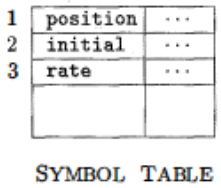
Paradygmaty i języki programowania

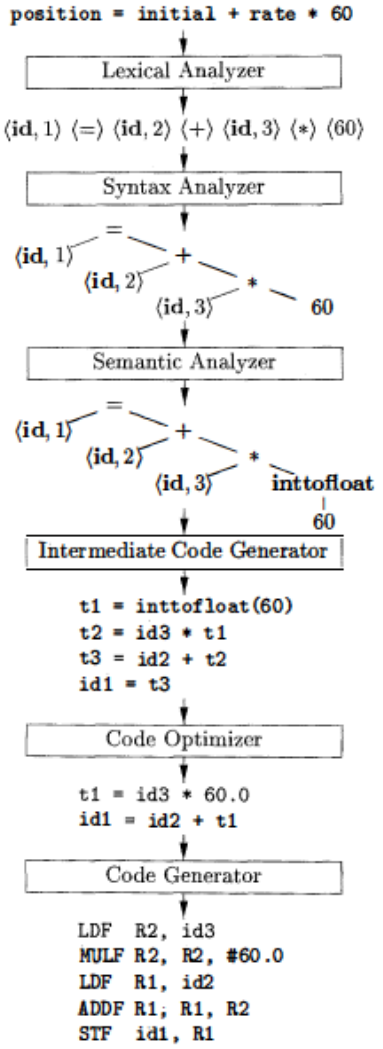
Analiza leksykalna

Skaner, RE, DAS, NAS, ε-NAS

Etapy pracy kompilatora

- Character stream
- [Lexical Analyzer]
  - token stream
- [Syntax Analyzer]
  - syntax tree
- [Semantic Analyzer]
  - syntax tree
- [Intermediate Code Generator]
  - intermediate representation
- [Machine-Independent Code Optimizer]
  - intermediate representation
- [Code Generator]
  - target-machine code
- [Machine-Dependent Code Optimizer]
  - target-machine code





