Formális nyelvek és fordítóprogramok

5–6. Véges automaták és reguláris nyelvek III.

Bodó Zalán

Babeș–Bolyai Tudományegyetem
Matematika és Informatika Kar
Magyar Matematika és Informatika Intézet

Reguláris kifejezések

Def. Reguláris kifejezés (*regex/regexp*)

A Σ ábécé feletti regex-et a következőképpen értelmezzük:

- $\emptyset$ reguláris kifejezés, mely az üres nyelvet jelöli.
- ε reguláris kifejezés, mely a $\{\varepsilon\}$ nyelvet jelöli.
- Ha $a \in \Sigma$, akkor a reguláris kifejezés is egyben, és az $\{a\}$ nyelvet jelöli.
- Ha x és y reguláris kifejezések, melyek az X és Y nyelveket jelölik, akkor $(x + y)$, (xy) és $(x*)$ is reguláris kifejezések, melyek rendre az $X \cup Y$, XY és $X*$ nyelveket jelölik.

Bármely Σ reguláris kifejezés csakis a fenti szabályok véges alkalmazásával állítható elő.

1. Példa

Példák reguláris kifejezésekre:

1. $R_1 = a*b(a + b)$, $L(R_1) = \{a^n ba, a^n bb \mid n \geq 0\}$
2. $R_2 = 0*01*10$, $L(R_2) = \{0^n 1^m 0 \mid n \geq 1, m \geq 1\}$
3. $R_3 = 0(0 + 1)(0 + 1)*$, $L(R_3) =$ $\{0\text{-val kezdődő}, \{0, 1\} \text{ felett értelmezett, legalább 2 hosszúságú szavak}\}$

Reguláris kifejezések tulajdonságai

$$
\begin{aligned}
x + y &\equiv y + x \\
(x + y) + z &\equiv x + (y + z) \\
(xy)z &\equiv x(yz) \\
(x + y)z &\equiv xz + yz \\
x(y + z) &\equiv xy + xz \\
(x + y)* \equiv (x* + y)* &\equiv (x + y*)* \equiv (x* + y*)* \\
(x + y)* &\equiv (x*y*)* \\
(x*)* &\equiv x* \\
x*x &\equiv xx* \\
xx* + \varepsilon &\equiv x*
\end{aligned}
$$

Def. Regex-ek ekvivalenciája

Két reguláris kifejezés ekvivalens, ha ugyanazokat a nyelveket jelölik, vagyis $R_1 \equiv R_2$, ha $L(R_1) = L(R_2)$.

Tétel (Kleene)

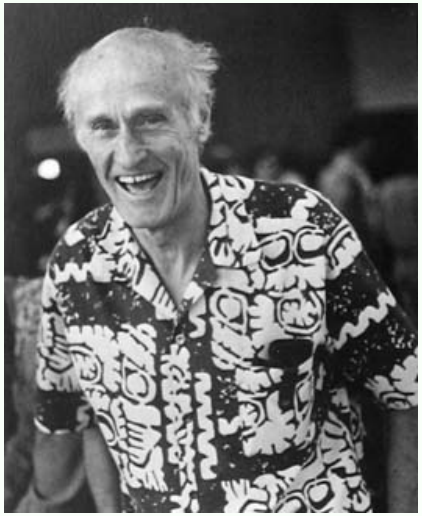
Az $L \subseteq \Sigma^*$ nyelv pontosan akkor reguláris, ha létezik olyan Σ feletti reguláris kifejezés, amely éppen L-et jelöli.

Bizonyítás.

1. Ha x regex, akkor L reguláris nyelv.
   Ha $x = \emptyset$, $x = \varepsilon$, $x = a, \forall a \in \Sigma$, akkor rendre $L = \emptyset$, $L = \{\varepsilon\}$, $L = \{a\}$. Mivel L mindhárom esetben véges, ezért reguláris.
   A $+$, a konkatenáció és a $*$ művelet esetén tudjuk, hogy a reguláris nyelvek osztálya zárt az egyesítésre, a konkatenációra (szorzatra) és az iterációra nézve, ezért ezen műveletekkel is reguláris nyelveket kapunk.*
2. Ha L reguláris nyelv, akkor hozzárendelhető egy reguláris kifejezés.
   Ez a része konstruktív: meg fogunk adni egy algoritmust, amely ezt megvalósítja (annak helyességét viszont nem fogjuk bizonyítani).
   Megjegyzés: Kleene nem ezzel a módszerrel bizonyította.

