expressions.

- how to handle tail anchored expressions
- how to handle head anchored

cleanup NFA transition vs. DFA transition.

NOTE: add info about notation
a (b, c) – shows set with states reached via lamda transition listed in ( ), this is not needed but is done as a guide to help show where parts of the set are coming from. This is also the only part of the set that will ever pick up the |= qualifier as it can only be inherited from λ|=

Extending DFAs with variable matching

Deterministic Finite Atomata (DFA) are a finite atomata (FA) that at any point have a single transition for any input symbol and do not have null transitions

Extending NFAs with variable matching

We start by extending the NFA with a new state type that performs variable matching (Illustration 2). It is modeled after Kleen closure on a state that matches arbitrary input characters (appendix ???), except the variable state transitions are conditional on the input characters matching the values in the variable. The condition on the transition is indicated by using a | symbol followed by the condition for the match, eg. .*|@ means match any character given it matches the current position in the variable. We shorten the .*| to the slightly shorter *| in the following discussion as it remains non-ambiguous, and less cluttered in the more complex illustrations.

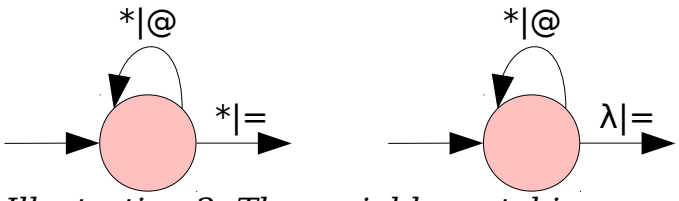


Illustration 2: The variable matching state. Left - a state that does not match null variables. Right - a state that can match null variables.

Two versions of the matching state are provided in Illustration 2, the left hand version can not match variables that are empty, while the right hand version can. The meaning of *|@ is slightly different in each version, in the left hand version it will consume all input matching the variable except the one matching the last character which is matched by *|=. Where in the right hand version *|@ matches all characters within the variable and then there is a lambda transition out of the variable state when the variable is fully matched. The left hand version can result in smaller DFAs, while the right hand version is more flexible. For the purposes of the following discussion we will use the λ|= variant. After the basics of the extended DFA construction are established we will explore this and other constraints and trade offs that can be made.

The use of the variable match state in an NFA requires that the match be augmented with a positional value. ???? Need to know where in the variable match we are, blah, blah, blah like in the DFA below
????. How much do we need to explain before transitioning to the DFA

Extending DFAs to support variable matching

Now that we have the basis for supporting variable matching in NFAs we would like to extend the support to DFAs. To do this we extend the subset construction so that it can convert an extended NFA that supports variable matching into a DFA that supports variable matching.

The subset construction (Appendix B – The subset construction) creates a DFA that is the equivalent of doing a breadth first NFA match, that is it works on sets of NFA states. Variable matching extends both NFA states and transitions with conditional information that will need to be carried into a variable matching DFA. To develop the rules needed for this we will work through a set of examples constructing NFAs matching a pattern, stepping through the NFA and then building its associated DFA.

Example 1: A pattern matching a single variable (\@)
We start by building a DFA that matches a single variable exactly (Illustration 4), and compare it to the DFA constructed for the expression .* (Illustration 3) which we have modeled the variable pattern match on.

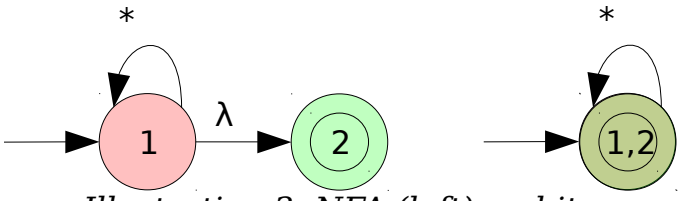


*Illustration 3: NFA (left) and its associated DFA (right) for matching .**

The NFA for the .* expression converts into a DFA with a single state that consumes all input, and is always the final accept state.

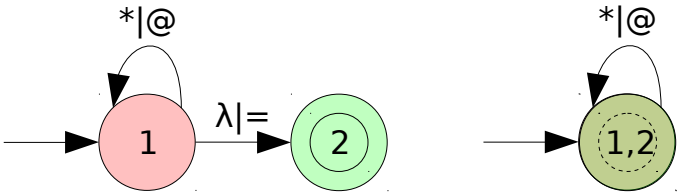


Illustration 4: NFA (left) and its associated DFA (right) for matching \@ where the variable string length >= 0

The variable match pattern is similar except the transition for input is conditional. When considering the next input symbol it is compared to the value at the current variable position, if there is a match the transition back to the variable state is taken and the match position within the variable is incremented. If the input symbol does not match, the DFA state does not have a matching transition defined so the DFA transitions to the non-matching trap state, which is not shown.

If the input is longer than the variable match the matching engine can either terminate the match in the accept state (non-anchored tail match), or fail - transitioning to the trap state as there are no matching transitions defined for the input (anchored tail match).

The accept condition for the variable match is different than that of the .* expression. In the .* expression a match can be accepted for any character, this is not the case for the variable match its accept is conditional on the variable match completing (|= condition). This is indicated by using a dotted line for the inner circle of the accept state in the diagram.

We can build an augmented transition table for the subset construction. Each State of the DFA is represent as a list of NFA states, with states reached via the λ transition shown in parenthesis (). For each input we have three possible values (columns): the input 'a' used by states that aren't a variable match state, or by a variable match state when a variable does not match; 'a|@' for when the input matches the variable, and 'a|=' for when a transition is conditional upon a variable having been matched.

The DFA states, listed in column 1, contain the set of NFA states that compose them; with the NFA states that where reached conditionally on a variable match marked with |=, and parenthesis () being used to group sets of states that are reached together. A similar notation is used for the transition columns.

DFA State	**a**	**a|@**	**a|=**	**b**	**b|@**	**b|=**
1 (2|=)		1(2|=)			1(2|=)	

Table 1: Transition table for subset construction of \@

Thus for Illustration 4 we have one state 1 (2|=), composed of NFA states 1 and 2, with 2 being reached by a conditional λ transition. There are no transitions

for input 'a' as neither NFA state 1 nor 2 have a plain (unconditional) transition. There is a transition back to DFA state 1 (2|=) on input 'a' given it matches the variable at its current position (a|@) from NFA state 1. And there is no transition on 'a|=' because no NFA state has a transition that is conditional on the state being reached after a variable was matched (we will see this in later examples, note this does not include the initial transition (|=) when the variable is matched). The transitions for input 'b' mirror those for input 'a'.

Example 2: A pattern matching a variable followed by a letter (\@a)
We now examine what happens when the pattern to match is extended by a single trailing character.

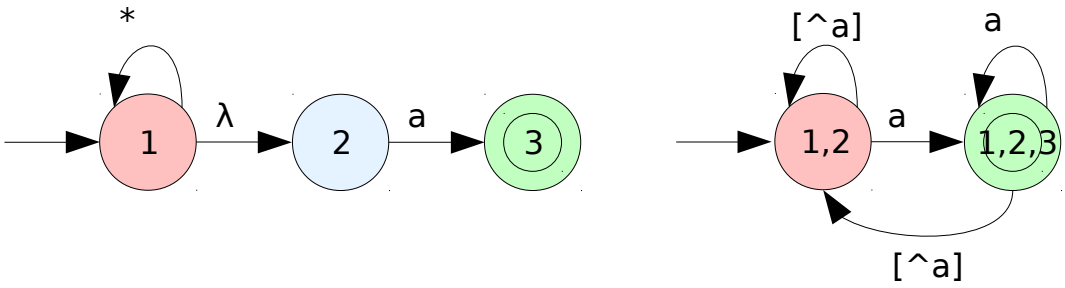


*Illustration 5: NFA (left) and its associated DFA (right) for matching .*a*

The DFA for the .*a (Illustration 5) reduces to two states, the {1,2} state which consumes most characters and the {1,2,3} state which consumes the letter a.