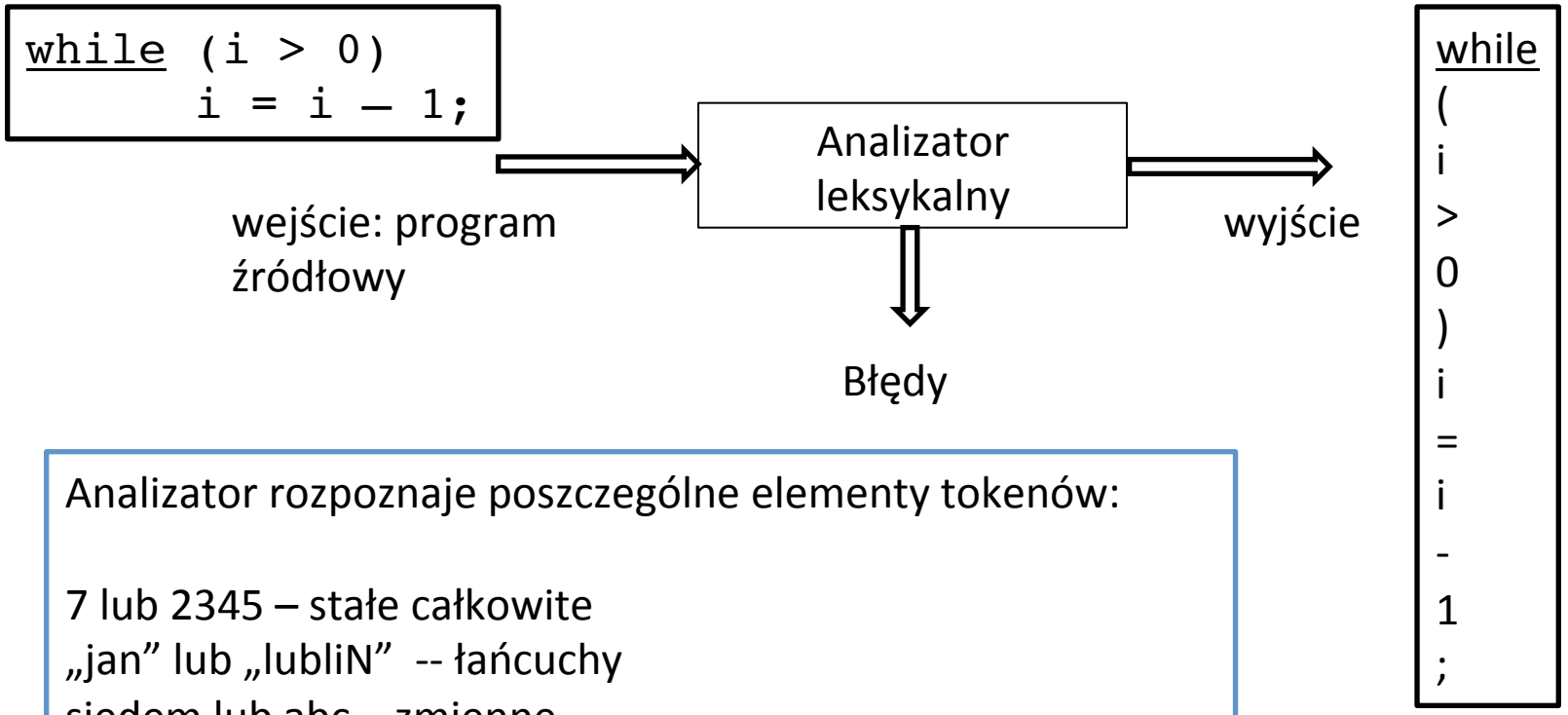
Analizator leksykalny (AL), skaner

- Skaner – pierwsza składowa kompilatora, która sprawdza czy można podzielić ciąg symboli wejściowych (program źródłowy) na jednostki leksykalne danego języka programowania, a więc na tzw. tokeny.
- Skaner nie zwraca uwagi na ‘śmieci’, typy, kolejność tokenów, niezadeklarowane nazwy, itd. Te zadania należą do analizatora składni
- Skaner nie ma pojęcia o grupowaniu i organizacji tokenów ...

AL

- Tokeny są to klasy leksemów, a więc podstawowych elementów języka programowania. Np.
  - stałe (int, double, char, string, itd)
  - operatory (arytmetyczne, logiczne, relacyjne)
  - znaki przestankowe
  - słowa kluczowe języka

Schemat skanera



Analizator rozpoznaje poszczególne elementy tokenów:

7 lub 2345 – stałe całkowite
„jan” lub „lubliN” -- łańcuchy
siedem lub abc – zmienne

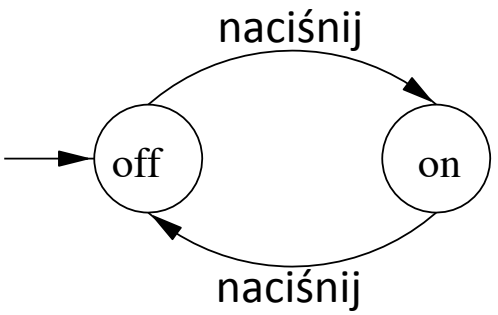
Elementy te noszą nazwę leksemów i należą do różnych klas tokenów (znaczników).

Budowa skanera

- Dwie metody realizacji
  - wprost (skaner *ad hoc*)
  - wykorzystanie wyrażeń regularnych i automatów skończonych
- Ad hoc: program oparty na pętli (*loop*) i przełącznikach (*switch*) – *loop&switch*
  - należy podać definicje tokenów
  - czytać wejście z lewa na prawo
  - analizować i jeśli OK wówczas
  - wykonać odpowiedni skok w programie
  - Metoda uciążliwa!

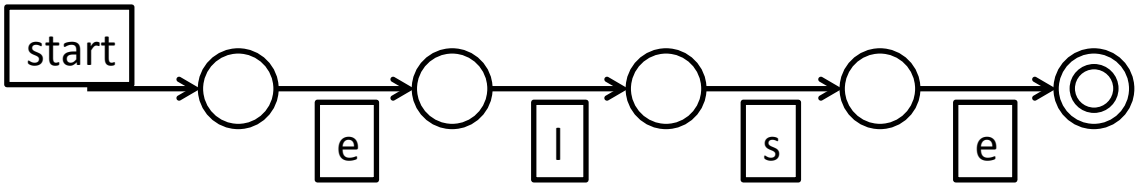
Deterministyczny Automat Skończony

- Przykład: przełącznik; dwa stany:
- początkowy, końcowy; funkcja
- przejścia.
- Zastosowania DAS:
  - Analiza obwodów cyfrowych
  - Analizator leksykograficzny (rozbija tekst na jednostki leksykograficzne)
  - Skanowanie tekstu w celu wyszukiwania konkretnych słow (wyszukiwarki sieciowe)
  - Weryfikacja systemów o skończonej liczbie stanów (transakcje elektroniczne w internecie)