*Megjegyzendő: Minden Chomsky-féle nyelvosztály zárt a reguláris műveletekre nézve.

Stephen Cole Kleene



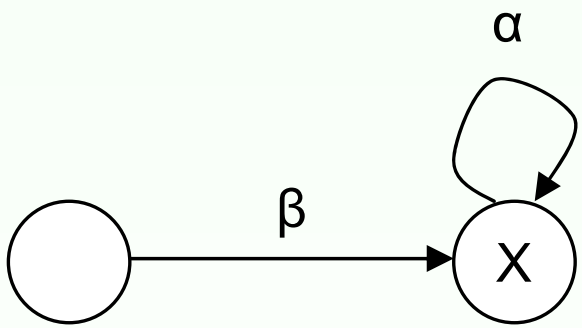
Stephen Cole Kleene (1909–1994), amerikai matematikus, az elméleti informatika (*theoretical computer science*) egyik atyja. A kiszámíthatóságelmélet mint tudományág megteremtője. A reguláris kifejezések bevezetője.
Róla nevezték el az iteráció műveletet ($*$, Kleene-csillag) – ezen kívül algebrát, tételeket és sok más dolgot is.

Egyik híres műve:
S. C. Kleene, *Representation of events in nerve nets and finite automata*, In: C. Shannon and J. McCarthy, (eds.) Automata Studies, Princeton University Press, NJ, 1956, pp. 3–41.

Regex hozzárendelése véges automatához

Formális egyenletek módszere:

- az állapotokhoz változókat rendelünk
- a változókkal egyenleteket írunk fel az automata állapotátmeneteinek megfelelően
  - az egyenletek lineárisak lesznek a változókban
  - minden egyenlet felírható $X = X\alpha + \beta$ vagy $X = X\alpha$ alakban, ahol α és β nem tartalmazza az X változót
  - az $X = X\alpha + \beta$ megoldása: $X = \beta\alpha*$, ami könnyen leolvasható az alábbi ábráról



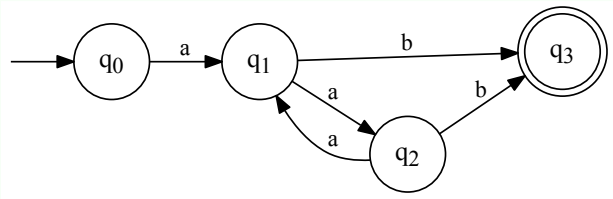
- megoldjuk az egyenletet a végállapotoknak megfelelő változókra

ALG 1 Véges automata $\rightarrow$ regex

1: Kizárjuk az elérhetetlen és nemproduktív állapotokat.
2: Minden állapothoz egy változót rendelünk.
3: **for** minden X változóra **do**
4: **for** minden (Y, a, X) átmenetre **do**
5: Hozzáadjuk az egyenlet jobb oldalához az Ya kifejezést.
6: **end for**
7: **if** X kezdőállapot **then**
8: Hozzáadjuk az egyenlet jobb oldalához az ε-t.
9: **end if**
10: **end for**
11: Megoldjuk az egyenletrendszert a végállapotokra.
12: Több végállapot esetén „összeadjuk” $(+)$ a kapott reguláris kifejezéseket.

2. Példa

Rendeljünk az alábbi automatához egy reguláris kifejezést.



A következő változókat rendeljük az állapotokhoz: $A \to q_0$, $B \to q_1$, $C \to q_2$, $D \to q_3$. Ekkor felírhatjuk a következő egyenleteket:

$$\begin{aligned} A &= \varepsilon \\ B &= Aa + Ca \\ C &= Ba \\ D &= Bb + Cb \end{aligned}$$

B-re kapjuk, hogy $B = a + Ca$; innen C-re kapjuk, hogy $C = aa + Caa$, amit megoldva kapjuk, hogy $C = aa(aa)*$. Visszahelyettesítve B-be kapjuk, hogy $B = a + aa(aa)*a$. Innen $D = ab + aa(aa)*ab + aa(aa)*b$.