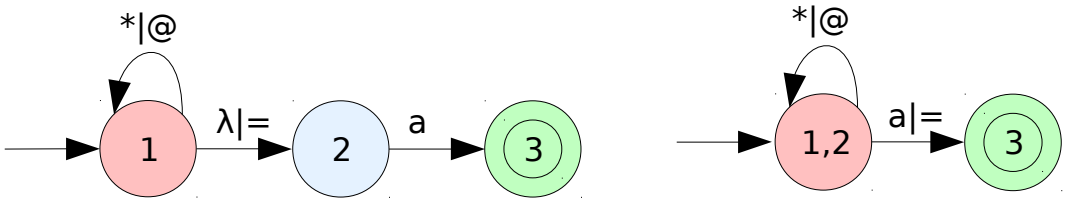


Illustration 6: NFA (left) and DFA (right) for matching \@a

The variable match DFA (Illustration 6) is again similar to the DFA created by the .*a expression. The λ transition pushes its condition (|=) on to the states and subsequent transitions reached by the conditional transition. To be more precise, if state is by reached λ-closure that traverses a λ|= condition then the transition also must inherit that condition.

Stepping through the subset construction we can see where some of the differences occur. Table 2 shows the transitions generated by the subset construction.

DFA State	**a**	**a|@**	**a|=**	**b**	**b|@**	**b|=**
1 (2|=)		1 (2|=)	3		1 (2|=)	
3						

Table 2: Transitions for the subset construction on \@a

The subset construction as shown in Table 2 has been modified to handle the NFA |@ and |= transition conditions. States reached via a condition are marked with the condition, which will then be used later to determine if the condition is applied to a transition. The |@ transition is implicit (not shown) in the transition table NFA state set as it can always be determined because of the variable state transitioning to itself.

Walking through the subset construction, we start in NFA state 1, and then do λ-closure(1) to get to the DFA start state of {1,2|=}. The 2 in the set picks up the |= condition because it is reached by the λ|= transition in the nfa.

Building the subsets for the transitions from the DFA start state {1,2|=}, from NFA state 1 there are no unconditional transitions inputs symbols for 'a' and 'b'. The a|@ and b|@ transitions loop back into NFA state 1, and there are no other potential transitions, λ-closure({1}) is then applied resulting in {1,2|=}. For the 2|= entry, in start state set, the only defined transition is a to NFA state 3, however since 2 is marked with |=, the transition is a|= instead of 'a'. λ-closure({3}) is then performed, but there are no λ transitions from 3 so the final state is {3}. Note the |= mark that was on 2 does not carry to state 3 as it is not reached via a conditional λ-transition. For DFA state {3} there are no possible transitions, so there are no entries in the table.

The mutually exclusive handling of the |@ and |= transition results in a dfa that doesn't loop on it self like the dfa for '.*a' does. This makes sense as the '.*a'

expression can't know in advance when the '.*' portion of the expression matches so any 'a' is treated as a potential final 'a'. With the variable match it is known when the variable is matched so only if that match is followed by an 'a' will the match succeed.

Example 3: A pattern matching a variable followed by any text (\@.*)
We now extend example 2 to examine what happens when a variable match is followed by an arbitrary match.

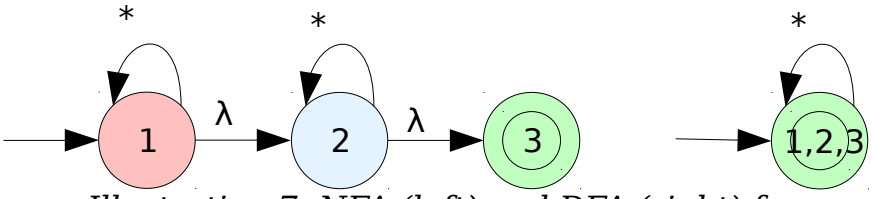


Illustration 7: NFA (left) and DFA (right) for matching ..**

Two consecutive arbitrary matches (.*.*) merge into a single match pattern as shown in Illustration 7. The variable match shown in Illustration 8 is much closer to the results from Illustration 5. This is because the variable match keeps the states from collapsing together because of the behavior of the conditional transitions.

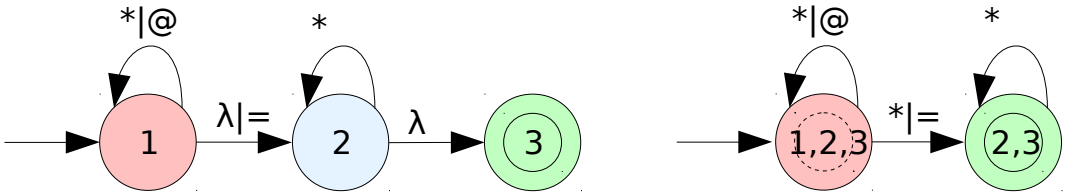


*Illustration 8: NFA (left) and DFA convertion (right) for matching \@.**

We start with NFA state 1 and then do λ-closure(1) to get to the DFA start state of {1,2|=,3|=}. We again see the formation of a conditional accept state, this is because NFA state 3 is reached through a conditional λ-closure as shown in Table 3. The state 3 is also marked with |= because it is reached via state 2 during λ-closure which can only be reach via a conditional lambda transition (λ|=).

Once the DFA start state is formed we follow the transitions for it constituent NFA states. NFA state 1 has conditional transitions to 1 on a|@ and 2 on a|=, which then have a λ-closure(2|=) to 3|=. 2|= has a transition to 2, but since 2|= was reached via a |= transition we the a|= and b|= columns instead of the a and b columns for its transitions. Note how in the table the transition for a|= and b|= do not mark the set of states it is transitioning to as conditional. That is because taking the transition guarantees that condition (|=) has been met so the conditional no longer needs to be carried. The a|= column could be tracked in the 'a' column but is not because it makes following where a given transition originates from easier. The |= mark is used by the subset construction to find the transitions that inherit the |= condition during λ removal, and also to determine when an accept state has been entered conditionally.

The transitions for the start state {1 (2|=,3|=)} reach a new DFA state {2,3}, so we enter that state in the table and follow its transitions. Neither state 2 or 3 have a conditional match, nor were they reached conditionally so only the unconditional columns 'a' and 'b' are used for the input match. 2 transitions to itself and then λ-closure is used to reach state 3. Since state 3 is reach via an unconditional λ transition the DFA state {2, 3} is an unconditional accept state.

The DFA generated loops to state {1 (2|=,3|=) } while the input matches the variable. If the last input character matches the last value of the variable, then state {1 (2|=,3|=)} accepts the match. If there are any more input characters beyond the variable match a transition to state {2,3} is made and the rest of the input is unconditionally matched.

DFA State	**a**	**a|@**	**a|=**	**b**	**b|@**	**b|=**
1 (2|=,3|=)		1 (2|=,3|=)	2,3		1 (2|=,3|=)	2,3
2,3	2,3			2,3		

Table 3: Subset construction for \@.*

The next two examples will revisit examples 2 and 3 to explore what should be done when the variable match is optional.

Example 4: an optional variable followed by a letter (\@?a)
In this case we must match an 'a' optionally proceeded by by a variable match. This means that the generated DFA will have to be able to deal with the first character if it is an 'a' potentially being the final character or the first character of the variable.