DAS

- Przykład: Wyszukiwanie, modelowanie łańcucha (słowa) **else**



- Ważną rolę w teorii automatów skończonych grają
  - Gramatyki (parser, przetwarzanie wyrażeń, E->E+E stwierdza, że wyrażenie można otrzymać z dwu wyrażeń połączonych znakiem plus; gramatyki bezkontekstowe)
  - Wyrażenia regularne (wzorce danych tekstowych, reprezentujące łańcuchy; UNIX: [A-Z][a-z]*[ ][A-Z][a-z]*)

Podstawowe pojęcia

- Alfabet: Σ – skończony, niepusty zbiór symboli
  - $\Sigma = \{0, 1\}$ – alfabet binarny;
  - $\Sigma = \{a, b, \ldots, z\}$ – zbiór małych liter
- Łańcuch , słowo – skończony ciąg symboli wziętych z jakiegoś alfabetu
- Łańcuch pusty – ε – słowo o zerowej długości (można wybrać z każdego alfabetu)
- Długość słowa – liczba symboli niepustych
- Potęgi alfabetu – Σ^k -- zbiór słów o długości k nad alfabetem Σ
- $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{0, 1, 00, 01, 10, 11\}$, …

Podstawowe pojęcia

- Konwencja
  - Małe litery z początku alfabetu łac.: a, b, c … - symbole
  - Małe litery z końca alfabetu: w, x, y, z – słowa, łańcuchy
- Zbiór wszystkich łańcuchów nad Σ oznaczamy przez Σ^* (* - operator Kleene'go)
  - $\Sigma^* = \Sigma^0 \mathbf{U} \Sigma^1 \mathbf{U} \Sigma^2 \mathbf{U} \ldots$

Język

- Językiem nazywamy zbiór łańcuchów (słów) wybranych z pewnego alfabetu Σ (inaczej: jeśli Σ jest alfabetem i $L \subseteq \Sigma^*$ to L nazywamy językiem nad Σ)
- Przykłady
  - język polski, angielski, C, C++, …
  - Języki abstrakcyjne
    - Język ze słów złożonych z n zer i n jedynek {ε, 01, 0011, 000111, …}
    - Język ze słów złożonych z 0 i 1 zawierających tyle samo zer i jedynek {ε, 01, 0101, 1010, 0110, …}
    - Zbiór liczb pierwszych zapisanych w postaci binarnej {ε, 10, 11, 101, 111, 1011, …}
    - Σ^*
    - $\varnothing$ - zbiór pusty
    - {ε}; zauważmy, że $\varepsilon \neq \varnothing$

Problem

- Dla danego $w \in \Sigma^*$ rozstrzygnąć czy $w \in L(\Sigma)$

  Przykład. Rozstrzygnięcie czy napis należy do języka liczb pierwszych L_p jest w ogólności trudnym problemem obliczeniowym

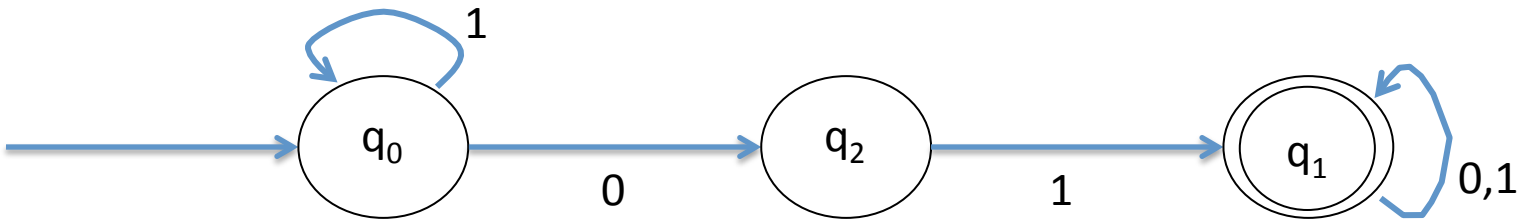
  Praca parsera!