Ez kissé bonyolult, próbáljunk egyszerűsíteni. Kiemelhetünk balról egy a-t, míg jobbról egy b-t:

$$D = a(\varepsilon + a(aa)*a + a(aa)*)b$$

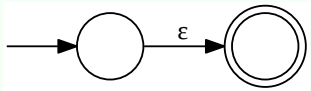
Ezt még tovább egyszerűsíthetjük a következőképpen: vegyük észre, hogy az $a(aa)*a$ a páros számú a-kat ismeri fel $(2, 4, 6, \ldots)$, míg az $a(aa)*$ a páratlan számúakat $(1, 3, 5, \ldots)$, végül az ε a 0-szor előfordulókat. Így a külső zárójelben levő kifejezés felírható az egyszerű $a*$ alakban, ahonnan D-re kapjuk, hogy

$$\boxed{D = aa*b}$$

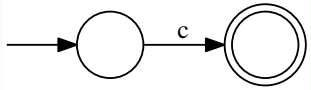
Véges automata hozzárendelése regex-hez

- Az algoritmus Thompson nevéhez fűződik (1968)
- Definiáljuk a következő **kanonikus automatákat**, melyeket felhasználunk az automatánk megépítéséhez (5 db.):

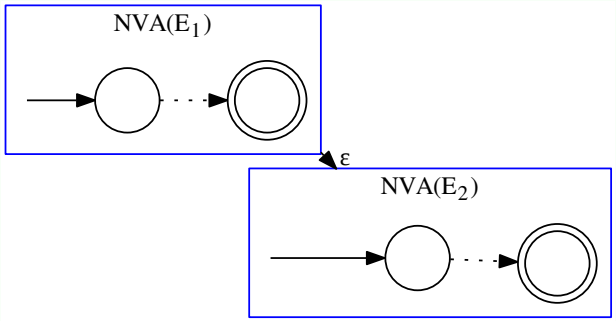
ε



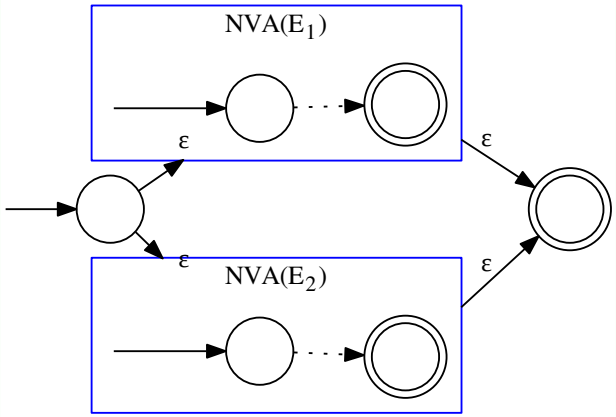
$c \in \Sigma$



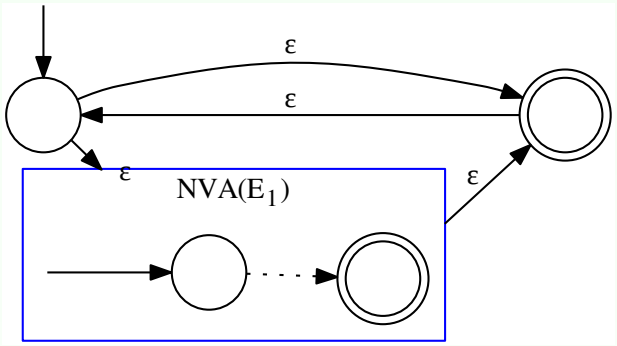
konkatenálás, $E_1 E_2$



vagy, $E_1 + E_2$



Kleene-csillag, E_1*



Ken Thompson

Ken Thompson és Dennis Ritchie

Kenneth Lane Thompson (1943–), amerikai informatikus. Dennis Ritchie-vel (1941–2011) a Unix operációs rendszer atyjai. Ő írta a B nyelvet, melyet a C nyelv elődjének tekintünk. Az ő nevéhez fűződik a Plan 9 operációs rendszer kifejlesztése és az UTF-8 kódolási rendszer megalkotása. Jelenleg a Google-nél dolgozik, a Go programozási nyelv egyik fő fejlesztője.

Egyik híres műve:
K. Thompson, D. M. Ritchie, *Unix Programmer's Manual*, Sixth, Bell Laboratories, 1975.