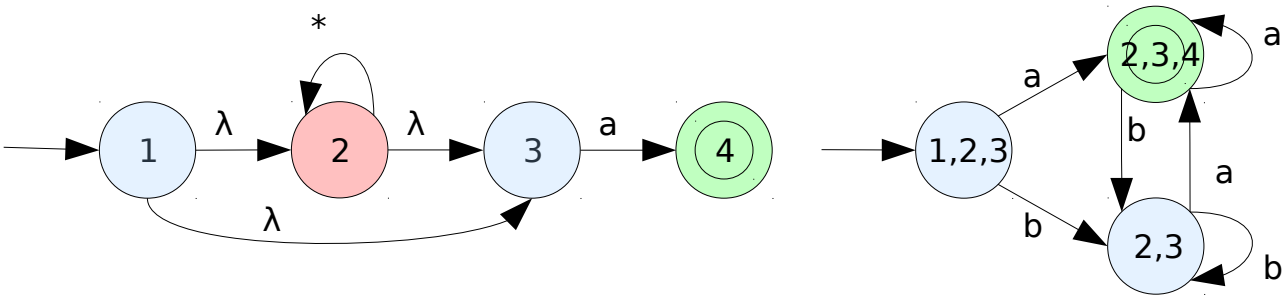


Illustration 9: NFA (left) and DFA (right) for matching (.)?a*

Comparing Illustration 9 and Illustration 10 we once again see that while the NFAs for the variable match and the pattern match it is modeled on are almost identical the resultant DFAs are different. This will continue to be the case, as the conditional transitions, not only provide more potential transitions to encode for but also more combinations of state sets to transition to. As such this is the last plain pattern match NFA and DFA to be presented for comparison to the variable match NFA and DFA.

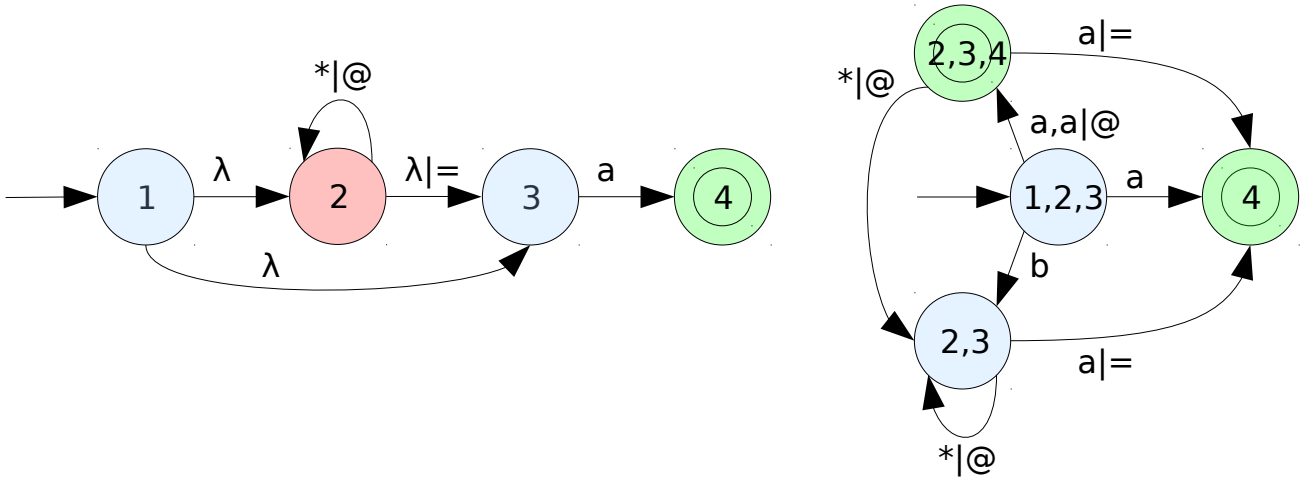


Illustration 10: NFA (left) and DFA (right) for matching \@?a

FIX ME: 1,2,3 – b → 2,3 is wrong in dfa it should be b|@, might as we make 1,2,3 – a → 4 a,a|= just to show its both of those like the a,a|@. Update discussion below for change

There are two important point to focus on in Illustration 10, and its transitions in Table 4. The first is the start state {1,2,3} does not have a conditional |= marker

despite state 3 being reached via a conditional |= transition from state 2. This is because both 3 and 3|= transitions are always to the same set of states, as |= after lamba removal just marks a state thats transition should be conditional on the |= match. The second point is the transition from {1,2,3} to {2,(3|=),4}. It is labeled with both 'a' and a|@ transitions, and there is a separate transition for just 'a' that goes to {4}. Looking at Table 4 we can see the transitions for {1,2,3}. The transition for 'a' is listed as {4}, while the transition for a|@ is listed as {2 (3|=)}, and yet in the DFA we have 'a' transitioning to {4} and 'a' and a|@ going to {2,(3|=),4}.

This happens because the conditional forces the split. For the input character 'a' there are two choices, if the a|@ condition is false it can transition directly to {4}. However if a|@ is true the NFA transitions to both {4} and {2 (3|=)}. This is no different than an ordinary NFA having two transitions listed for an input character, where the destination state sets are combined by the subset construction to find the final state. The difference being that the |@ condition is true only sometimes giving us two different states to transition to.

DFA State	**a**	**a|@**	**a|=**	**b**	**b|@**	**b|=**
1 (2,3)	4	2 (3|=)			2 (3|=)	
2 (3|=) 4		2 (3|=)	4		2 (3|=)	
2 (3|=)		2 (3|=)	4		2 (3=)	
4						

Table 4: Subset construction for \@?a

While the syntax of a,a|@ was used in this example DFA to emphasize that this transition is using the targets for both 'a' and a|@, it is incorrect and confusing (a DFA can only have a single transition for an input character, and the state label encodes which NFA states have been combined). The meaning of |@ in the NFA and table is different than that in the DFA, in the DFA |@ transitions to everything from a|@ and 'a' in the NFA. All other DFAs in this paper will use only a|@ or 'a' to label transitions, dependent on whether the transition is conditional.

Example 5: an optional variable followed by any text (\@?.*)
We now revisit example 3, this time with the variable match being conditional.

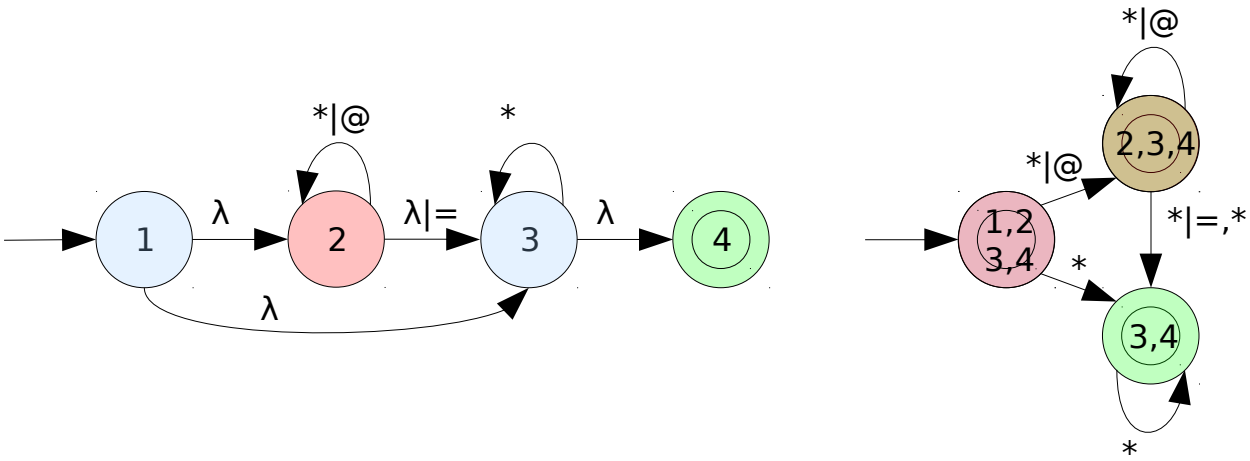


*Illustration 11: NFA (left) and DFA (right) for matching \@?.**

??? can we actually remove though??? We don't follow |@ but can we follow a|= as a? What if we are in a case of |!=. That is our var does NOT match.

