TDT4205 Compiler Construction

Recitation Lecture PS1

haavakro@stud.ntnu.no

Department of Computer Science

2023-01-20

Table of Contents

1 Regular expressions

2 Regex to NFA conversion

3 NFA to DFA conversion

4 DFA minimization

5 Help with C programming

Table of Contents

1 Regular expressions

2 Regex to NFA conversion

3 NFA to DFA conversion

4 DFA minimization

5 Help with C programming

Regular expressions

"A langauge for specifying regular languages."

$\mathcal{L}(\texttt{abc}) = \{abc\}$.

$\mathcal{L}(\texttt{ab?c}) = \{ac, abc\}$.

$\mathcal{L}(\texttt{ab*c}) = \{ac, abc, abbc, abbbc, \ldots\}$

Terminology

If a string $S \in \mathcal{L}(R)$, we say that the regular expression R **matches** the string S. We call S a **match**.

Regex operator precedence

- $\mathcal{L}(\texttt{a|b}) = \{a, b\}$
- $\mathcal{L}(\texttt{ab|cd}) = \{ab, cd\}$
- $\mathcal{L}(\texttt{a(b|c)d}) = \{abd, acd\}$
- $\mathcal{L}(\texttt{ab*}) = \{a, ab, abb, abb, abbb, \cdots\}$
- $\mathcal{L}(\texttt{ab+}) = \{ab, abb, abbb, abbbb, \cdots\}$
- $\mathcal{L}(\texttt{ab?}) = \{a, ab\}$
- $\mathcal{L}(\texttt{(ab)?}) = \{ab, \epsilon\}$
- $\mathcal{L}(\texttt{a|b+}) = \{a, b, bb, bbb, bbbb, bbbbb, \cdots\}$
- $\mathcal{L}(\texttt{(a|b)+}) = \{a, b, aa, ab, bb, ba, aaa, aab, aba, abb, \cdots\}$

Turning a description into regex

Let $\mathcal{L}$ be all strings over the alphabet $\{a, b\}$, with exactly two 'a's.

Turning a description into regex

Let $\mathcal{L}$ be all strings over the alphabet $\{a, b\}$, with exactly two 'a's.

It can start with a, but also b.
We can never have `a?`, `a+` or `a*`, since we would lose count of 'a's.
We can get the two 'a's with any number of `b` in between.

Turning a description into regex

Let $\mathcal{L}$ be all strings over the alphabet $\{a, b\}$, with exactly two 'a's.

It can start with a, but also b.
We can never have `a?`, `a+` or `a*`, since we would lose count of `'a`'s.
We can get the two `'a`'s with any number of `b` in between.

$R =$ `b*ab*ab*`

(Other posibilities can also work)
PS1 contains a problem like this, with a link to a website with tests.

Table of Contents

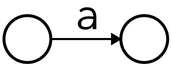
1. Regular expressions
2. Regex to NFA conversion
3. NFA to DFA conversion
4. DFA minimization
5. Help with C programming

Turning a regex into an NFA

Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.

Turning a regex into an NFA

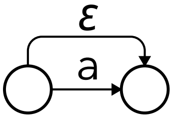
Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.



a

Turning a regex into an NFA

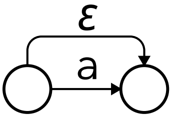
Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.



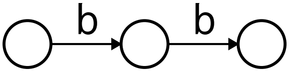
a?

Turning a regex into an NFA

Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.



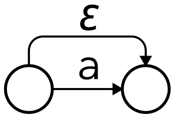
a?



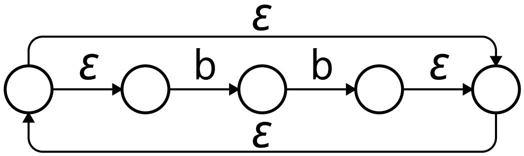
bb

Turning a regex into an NFA

Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.



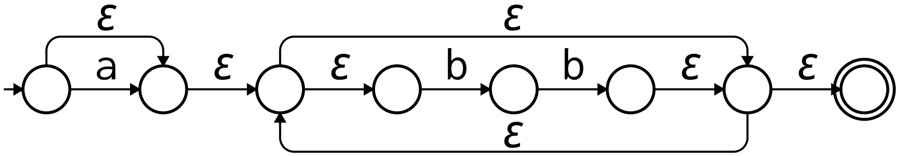
a?