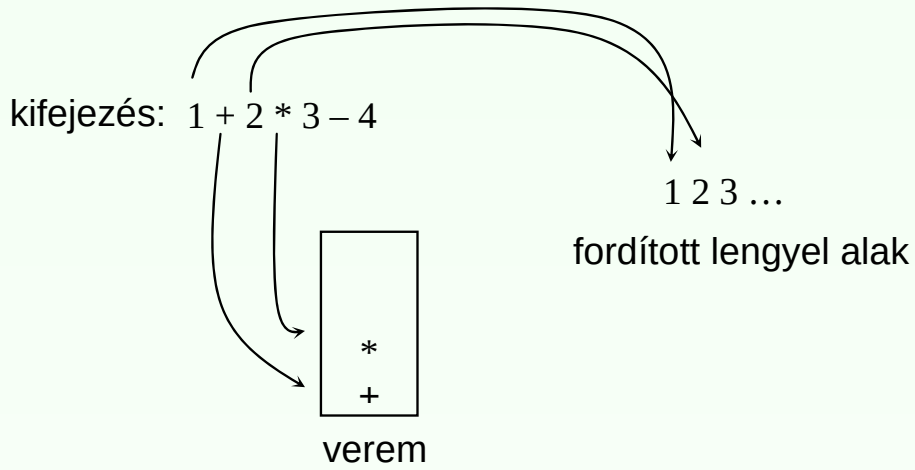
Az algoritmus lépései:

1. Átalakítjuk a reguláris kifejezést fordított lengyel alakba.
2. Az egyszerű aritmetikai kifejezésekhez hasonlóan „kiértékeljük” a reguláris kifejezést, amely itt nem jelent mást, mint hogy megépítjük a neki megfelelő Thompson-automatát. (Ez a „kiértékelés” nem a tulajdonképpeni kiértékelés.)

ALG 2 Aritmetikai kifejezések fordított lengyel alakja

```
while kifejezés végére érünk do
    if operandus then
        Beírjuk a FLA lista végére
    else if operátor then
        if prioritása kisebb mint a verem tetején levő operátornak then
            while prioritása kisebb do
                Kivesszük a veremből és beírjuk a FLA lista végére
            end while
        else
            Beírjuk a verem tetejére
        end if
    else if „)” then
        Beírjuk az FLA lista végére
    else
        Kivesszük az elemeket a bezáró „)”-ig és beírjuk az FLA lista végére (kivéve a zárójeleket)
    end if
end while
```

A fordított lengyel alakba való felírás grafikus vázlata:



ALG 3 Aritmetikai FLA kiértékelése

```
1: while FLA lista vége do
2:     if operandus then
3:         Beírjuk a verembe
4:     else
5:         Kivesszük a felső 2 elemet
6:         Végrehajtjuk a műveletet
7:         Visszatesszük az eredményt a verembe
8:     end if
9: end while
```

3. Példa

Legyen az aritmetikai kifejezésünk $1 + 2 * (3 - 2)$.
A műveletek prioritásai: $1 = p(+) < p(-) = 2$, $p(*) = p(/) = 3$ (vagy $3 = p(*) < p(/) = 4$).
FLA: 1, V: $\emptyset \rightarrow$ FLA: 1, V: $+ \rightarrow$ FLA: 12, V: $+ \rightarrow$ FLA: 12, V: $+* \rightarrow$ FLA: 12, V: $+ * ($ $\rightarrow$ FLA: 123, V: $+ * ($ $\rightarrow$ FLA: 123, V: $+ * (-$ $\rightarrow$ FLA: 1232, V: $+ * (-$ $\rightarrow$ FLA: 1232, V: $+ * (-)$ $\rightarrow$ FLA: $1232 - *+$
Kiértékelés: V: $1 \rightarrow 12 \rightarrow 123 \rightarrow 1232 \rightarrow 121 \rightarrow 12 \rightarrow 3$

Reguláris kifejezés fordított lengyel alakja:

- hasonló az aritmetikai kifejezésekhez
- itt most 2 bináris operátorunk van, a konkatenáció (ezt .-tal fogjuk jelölni) és a vagy művelet (+)
- 1 db unáris művelet, a Kleene-csillag (*)
- prioritások: $p(.) > p(+)$
- az unáris műveletnek nincs prioritása, úgy viselkedik mint az operandusok: beírjuk az FLA listába
- a zárójelek ugyanúgy működnek

Kérdés: Miért szükséges a helyes működéshez az előző példában látott alábbi egyenlőtlenség?

$$p(+) < p(-)$$

Feladat: Végezzük el az

$$1 - 2 + 3$$

kiértékelését

(a) $p(+) = p(-)$ esetén;

(b) $p(+) < p(-)$ esetén.