Following through the subset construction shown in Table 5, we see the start state is {1,2,3,4} and following through the NFA, we see for λ-closure (1) there are two paths that can be taken 1(2, 3|=, 4|=) and 1(3, 4). Looking at the transitions in Table 5 we can see that when unconditional and condition transitions overlap, like in 3|=, 3 and 4|=, 4 in 2(3|=,3,4|=,4) they make the exact same transitions. This will always be the case as |= transition is a marker on the state used by the subset construction it does not indicate a separate state. So when merging the conditional and unconditional states in the subset that are the same other wise (3|=, 3) we can remove the |= conditional.

Removing the |= from states in this example reduces 1(2,3|=,3,4|=,4) to 1(2,3,4) and 2(3|=,3,4|=,4) to 2(3,4), as shown in Illustration 11. The start state is unconditional accepting, which is correct as the NFA can accept null input. Further examples will use the above |= conditional removal simplification, as part of the subset construction.

DFA State	**a**	**a|@**	**a|=**	**b**	**b|@**	**b|=**
1 (2,3|=,3,4|=4)	3(4)	2(3|=, 4|=)	3(4)	3(4)	2(3|=,4|=)	3(4)
2(3|=,3,4|=,4)	3(4)	2(3|=, 4|=)	3(4)	3(4)	2(3|=,4|=)	3(4)
3(4)	3(4)			3(4)		

*Table 5: Subset construction for \@?.**

The other points to focus on in this example are the transitions for {2,3,4} (ie. 2(3|=,3,4|=4) in the transition table. Its *|@ transition, contains the a|@ and b|@ transitions, and both of these are examples of the a|@ transition from the

previous example, with the transitions sets of a and a|@ being combined to find the final target set.

The transition from {2,3,4} to {3,4} is double labeled with *|= and '*'. Unlike in the previous example this is actually the case as 'a' and 'b' transition to {3,4} when no conditionals are met (not a|@ and not a|=), and a|=, b|= transition to {3,4} as well. Notice that in this case the subset construction transition listed in Table 5 for a|= is the same as the final target state, because 'a' transition in the table is the same, so the merging of the two sets does create a new target state.

With both *|= and '*' transitioning to the same state, one could question why there isn't conditional simplification being done like was done with |= condition removal in the state labeling. For this case such simplification could be done but whether it is useful will depend on implementational details, so we choose to leave the full transition information in the examples.

There is one more point in this example that is worth taking note of. The accept state is reached both conditionally and unconditionally via λ transitions, and all the resultant DFA states have unconditional accept states. With proper DFA minimization this example could be reduced to a single accept state, with no conditional transitions. We will revisit this in the section on minimizing DFAs.

Example 6: the letter 'a' followed by a variable (a\@)
We now reverse some of the previous examples and see what happens when the variable match is at the tail of the expression.

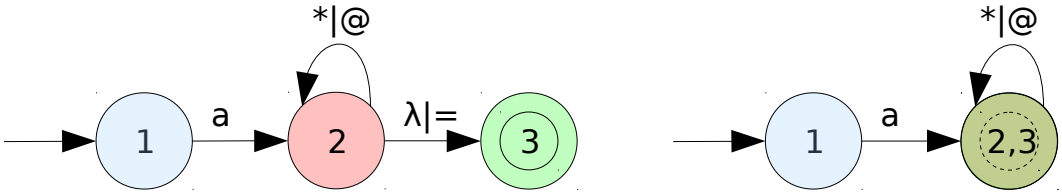


Illustration 12: NFA (left) and DFA (right) for matching a\@

There is nothing surprising or to build on from this example, the subset construction works as we expect it should, both 'a' and then the variable must be matched before the conditional accept state will accept the input.

DFA State	a	a|@	a|=	b	b|@	b|=
1	2 (3|=)					
2 (3|=)		2 (3|=)			2 (3|=)	

Table 6: Subset construction for a\@

Example 7: an arbitrary number of 'a's followed by a variable (a*\@)
We now look at an example that forces us to extend both the subset construction and how the variable matching is performed.

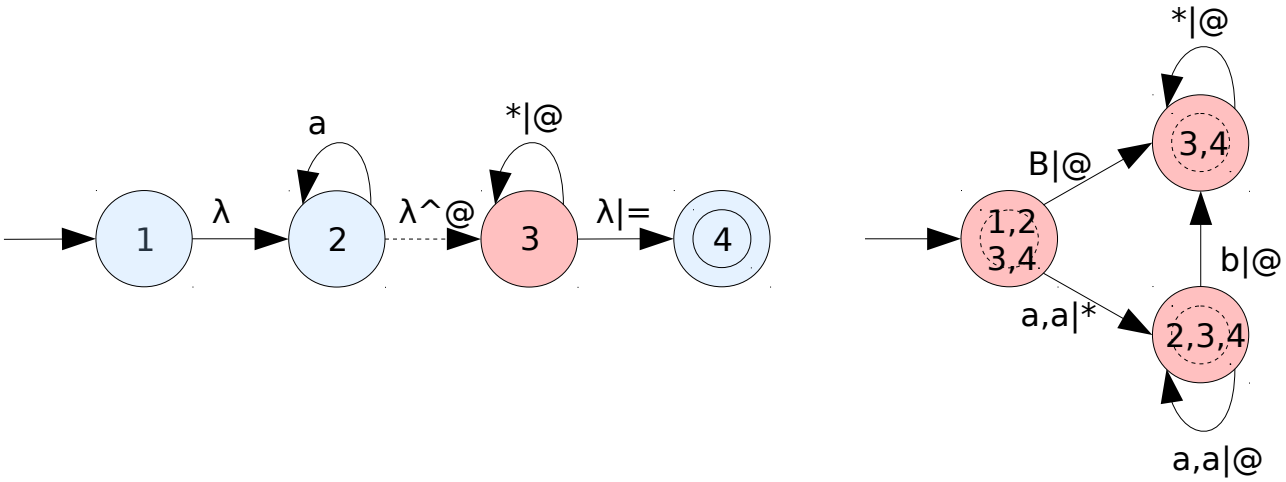


Illustration 13: NFA (left) and incorrect DFA from current subset construction rules (right) for matching a\@. Note: λ^@ is a new notation explained below*

At first glance the DFA generated in Illustration 13 would seem correct and it will work for some input and variable combinations. It correctly matches the null string if the variable is empty. It also correctly handles a transition to state {3,4}, which only does variable matching, when a 'b' is matched in the variable. However it handles the interaction of a* and \@ matching incorrectly.

DFA State	a	a|@	a|=	b	b|@	b|=
1 (2, 3, 4|=)	2 (3, 4|=)	3 (4|=)			3 (4|=)	
2 (3, 4|=)	2 (3, 4|=)	3 (4|=)			3 (4|=)	
3 (4|=)		3 (4|=)			3 (4|=)	

Table 7: incorrect subset construction for a\@*

To determine what is wrong we examine a couple input and variable values, using the example pattern in a fully anchored situation (ie. ^a*\@$)

Remaining input	State transition	Variable position
aa	{1,2,3,4} → {2,3,4}	0 → 1
a	{2,3,4} → {2,3,4}	1 → ?