(bb)∗

Turning a regex into an NFA

Known as Thompson's construction.
We will use `a?(bb)*` as our example regex.



a?(bb)*

Using some human intuition when making our NFA

We still use `a?(bb)*` as our example regex.

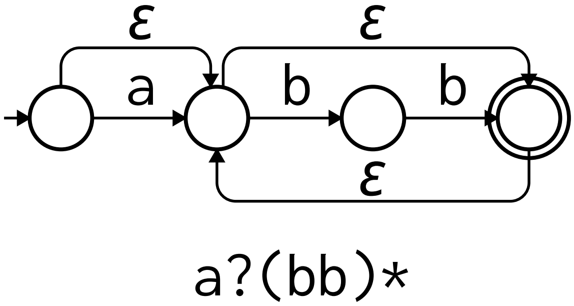


Table of Contents

1 Regular expressions

2 Regex to NFA conversion

3 NFA to DFA conversion

4 DFA minimization

5 Help with C programming

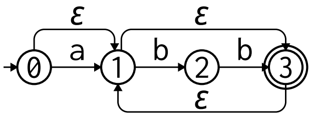
NFA to DFA

Also known as Subset construction.

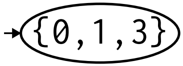
$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "a"] \arrow[r, bend left=60, "\varepsilon"] & 1 \arrow[r, "b"] \arrow[rr, bend left=40, "\varepsilon"] & 2 \arrow[r, "b"] & 3 \arrow[ll, bend left=40, "\varepsilon"]
\end{tikzcd}$$

a?(bb)*

NFA to DFA

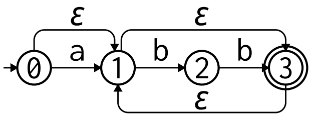


a?(bb)*

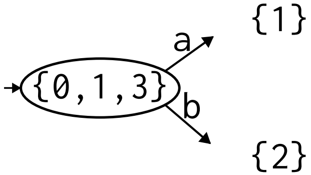


We need to know every state we can get to without consuming any input:
$\epsilon\text{closure}(\{0\}) = \{0, 1, 3\}$

NFA to DFA

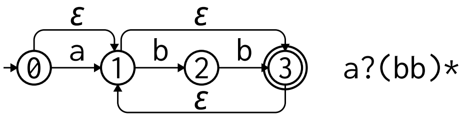


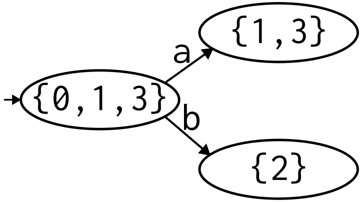
a?(bb)*



Now we need to know where we can end up, for each input
move($\{0, 1, 3\}, a$) $= \{1\}$
move($\{0, 1, 3\}, b$) $= \{2\}$

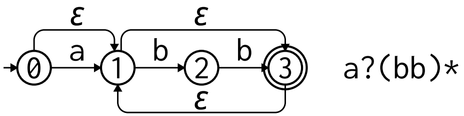
NFA to DFA

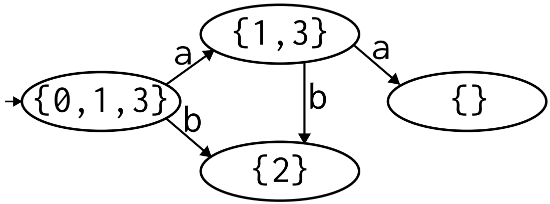




We must also include states accessible throuhg ϵ:
ϵclosure$(\{1\}) = \{1, 3\}$
ϵclosure$(\{2\}) = \{2\}$

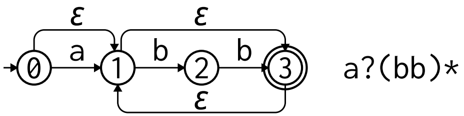
NFA to DFA

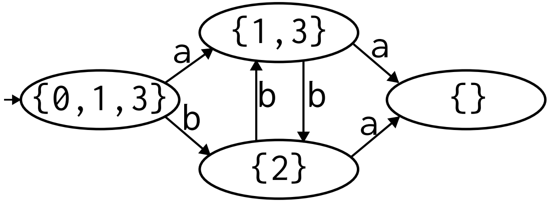




Now we do the same for $\{1, 3\}$:
$\epsilon\text{closure}(\text{move}(\{1, 3\}, a)) = \{\}$
$\epsilon\text{closure}(\text{move}(\{1, 3\}, b)) = \{2\}$