4. Példa

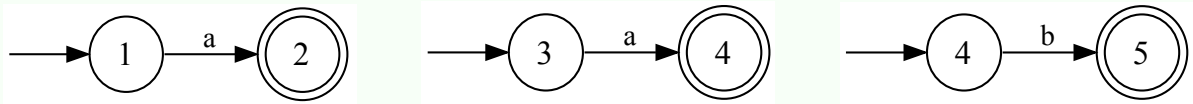
Legyen a reguláris kifejezésünk: $a(a+b)*c(c+d)$
Jelölve a konkatenálásokat: $a.(a+b)*.c.(c+d)$
A regex lengyel alakban: $aab+*ccd+...$

Véges automata megépítése:

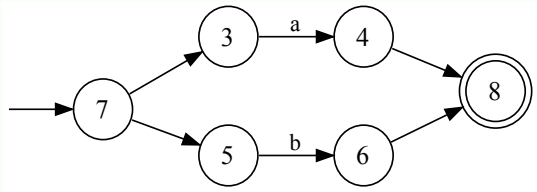
- hasonló az aritmetikai kifejezés kiértékeléséhez
- a verembe most automaták fognak kerülni
- használva a Thompson-féle **kanonikus** automatákat, műveletek esetén elvégezzük azokat:
  - bináris művelet esetén (. és +) kivesszük a felső 2 automatát a veremből, elvégezzük a műveletet, az eredményt visszatesszük a verembe
  - unáris művelet esetén (*) csak a legfelső automatát vesszük ki, elvégezzük a műveletet, és hasonlóan visszatesszük az eredményt

5. Példa

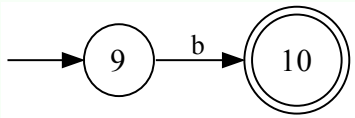
Legyen a reguláris kifejezésünk: $a(a+b)b*a$
Jelölve a konkatenációt: $a.(a+b).b*.a$
Fordított lengyel alakban: $aab+b*a...$
Az a, a és b szimbólumokhoz tartozó automaták:



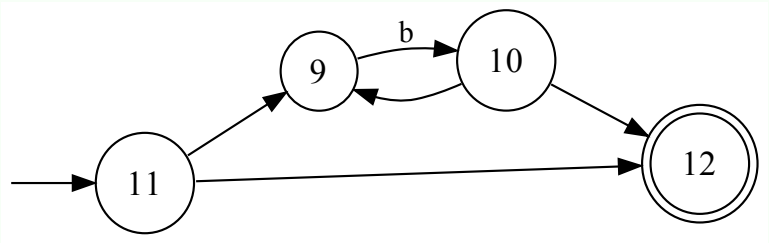
Az utolsó kettőre alkalmazva a $+$ műveletet, a következő automatát kapjuk (az szimbólum nélküli átmenetek ε-átmeneteket jelölnek):



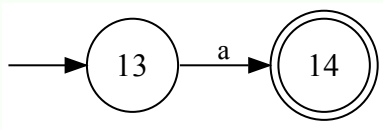
A következő szimbólum megint b, tehát:



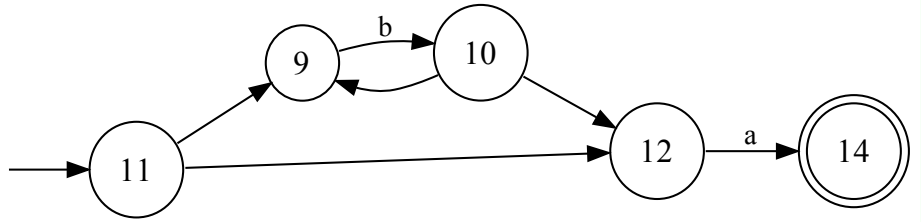
Most a $*$ következik, amely egy unáris operátor, ezért az utolsó automatát az szabálynak megfelelően átalakítjuk, azaz