Table 8: Transition for @ = "a" with input aa for pattern ^a\@$*

Examining the transitions in Table 8 we see a problem even for a very simple set of input and variable values. Clearly the input should match, as the variable matches last the 'a', and a* consumes the first 'a'. The problem with the match is with the character match position in the variable. A variable match starts on the transition from state {1,2,3,4} to state {2,3,4} as the 'a' input matches both a* and the start of the variable, causing the position to increment.

If we were not to consume any more input we would have a valid match. The a* expression consumes nothing and 'a' is matched by the variable. If the next input symbol was 'b' the match would stop here and for an unanchored match, this would have been correct. However this does not work for longest left match, nor tail anchored matches.

So we need to modify our variable match and position increment rules, in this case we could get away with resetting the match position to 0 and rerunning the match against the remainder of the input. This however would not work for the input “aaa” with a variable value of “aa”, as shown in Table 9.

Remaining input	State transition	Variable position
aaa	{1,2,3,4} → {2,3,4}	0 → 1
aa	{2,3,4} → {2,3,4}	1 → 2
a	{2,3,4} → {2,3,4}	0 → 1

Table 9: Transitions for @ = "aa" with input aaa for pattern ^a\@$*

In this case the variable matches with the next input character, and then resets to 0 but there is not enough input to fully match the variable again. A successful match needs to begin when the first variable match is at position 1. The DFA either needs to know that the first match being performed will not lead to a successful match or it needs to be able to track another set of positions simultaneously.

The DFA can not know that the first match being performed will not lead to a successful match unless it can look ahead by completing the match. This can be done by using a depth first search and backtracking. However this may require rerunning match comparisons several times for each input character and does not work for online matches where the input is not available to revisit.

Instead of using backtracking the variable will be allowed to track multiple positions (breadth first). A variable match starts when the match variable is transitioned to, but not every transition into a variable consuming state starts a match. The previous examples worked because there was only a single transition into the variable match state (state 2 to state 3 in Illustration 13), and transitions with the variable match state looping back on it self did not result in the need for a new match to begin.

We augment the NFA and DFA with a new flag indicating whether a transition instantiates a new match for the variable. For the subset construction when a transition contains a transition flagged to instantiate a match, the DFA transition also picks up the match. That is to say if the NFA transition 1 → 2 is flagged as the start of a match then any DFA transition {X} → {Y} where {X} contains 1 and {Y} contains 2, also is flagged as instantiating a match. We indicate this as shown in Illustration 14 with a dotted transition line and ^@ to indicate the instantiation of a variable match.

<< diagram of dummy start state >>

Match instantiation as described won't work if the variable consuming state is also the start state. To handle this situation a dummy start state with a null transition to the variable consuming state is added to the NFA. This will generate a DFA with the needed transition but also results in an extra state in the produced DFA as well.

The dummy transition solution will not work for acceptance of a null input stream, when a variable is allowed to have a null value. There are two solutions to this problem, the start state when it also a variable consuming state can be flagged and it can instantiate its variables at the start of the match. Alternately conditional consuming states can always return a match for null valued variables.

The addition of potentially multiple match positions for a variable requires further modifications to variable matching and subset construction. It is possible that any of the positions may not match on a given transition. If this occurs the instance is discarded as it will not lead to a successful match.

If there are more than one variable instance and one of them resolves to a match, the others may still be in a valid matching state. That is to say there are both |= and |@ transitions happening simultaneously. We deal with this just as we have for 'a' and a|@ happening simultaneously, that is we create a transition to a new state created the set of 'a', 'a|@', and 'a|=' transitions. Instead of labeling the transition with |@ and |= (which are still valid conditions and could exist independently), we represent the new transition condition by '|*'. This means for a state that matches against a variable we now have 4 potential transitions per character 'a', 'a|@', 'a|=', and 'a|*'.

When a '|=' or '|*' transition occurs the variable instance that caused the match is removed from the set of variable positions (ie. Its consumed). Conditional accept states do not remove variable instances but if the match continues to run those instances will be removed, as they no longer match anything.

If a matching engine is doing a tail anchored match, then it should be run, ignoring accept states, until all input is consumed. If the engine is doing a match that is unanchored on the tail then, if it is doing a shortest left match it should stop at the first accept state, and if it is doing a longest left match it should remember the last encountered accept state until input is exhausted or the match engine enters the trap state indicating that no match is possible.

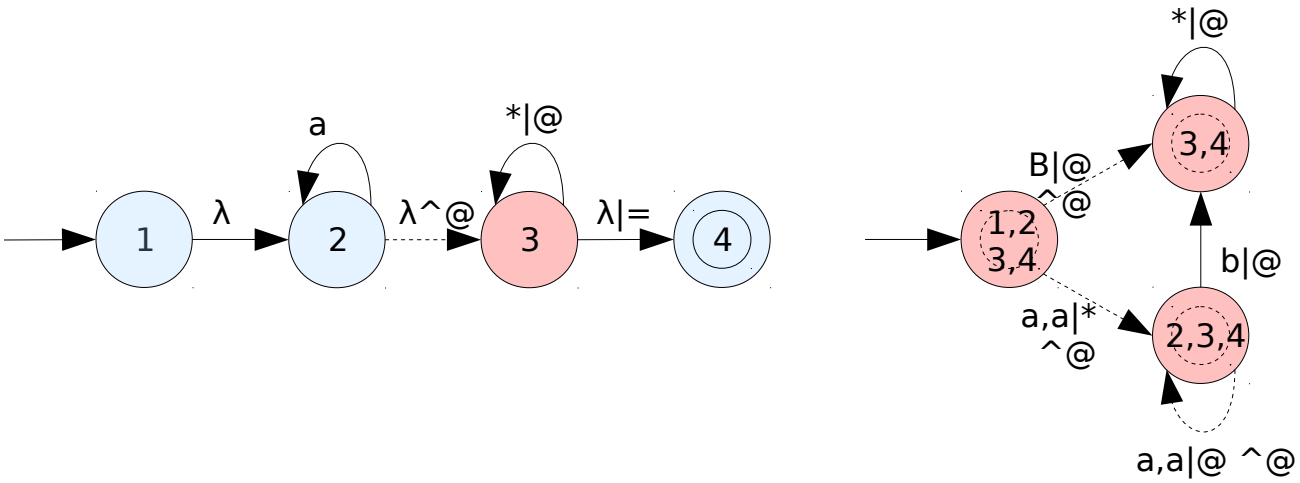


Illustration 14: NFA (left) and (DFA) right with variable instantiation transitions for a\@*

Examining Illustration 14 and its subset construction in Table 10 we can see the mark for the variable instantiating transition but not a '|*' transition. That is because it is not needed for this particular pattern as the DFA doesn't contain any a|= transitions.

DFA State	a	a|@	a|=	b	b|@	b|=
1 (2, 3, 4|=)	2 (3, 4|=)	3 (4|=)			3 (4|=)	
2 (3, 4|=)	2 (3, 4|=)	3 (4|=)			3 (4|=)	
3 (4|=)		3 (4|=)			3 (4|=)	

Table 10: Subset construction using the revised rules for a\@*

Table 11 shows the transitions of the previous example reworked for the new rules. The first input character causes the first variable to instantiate and move to position 1. The next input char consumed moves the first instance to position 2, and starts a new instance which is at position 1. At this point if an unanchored match was being used the state {2,3,4} could be tested for the accept condition and find it to be true. Assuming an anchored match is being done, the state is not checked for acceptance and the next input character is consumed. This causes the matching instance to be removed, the next instance to be moved from 1 → 2, and a new instance to be added.

Remaining input	State transition	Variable position
aaa	{1,2,3,4} → {2,3,4}	0 → 1
aa	{2,3,4} → {2,3,4}	~~1 → 2~~, 1 → 2, 0 → 1
a	{2,3,4} → {2,3,4}	0 → 1

Table 11: Reworked transitions for @ = "aa" with input aaa for pattern ^a\@$*

Example 8: (a*\@a)
We now examine an example that uses the new '|*' transition condition. There is nothing new in this example that requires further extension but it makes full use of the revised subset construction.