Definicja DAS

- A = ($Q, \Sigma, \delta, q_0, F$)
  - A = nazwa DAS
  - Q = skończony zbiór stanów
  - Σ = zbiór symboli wejściowych
  - δ = funkcja przejścia (ze stanu do stanu)
  - q_0= stan początkowy
  - F = zbiór stanów końcowych

Definicja DAS

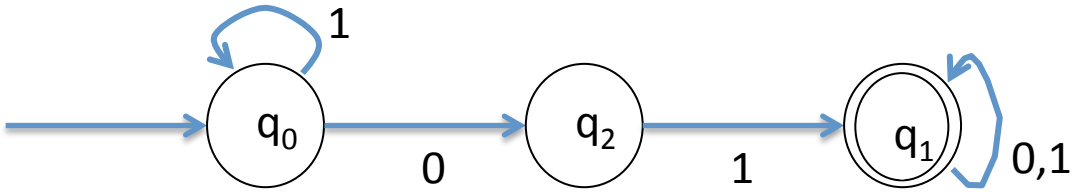
- Notacja wygodniejsza
  - Diagram przejść
  - Tablica przejśc
- Przykład: DAS akceptujący łańcuchy zawierające podłańcuch 01



DAS, funkcja przejścia

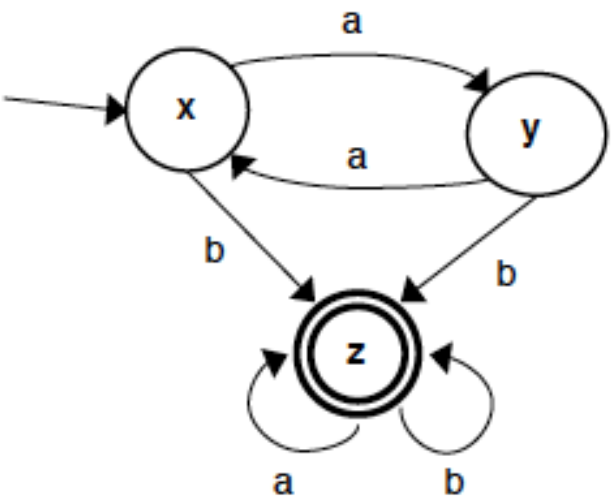
- Funkcja przejścia dla DAS z poprzedniego przykładu; δ(*stan, symbol_wejściowy*) →*stan*;