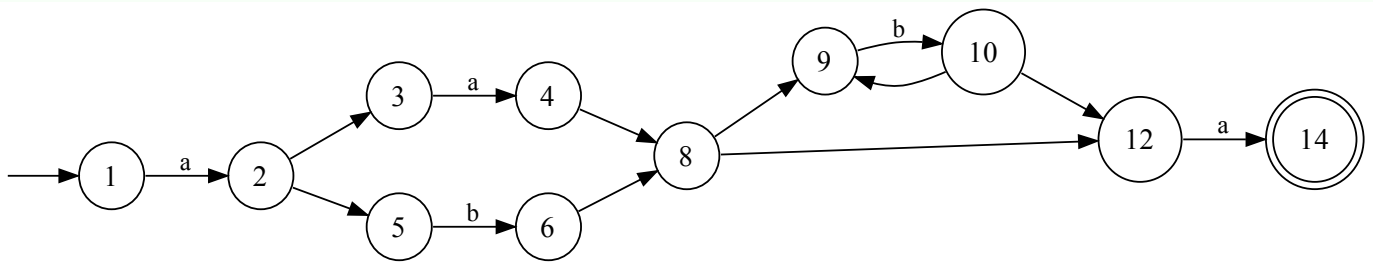
A következő szimbólum *a*:



Konkatenálás következik, tehát:



Ezután következik még két konkatenálás, melyek eredményét összevonva az alábbi automatát kapjuk:



Reguláris kifejezések kiértékelése

- reguláris kifejezések – legtöbbször keresésre használjuk: *grep*, *sed*, valamely programozási nyelv (*Java*, *Perl*, ...) reguláris kifejezései, szövegszerkesztők reguláris kifejezései (*Sublime*, *Notepad++*, *Textpad*, *vim*, *emacs*, ...), ...
- itt: kiértékelés = keresés
- feladat: megkeresni egy szövegben azokat a részeket, melyek illeszkednek egy adott reguláris kifejezésre (*fordítva* is lehet mondani, hogy a reguláris kif. illeszkedik a szövegre)
- megvalósítás:
  - ε-átmenetek kiszűrése
  - NDVA $\rightarrow$ DVA
  - alkalmazás a szövegre – meg kell oldani, hogy minden betűnél újrakezdjük a keresést
  - mindig jelezni kell, ha végállapotba értünk

Gyors kiértékelési módszerek

- gyors szöveges keresések
- R. Baeza-Yates és G. Gonnet
- a bit-párhuzamosságra (angolul *bit-parallelism*) alapulnak: egy szóban tárolunk sok kicsi értéket, és ezeket bitaritmetikával egyidőben módosítjuk, változtatjuk
- a reguláris kifejezést felismerő automatákat bitsorozatokban tároljuk
- két algoritmus: **Shift-And** és **Shift-Or**

Def.

Egy bitsorozatot másképpen bitmaszknak is nevezünk. A bitmaszkokat jobbról balra olvassuk, vagyis az első bit a legjobboldali. Legyen M_1 és M_2 két azonos hosszúságú bitmaszk. Ekkor a következő műveleteket és jelöléseket definiáljuk:

- $M_1|M_2$ – bitenkénti „vagy”-ot jelent (OR)
- $M_1 \& M_2$ – bitenkénti „és” (AND)
- $M_1 \wedge M_2$ – bitenkénti XOR (kizáró vagy)

- $\sim M_1$ – bitenkénti tagadás (NOT)
- $M1 \ll k$ – M1 k darab bittel balra való eltolása
- $M1 \gg k$ – M1 k darab bittel jobbra való eltolását jelenti.

Feladat: Adott egy T szöveg és egy E kifejezés, és meg kell keresnünk E előfordulásait T-ben. T és E egyszerű szövegek. Definiáljuk a következő két bitmaszkot:

- B, $|B| = m$; ez lesz a keresett kifejezés és a keresés közötti egyetlen kapcsolat; $B[c]_i = 1$ ($B[c] = c_m c_{m-1} \dots c_1$), ha $E_i = c$ ($E = e_1 e_2 \dots e_m$), különben $B[c]_i = 0$;
- D, $|D| = m$; ez a bitmaszk modellezi a nemdeterminisztikus véges automatát.