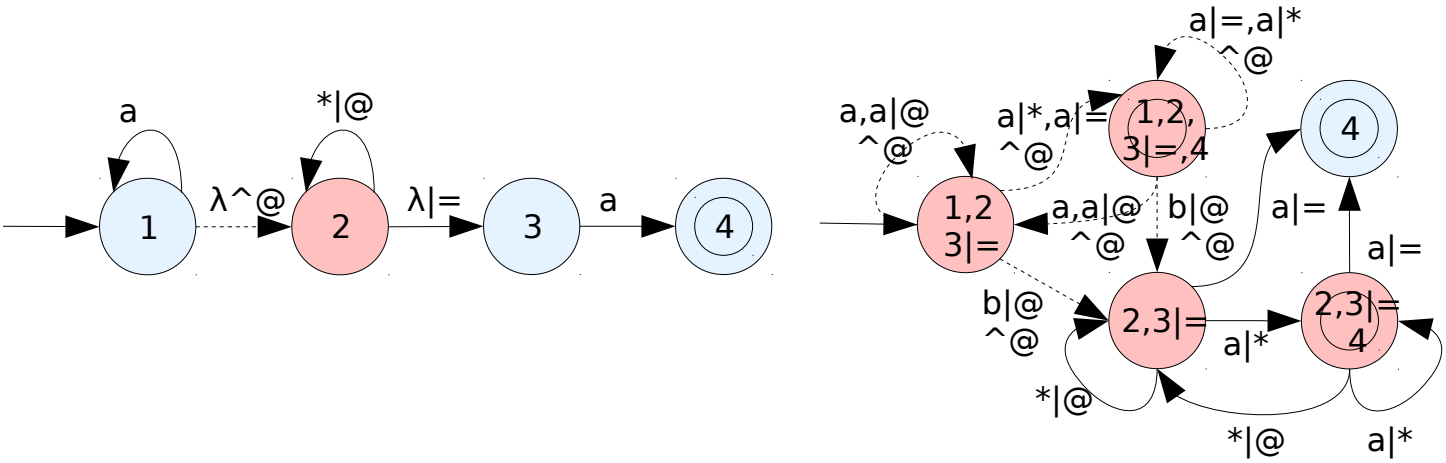


Illustration 15: NFA (left) and DFA (right) for a\@a*

Looking at Illustration 15 we see the existence of a transition types, some times sharing a transition with another and sometimes being completely separate. The |* transition from 2(3|=) is particular interest because it forces a new state 2(3|=)4 to exist, that can not be reached by any transition, not even the a|= and a|@ transitions that it is derived from.

DFA State	a	a|@	a|=	b	b|@	b|=
1 (2,3|=)	1(2,3|=)	2(3|=)	4		2(3|=)	
1(2,3|=)4	1(2,3|=)	2(3|=)	4		2(3|=)	
2(3|=)		2(3|=)	4		2(3|=)	
2(3|=)4		2(3|=)	4		2(3|=)	
4						

Table 12: Subset construction for a\@a*

Example 9: (a*\@a*)
This example clarifies a part of the subset construction that we have so far glossed over. The DFA State column in the subset construction tables has shown the |= mark, but the DFA diagrams have dropped it. This is not entirely accurate as we will see.

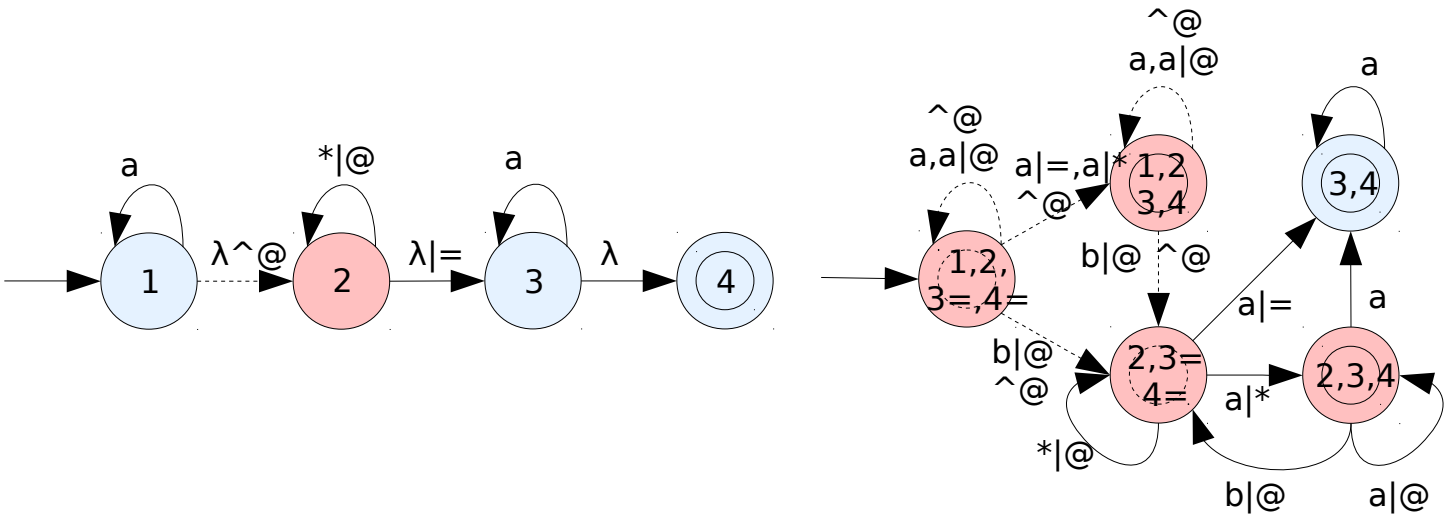


Illustration 16: NFA (left) and DFA (right) for a\@a**

Looking at Illustration 16 and Table 13 we can see the existence of two pairs of states with the same set of NFA states 1(2,3|=,4|=), 1(2),3(4) and 2(3|=,4|=), 2,3(4). Each of these pairs differ in two ways. For each pair one is conditionally accepting and one is not, and one has a transition on |= and the other does not. We need to examine what is happening to determine why this warrants a new state and when it doesn't.

The start state 1,(2,3|=,4|=) and its twin 1(2),3(4) cover the case where input and variable consist only of 'a's. The start state covers the cases where the input terminates on a variable match. Its twin covers input conditions where at least one potential variable match has been encountered, meaning the requirement that the variable exists has been met and a match is guaranteed as long as the remaining input consists of 'a's.

If these two states where combined into a single state we need to determine whether the resultant state would conditionally accept. If it was not conditionally accepting then any set of input that just consisted of 'a's would be accepted as long as the variable contained just 'a's as well. However this would lead to incorrect matches when the variables length is longer than the input.

The alternate case of making the combined state conditionally accepting is more interesting, as it actually will work for a single valued variable. To explain why the state split is done we need to look at the 'b|@' transition out of 1(2),3(4). The transition of 'b|@' out of 1(2),3(4) may seem strange in that we known 1(2),3(4) can only be reached if the variable transitioned on only contains 'a's. However the subset construction doesn't have a memory so it doesn't take the previous condition into account, nor is there anything in the rules that prevents a variable that is partially matched from containing a 'b'.

The extended subset construction supports multivalued variables, the only requirements are that when a variable match is instantiated, all its potential match values are instantiated and that the acceptance check tests multiple

values for acceptance. The state will then match and consume all potential values discarding those that can not match at the given position.