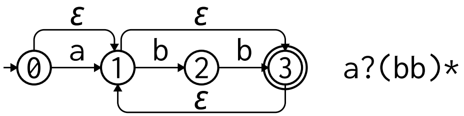
NFA to DFA

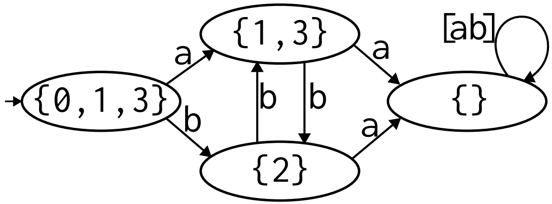




Now we do the same for $\{2\}$:
$\epsilon\text{closure}(\text{move}(\{2\}, a)) = \{\}$
$\epsilon\text{closure}(\text{move}(\{2\}, b)) = \{1, 3\}$

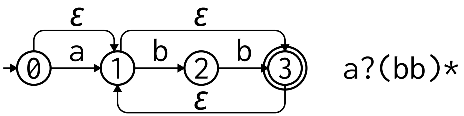
NFA to DFA

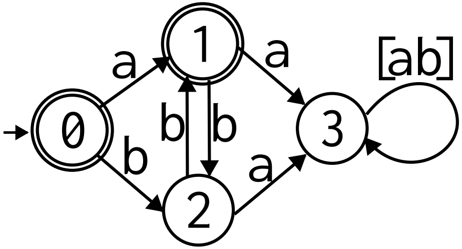




Since every state must have exactly one edge per input, we add a loop to the “all VMs are dead” state.

NFA to DFA





NFA subsets that contain accepting states, become accepting DFA states. We finally renumber the states, and forget that the DFA originally came from an NFA.

Table of Contents

1 Regular expressions

2 Regex to NFA conversion

3 NFA to DFA conversion

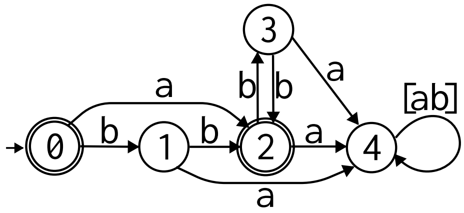
4 DFA minimization

5 Help with C programming

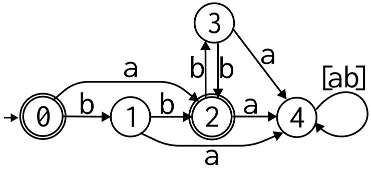
Minimal DFAs

In the previous slides we actually got a minimal DFA from the Subset construction, but that is not always the case.

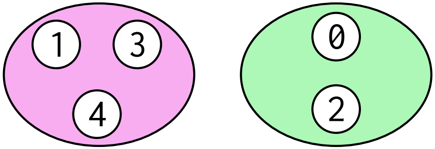
The following DFA accpets the same langauge `a?(bb)*`, but uses one more state.



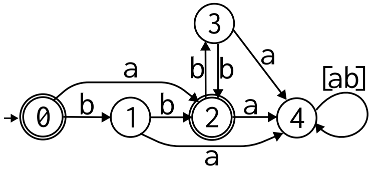
Moore's algorithm



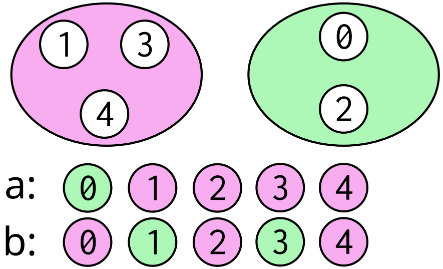
We start by grouping the states into accepting and non-accepting.



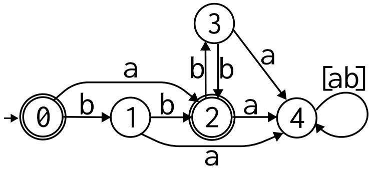
Moore's algorithm



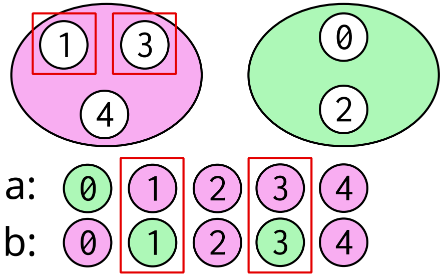
For each possible input, note which group you end up in.