ALG 4 Shift-And

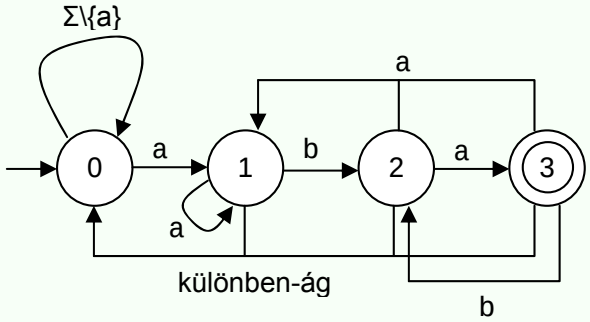
1: $D = 0^m; i = 0; c = T_i$
2: $D = ((D \ll 1)|0^{(m-1)}1)\&B[c]$
3: $i = i + 1$
4: **if** D legutolsó bitje 1 **then**
5: TALÁLAT, kiír$(i - m,\ i)$;
6: **end if**
7: $c = T_i$
8: **if** $(i! = n)$ **then**
9: Ugrás a 2. lépésre
10: **end if**
11: STOP

- a D és B bitmaszkokkal az E kifejezés által leírt automatát modellezzük
- ha D i-edik bitje 1, akkor az i-edik állapot aktív (jelenleg abban az állapotban vagyunk), ellenkező esetben inaktív

- érdekessége és egyben előnye: egyszerre több állapot is lehet aktív $\Rightarrow$ a nemdeterminisztikus keresését nem visszalépéssel, hanem párhuzamossággal valósítjuk meg
- legfontosabb lépés a 2.:
  - feltételezzük, hogy az i-edik állapot aktív
  - ekkor megpróbálunk előrelépni a köv. állapotba (eltolás)
  - a $0^{(m-1)}1$-gyel való bitenkénti **vagy** azért szükséges, hogy mindig elölről tudjuk kezdeni a keresést
  - a $B[c]$-vel való bitenkénti **és** pedig azért, mert azzal „biztosítjuk", hogy ez a lépés tényleg lehetséges

6. Példa

Legyen $T = ababa$ és $E = aba$. Cél: megtalálni E összes előfordulását T-ben.
Az automata, melyet a Shift-And algoritmus az E kifejezéshez (sztringhez) „épít”:



$T = ababa$, $E = aba$, $B[a] = 101$, $B[b] = 010$

$T = \underline{a}baba$
$D = (000|001)\&101 = 001\&101 = 001$
$T = a\underline{b}aba$
$D = (010|001)\&010 = 011\&010 = 010$
$T = ab\underline{a}ba$
$D = (100|001)\&101 = 101\&101 = 101 \Rightarrow$ találat, pozíció: $(0, 2)$

▷

$T = aba\underline{b}a$
$D = (010|001)\&010 = 011\&010 = 010$
$T = abab\underline{a}$
$D = (100|001)\&101 = 101\&101 = 101 \Rightarrow$ találat, pozíció: $(2, 4)$

Shift-Or

- megfordítjuk a biteket: 0 jelöli az aktív, 1 az inaktív állapotot
- emiatt a művelet is változik: bitenkénti **és** helyett **vagy** lesz
- eltolásnál (automatikusan) 0 kerül a bitsorozat elejére
- így megspórolunk egy bitműveletet (ez regex-ek esetén nem fog számítani)

ALG 5 Shift-Or

1: $D = 1^m; i = 0; c = T_i$
2: $D = (D \ll 1) | B[c]$
3: $i = i + 1$
4: **if** D legutolsó bitje 0 **then**
5: TALÁLAT, kiír$(i - m,\ i)$;
6: **end if**
7: $c = T_i$
8: **if** $(i! = n)$ **then**
9: Ugrás a 2. lépésre
10: **end if**
11: STOP

---

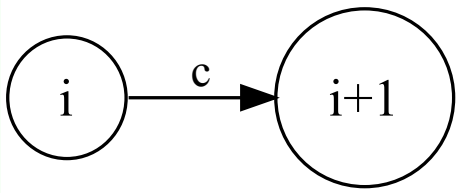
7. Példa

Legyen $T = ababa$ és $E = aba$. Keressük meg E összes előfordulását T-ben a Shift-Or algoritmussal.
$T = ababa$, $E = aba$, $B[a] = 010$, $B[b] = 101$