The other peculiarity for to examine from {1,2,3,4} is that it doesn't have a |* transition despite being a variable consuming state with |@ transitions. To understand why this occurs, we need to look at how we have extended the NFA and subset construction.
We begin by examining what happens when transitioning into a variable consuming state. On entry the variable state tests for a match before any |@ transitions can occur, this is shown by λ-closure on the entry transition. The subset construction has two sets of states it needs to simulate for this transition, the variable state, and the set of state conditionally reachable via λ-closure. The modified λ-closure ensures that this case is covered and that states reached via the conditional transition are marked so. Accept states that are marked as |= have their acceptance become conditional as well as, they are only valid as long as the condition is met. Since |@ and hence |* transition possibilities can't occur yet all cases have been covered.

Looking at the transitions once we have entered the variable consuming state, there two options the |@ transition and the λ-closure set of states. The |@ transition always transitions back to the NFA variable consuming state, so any DFA state reach via |@ will contain it and its λ-closure set of states. The |= transition comes from states from the λ-closure set. If a state in that λ-closure set does not have any transitions of its own then there are no |= transitions for it to contribute, at most it will contribute a conditional accept. If it does have a transition the we have a |= transition and hence also a |* transition.

Looking at Illustration 16 we see that NFA state 3 has a transition on 'a' and the λ-closure for a|@ in Table 13 reflects this as well. The |= and |* transitions are present for {1(2,3|=,4|=)} and are missing for {1(2),3(4)}. The reason lies in the conditionals marked on the states, that makes these two state different. State {1(2,3|=,4|=)} represents being in 4 different NFA states, 3 and 4 conditionally so, where state {1(2),3(4)} represents being in the same set of NFA states but with no condition. From these states we then do a breadth first walk to the next set of states. {1} → {1,2,3,4)}, {2} → {2,3,4}, and {3} → {3,4}. So the final set of states that we can be in is {1,2,3,4}, however for some of these states we have paths that are both conditional and unconditional. We can reach 3 and 4 conditionally but we can also reach them unconditionally, meaning we will always reach them whether the condition is true or not, effectively making the condition meaningless. The conditional transitions that overlap with no conditional transition can be removed. The remaining set including conditions represent a unique combination to reach the set of states. In the case of case of {1(2),3(4)} all conditions have been removed, but it is unique from its sibling {1(2,3|=,4|=)} because it can only be reach under different conditions, and that must be reflected. Hence {1(2,3|=,4|=)} and {1(2),3(4)} become distinct states in the DFA.

What happens if we do |= and |* transitions that are removed. Ie show what

happens when we treat {1(2),3(4)} as {1(2,3|=,4|=),2(3|=,4|=),3(4)}. But the example doesn't really argue the generic case.

DFA State	a	a|@	a|=	b	b|@	b|=
1,(2,3|=,4|=), 2(3|=,4|=), 3(4)	1(2,3|=,4|=) 3(4)	2(3|=,4|=) 2(3|=,4|=)	3(4) 3(4)		2(3|=,4|=) 2(3|=,4|=)	

	a - a	a|@ - a | a|@	a|= - a + a|=	a|* - a + a|@ + a|=	b - b	b|@ - b + b|@	b|= - b + b|=	b|* - b + b|@ + b|=
1,2,3|=,4|=,3,4	1,2,3|=, 4|=,3,4	1,2,3=, 4|=,3,4	1,2,3|=, 4|=,3,4	1,2,3|=, 4|=,3,4		2(3|=,4|=)		
1(2),3(4)								

|@ move pointer state |= removes them from stack

<< this whole argument sequence is weak and needs to be reworked/expanded >>
<< this doesn't really explain why the a|@ conditional transition remains. The answer is |= and |* are tighed to the accept state, while |@ isn't. This comes out of the design decision to all null valued vars and λ|= transitions instead of taking the |= transition from the character of the variable. This is also the reason conditional accept states exist. >>

DFA State	a	a|@	a|=	b	b|@	b|=
1(2,3|=,4|=)	1(2,3|=,4|=)	2(3|=,4|=)	3(4)		2(3|=,4|=)	
1(2),3(4)	1(2,3|=,4|=) 3(4)	2(3|=,4|=)			2(3|=,4|=)	
2(3|=,4|=)		2(3|=,4|=)	3(4)		2(3|=,4|=)	
2,3(4)	3(4)	2(3|=,4|=)			2(3|=,4|=)	
3(4)	3(4)					

Table 13: Subset construction for a\@a**

Examining the pair of 2(3|=,4|=) and 2,3(4) we see a very similar situation, except that the two states handle 'b' as input as well has 'a's. If the two were merged ???

What, why is there conditional accept and |= off the same state.

is the split only based on accept states that are conditional

- no, it may result in alternate transitions
-

Example 9: a variable followed by a variable (\@\@)

Example 10: a variable or a variable (\@|\@)

Example 11: a variable or a variable offset so that match does not begin the same (\@|a\@)

looking for the longest match that is not anchored at the tail (all input consumed),

Two var example where same var has 1 variable is in accept and another isn't on the dfa state so that accept condition and consumption set are different.

Example?: A pattern matching a variable followed by the same variable (\@\@)

Example 4: A pattern matching the letter 'a' followed by a variable (a\@)

Example 5: A pattern matching any text followed by a variable (.*\@)

Example 5.1: A pattern matching an arbitrary number of “a”s followed by a variable (a*\@)

Example 6: A pattern matching

pattern with 2 variables
extend syntax to allow for multiple variables.

Section on Implementing extended DFAs with variable matching

Extending the Aho Cohsec DFA construction algorithm from the Dragon Book

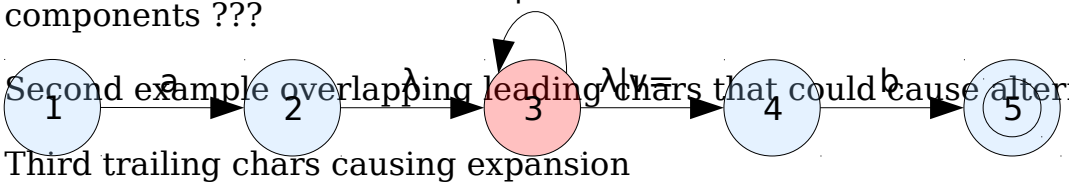
- followpos computes an NFA without eps transitions
  - model \@ transition after .*
  - carry |= conditional through, maybe store in two sets
  - at the end of followpos we have NFA with no lambda or conditional lambda transitions but we have conditional states and variable states that will lead to conditional transitions
- compute DFA from followpos
  - states marked as conditional add conditional transitions
  - conditional transitions stored as extra entries in table
  - split

??? Section on Implementation details

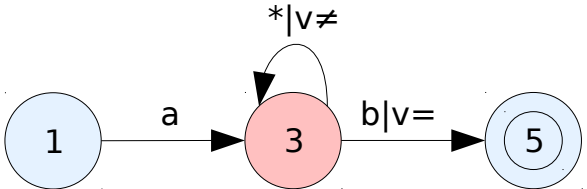
Backreferences and multivalued variables.

??? how to work this in
Beyond providing the special case ability to have efficient pattern matching with late bound variables this provides the back half of an implementation needed to deal with back references, which are a widely used feature in advanced pattern matching.

Modeling as an NFA
??? First Example – showing how to model it and put it together with other NFA components ???



Second example overlapping leading chars that could cause alternate evals
Third trailing chars causing expansion



fourth example, two alternating paths with the same variable slightly offset (multiple activations)

Fith example multiple variables

Converting to a DFA
- subset construction

modifying ahos algorithm

When conditional and unconditional transitions overlap
a dfa allows only a single transition per character

Encoding conditional transitions
- expanding the vector
- augmenting with additional information
-

Using a pattern to Limit state expansion
So far the variable has been matched using a customized state that matches the pattern used to match .* expressions. This however results in the same exponential state expansion that kleen closure expressions can have when constructing a DFA.