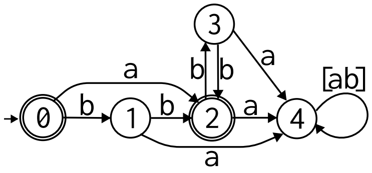
Moore's algorithm



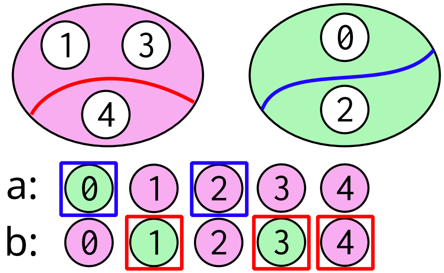
Note how states 1 and 3 are identical in all 3 partitionings.



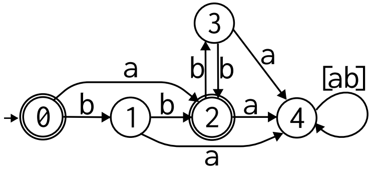
Moore's algorithm



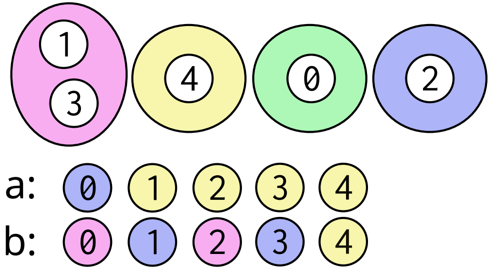
We actually care more about states that have differences, since that is what forces us to split up groups.



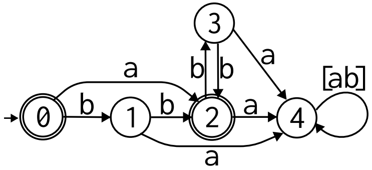
Moore's algorithm



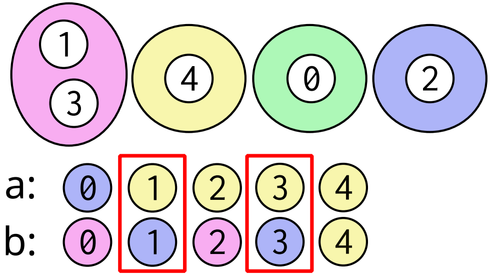
We split up the groups along the lines. This gives us more groups and more colors.



Moore's algorithm

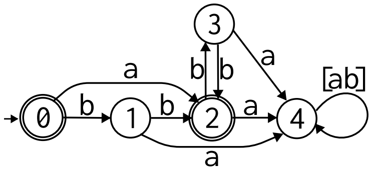


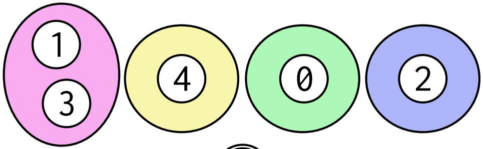
We must keep going until all groups only contain "identical" states.
In our case, only the pink group has more than one state, and they are identical.

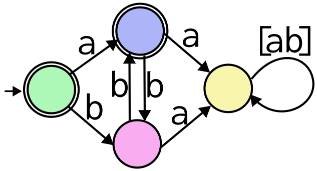


Finishing the DFA minimizaton

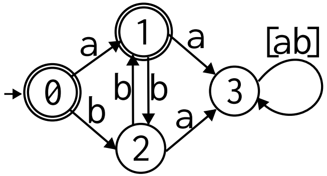
We can now create a DFA consisting of the **groups** instead. Any edge from the original DFA that crosses groups, gets added to the new DFA.







Finishing the DFA minimizaton



We now see that our original DFA was minimal.
Now that we are done, we can re-number our states (instead of colors).