$T = \underline{a}baba$
$D = 110|010 = 110$
$T = a\underline{b}aba$
$D = 100|101 = 101$
$T = ab\underline{a}ba$
$D = 010|010 = 010 \Rightarrow$ találat, pozíció: $(0, 2)$
$T = aba\underline{b}a$
$D = 100|101 = 101$
$T = abab\underline{a}$
$D = 010|010 = 010 \Rightarrow$ találat, pozíció: $(2, 4)$

Reguláris kifejezések és a Shift-And

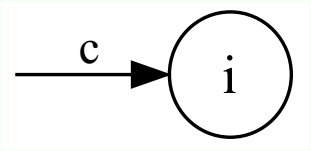
- Thompson-automata megépítése
- az eltoláshoz minden nem ε-átmenetre teljesülnie kell, hogy az állapotok sorszámai egymást követő természetes számok, vagyis



- a következőképpen járunk el:
  - ha c karakter, akkor megépítjük a kanonikus automatát a fenti módon, ahol i a legkisebb még nem használt természetes szám
  - konkatenációnál nem „olvasztjuk egybe” az első automata végállapotát és a második kezdőállapotát, hanem egy ε-átmenettel megyünk át
  - alternálás: mindegy a számozás
  - Kleene-lezárás: mindegy a számozás

A Shift-And-ben használt változók:

- B, $|B| = m$; ez lesz a keresett kifejezés és a keresés közötti egyetlen kapcsolat; $B[c]_i = 1$ ($B[c] = c_m c_{m-1} \dots c_1$), ha az i-edik állapotba egy c karakterrel megyünk, azaz hogyha



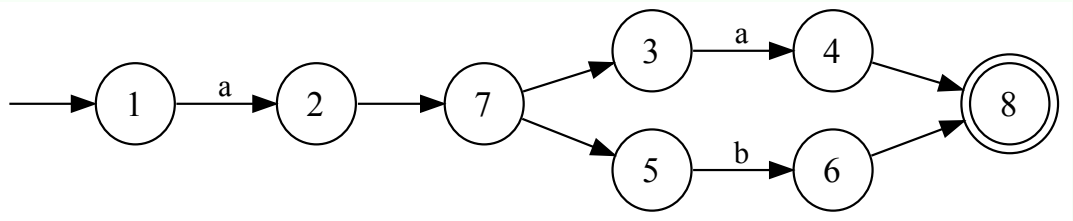
- D, $|D| = m$; ez a bitmaszk modellezi a nemdeterminisztikus véges automatát
- új szimbólum: $E[]$, egy bitmaszk ε-lezártját adja meg: a bitmaszk jelenleg aktív állapotai + az ezekből csupa ε-lépésekkel elérhető állapotok is aktívak lesznek
- magától értetődik, hogy egy $\dots 0$ bitmaszk ε-lezártjának első (legkisebb helyiértékű) bitje mindig 1 lesz, azaz $E[\dots 0] = \dots 1$ lesz, mivel a kezdőállapot minden lépésben elérhető, vagyis *a keresés mindig újrakezdhető*

ALG 6 Regex Shift-And

1: Megépítjük az E reguláris kifejezéshez tartozó Thompson-automatát.
2: $D = E[0 \ldots 0]$ (kezdőállapot ε-lezárása); $i = 0$; $c = T_i$
3: $D = (D \ll 1) \& B[c]$
4: $D = E[D]$
5: **if** D legutolsó bitje 1 **then**
6: TALÁLAT, kiir(i);
7: **end if**
8: $i = i + 1$
9: $c = T_i$
10: **if** $(i! = n)$ **then** Ugrás a 3. lépésre
11: **end if**
12: STOP

8. Példa

$T = aab$, $E = a(a + b)$
Az E-hez tartozó Thompson-féle automata:



$B[a] = 0000\ 1010$
$B[b] = 0010\ 0000$
$D = 0000\ 0001$
$T = \underline{a}ab$; $D = 0000\ 0010 \& 0000\ 1010 = 0000\ 0010$; $E[D] = 0101\ 0111$
$T = a\underline{a}b$; $D = 1010\ 1110 \& 0000\ 1010 = 0000\ 1010$; $E[D] = 1101\ 1111 \Rightarrow$ TALÁLAT, pozíció: 1
$T = aa\underline{b}$; $D = 1011\ 1110 \& 0010\ 0000 = 0010\ 0000$; $E[D] = 1010\ 0001 \Rightarrow$ TALÁLAT, pozíció: 2
STOP