If the variable in question, has a limited set of values the .* pattern can be replaced with a more limiting pattern. This will result in dfa construction that is bounded by the characteristics of the selected subpattern which in most cases will be better than the bounds afforded by .* expressions.

Using variable variable sets to control state expansion
There is alternate way to control state expansion with variables. Instead of expanding the number of states used to track the current position and variables. The state can be partially carried in the set of variables being matchd against.

Extending to backreferences – multivalued variables + pattern group tracking

???? Rework earlier example ???

DFA
- state explosion
  - what causes it
    - counting constraints
    - rule overlap
    -
- use stored state instead of exponential states

- explosion of stored state
  - control with (nfa breadth, vs. depth first traversal, vs. hybrid)

NFA

HFA

- NFA decomposed into DFA segments

Integrating in variables

- basic simple example, no multi entry etc
  - nfa
  - dfa
- multi-entry example
  - nfa
  - dfa
- 2 var example to show combining
  - what of trailing?
- Encoding into next check/tables
  - as if extending the dfa to have more transitions
  -

Multivars

- single accept
- multiple accept
-

Dealing with overlapping vars and more than 1 permission set

Dealing with vars and minimization. Overlapping vars + more than 1 perm set + minimization

Dealing with variables and back references in a dfa

- dealing with a variable in an nfa
  - instantiate match on entry to variable state
  - 3 conditional transitions equivalent to extending the alphabet by 3* (==, !=, and == && != holding at the same time)
  - variables that can be null and lambda values
  - back tracking vs multithread approach
- converting the nfa to a dfa
- empty var – lambda pushes conditional into previous state
- multi-var/overlapping var example
- efficient handling on the consume action
- representation and compressing
- reducing the state expansion
  - variables effect on dfa expansion is the same as .*
  - associate the variable with a limiting pattern, each state within the pattern is a consuming state, for the variable
- extending aho algorithm to create dfas from expr trees

- variable relation to back reference
  - back reference is a dynamically defined variable that is created by matching
  - back references create a set of potential values for the state
  - back references need transition tagging to buildup back reference values
  - variable matching state (consuming state) must process all potential value in the value set

Hybrid Atomata

NFA implementations can be split into two broad categories, depth first traversal where potential matches are tried one at a time and backtracking approach is used to find the first or “best” match, and breadth first traversal where each potential set of transitions is tested simultaneously.

Each implementation has advantages and disadvantages …

depth first – least memory, variable time if doing first match, potentially slowest if doing best match
breadth first – larger memory foot print, constant time to match

DFA equiv to breadth first search, fast but has large memory over head (trade off time for space)

HFA – An NFA with sections that are dfas, try to have best properties of both. Fixed memory overhead known in advance

Implemention

Now that we have established how to construct a variable matching DFA we will consider an implementation.

Since we are using a single state to match against a string, the *|@ transition will need to operate on some memory to remember the matches current position. This can be done with either a pointer or index value into the variables string, which is incremented with each transition. The memory is set up by the transition into the state, but not by the *|@ transition that loops back into the state, and can be discarded when the state is left using either the *|= or λ|= transitions.

Appendix A – Thompson's construction

need to cover base of Thomposon's construction

Appendix B – The subset construction

The subset construction works on sets of states of an NFA. Each DFA state is created from a set (one or more states) from the NFA.

```
Subset construction
   let S, T be sets of NFA states
   work_queue := λ-closure({start state})
   while (work_queue is not empty)
      S := pop(work_queue)
      foreach input symbol c do
         T :=  λ-closure(move(S, c))
         if T is not in States
            insert T into States
            push T onto work_queue
         S.transition[c] := T
```

Move finds the set of NFA states that can be transitioned to from the DFA state S (which is a set of NFA states) for the input character c

```
move(S, c)
   let T be a set of NFA states
   let x be an NFA state
   foreach x in S do
      foreach NFA state t that c can transition to from x
         push(T, t)
   return T
```

Lambda closure computes the set of NFA states that can be transitioned to from the set of NFA states S using only lambda (null transitions)

```
λ-closure(S)
   let work_queue be a set of NFA states
   let t be an NFA state
   work_queue := S
   while work_queue is not empty
      s := pop(work_queue)
      foreach e
```

Appendix C – Anchored and Unanchored regular expressions

Anchored and unanchored matches in a DFA (appendix ?)

When converting a regular expression into a DFA extra states are required to handled unanchored expressions. An anchored expression has a straight conversion to an NFA using Thompson's construction as shown in Illustration 17.

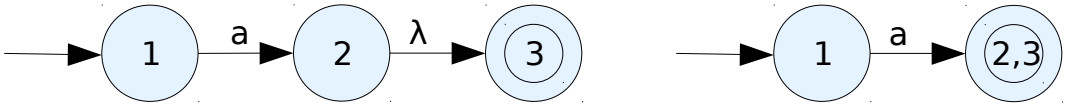


Illustration 17: NFA (left) and DFA (right) for the anchored regular expression ^a$

An unanchored expression requires modified start and possibly accept states, depending on implementation, as shown in Illustration 18. The start state loops on itself to consume leading input that does not match the expression, while the accept state can be made to loop on it self to consume all remaining input.

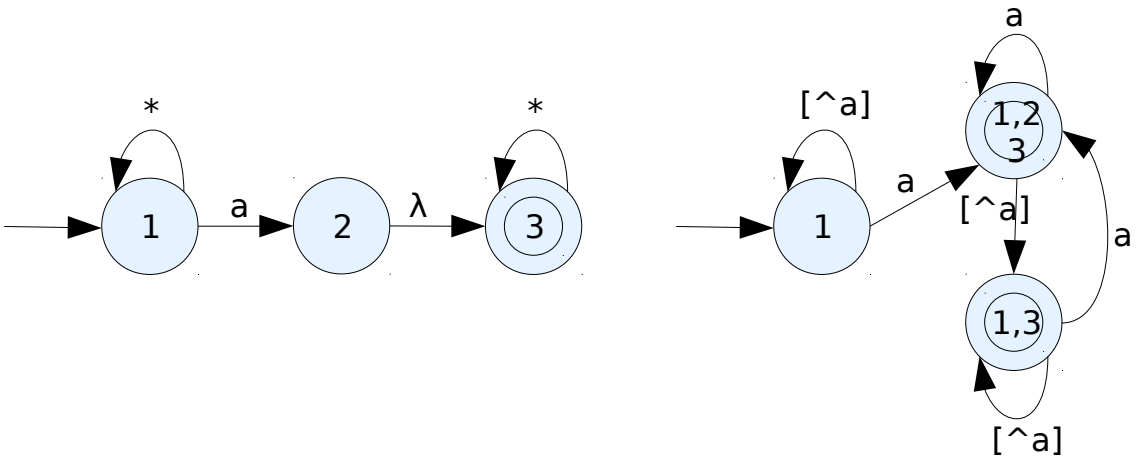


Illustration 18: NFA (left) and DFA (right) for the unanchored regular expression (a)

The modified accept state is not needed unless the DFA is to consume all input, as long as the accept condition is checked after each input character is consumed, matching can stop when the first accept state is encountered (see longest and shortest matches for more).

The modified start state is needed for an efficient match as restarting a match for every character of the input is inefficient and does not work for on-line matches where the input is available only once. However in setting up the modified start condition we lose the ability to determined when the actual match we are interested in starts. To be able to reliably determine where the start of a match input occurs the dfa needs to be extended as discussed in ????Tracking pattern grouping positions????.

????

Appendix D: Shortest and Longest match in a DFA

how to perform shortest match/longest matching

right anchor is a longest matching

References

foo