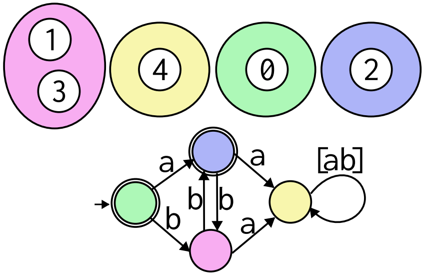


Table of Contents

1 Regular expressions

2 Regex to NFA conversion

3 NFA to DFA conversion

4 DFA minimization

5 Help with C programming

Surviving and thriving in C

C lets you shoot yourself in the foot.

Accessing memory you didn't intend to is easy.

The compiler will try to help with some things,
but C isn't even strongly typed.
(If you want to treat a `char*` as a `double`, you can)

I will focus on *runtime* debugging.

What happens when I do something bad?

If you access memory that your process doesn't have, you will get a `Segmentation Fault`. The OS kills you.

If you access `array[N]` in an array of length `N`, your process probably owns that memory too. No crash.

If you read "out of bounds", you get garbage.
If you write there, you can mess things up. The program might crash down the line.

Debugging crashes

If you get a `Segmentation Fault`, try compiling your program with `-g` to include debug info in the executable.

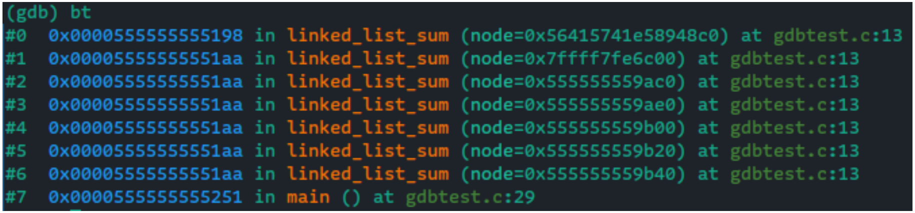
Run it again using `gdb <program>`

Inside gdb, start the program with `run <args here>`

Once the program crashes, type `bt` to get a **backtrace**, aka the callstack leading up to the crash.

Example backtrace

Here I was trying to sum up a linked list with a recursive function.
It gave a `Segmentation Fault`.



```
(gdb) bt
#0  0x0000555555555198 in linked_list_sum (node=0x56415741e58948c0) at gdbtest.c:13
#1  0x00005555555551aa in linked_list_sum (node=0x7ffff7fe6c00) at gdbtest.c:13
#2  0x00005555555551aa in linked_list_sum (node=0x555555559ac0) at gdbtest.c:13
#3  0x00005555555551aa in linked_list_sum (node=0x555555559ae0) at gdbtest.c:13
#4  0x00005555555551aa in linked_list_sum (node=0x555555559b00) at gdbtest.c:13
#5  0x00005555555551aa in linked_list_sum (node=0x555555559b20) at gdbtest.c:13
#6  0x00005555555551aa in linked_list_sum (node=0x555555559b40) at gdbtest.c:13
#7  0x0000555555555251 in main () at gdbtest.c:29
```

The first 5 nodes have very similar addresses, all from `malloc()`.
Then we get a very different address, something that is not a node. The last list element didn't set `next` to `NULL`!

Extra gdb tips

- If your program is hanging forever, run in `gdb`, and hit `Ctrl-Z` to pause your program. Now you can use `bt` to print where in the code the program is stuck.
- You can print the value of variables using `p <variable name>`, both on the local stack, and global variables.
- You can even do things like `p nodes[x+5]->value`, if `nodes` and `x` are defined.

More sneaky memory bugs

You might have a memory bug if:

- Your program produces different results each time
- Your program crashes in library code
- You have values that make no sense
- Your program crashes, but not in the debugger!

Valgrind

An emulator that keeps track of memory. Will tell you about:

- Accessing out of bounds of arrays
- Accessing out of bounds of a `malloc`
- Accessing uninitialized memory
- Accessing memory you have `free()`d
- Accessing variables from scopes that no longer exist
- Calling `free()` on the same allocation twice
- Forgetting to `free()` some memory you `malloc()`ed.

Run using `valgrind <program> <args>`.
The downside? It runs slow.

Some final C compiler flags

Remember to include `-g` to get debug info.

Enable warnings using `-Wall`, and maybe even `-Wextra`.

Enable address sanitazion and undefined behaviour sanitazion. It's not as thorough as valgrind, but it runs much faster.

```
    -fsanitize=address,undefined
```

You can add extra parameters by modifying `CFLAGS` in the Makefile.