	0	1
→ q_0	q_2	q_0
*q_1	q_1	q_1
q_2	q_2	q_1



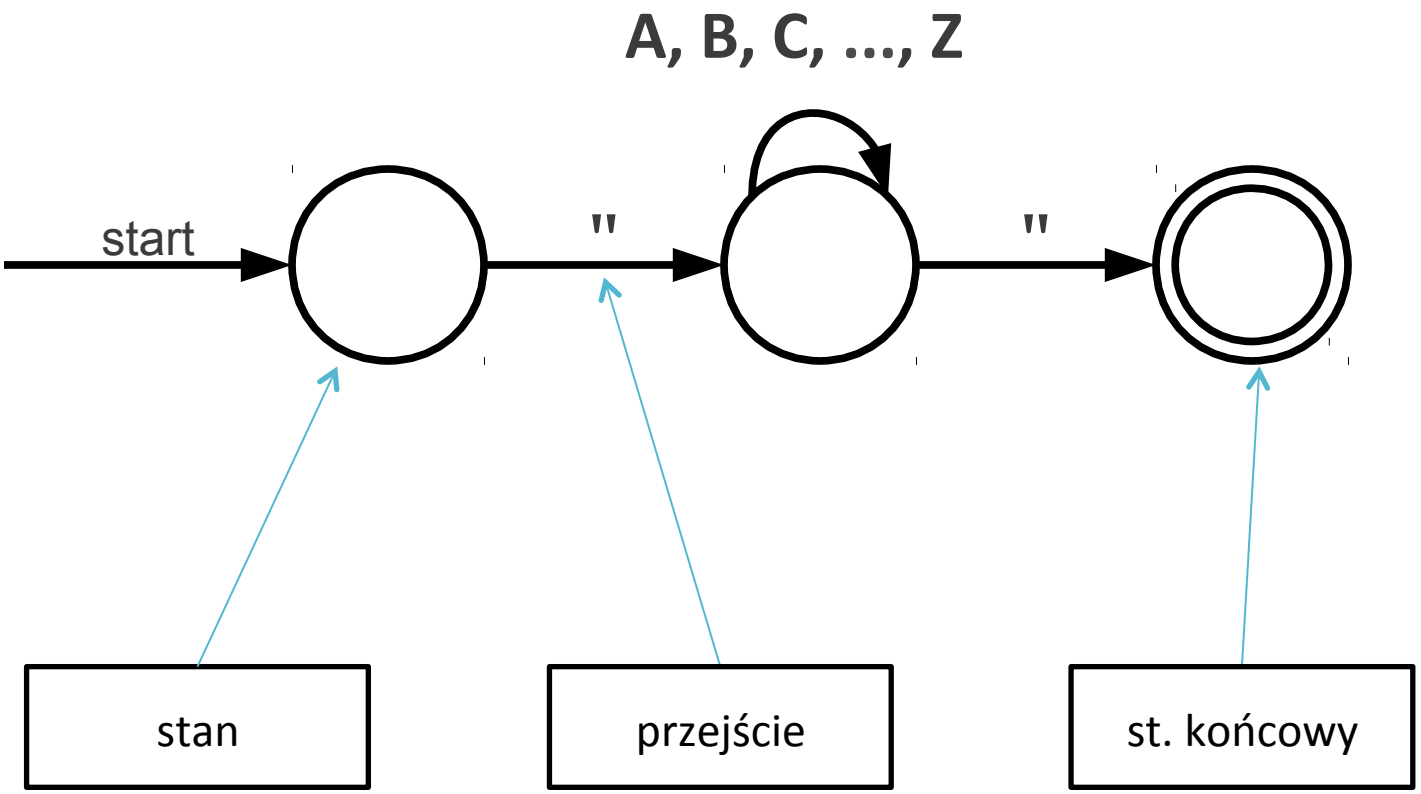
(gwiazdka oznacza stany akceptujące)

DAS i δ(stan,symbol)

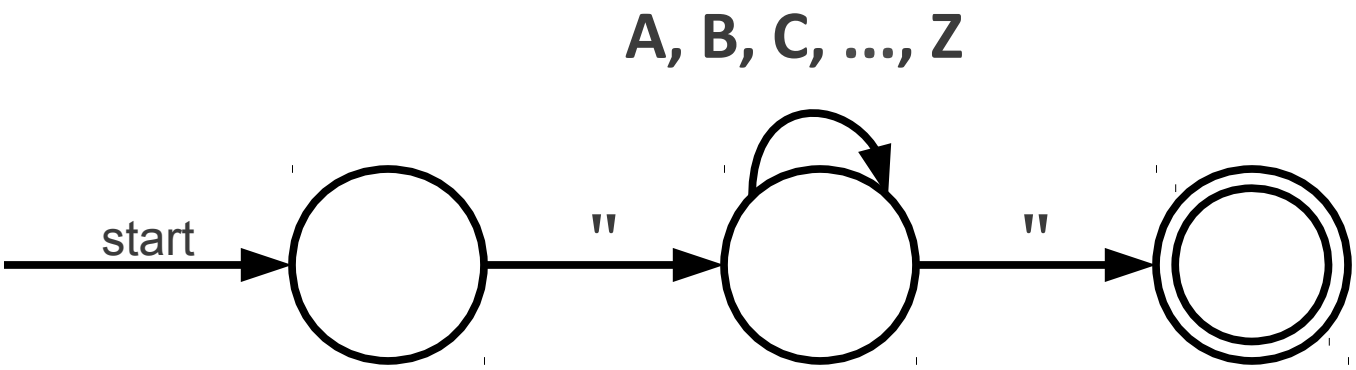


		a	b
(start)	x:	y	z
	y:	x	z
(final)	z:	z	z

Prosty automat

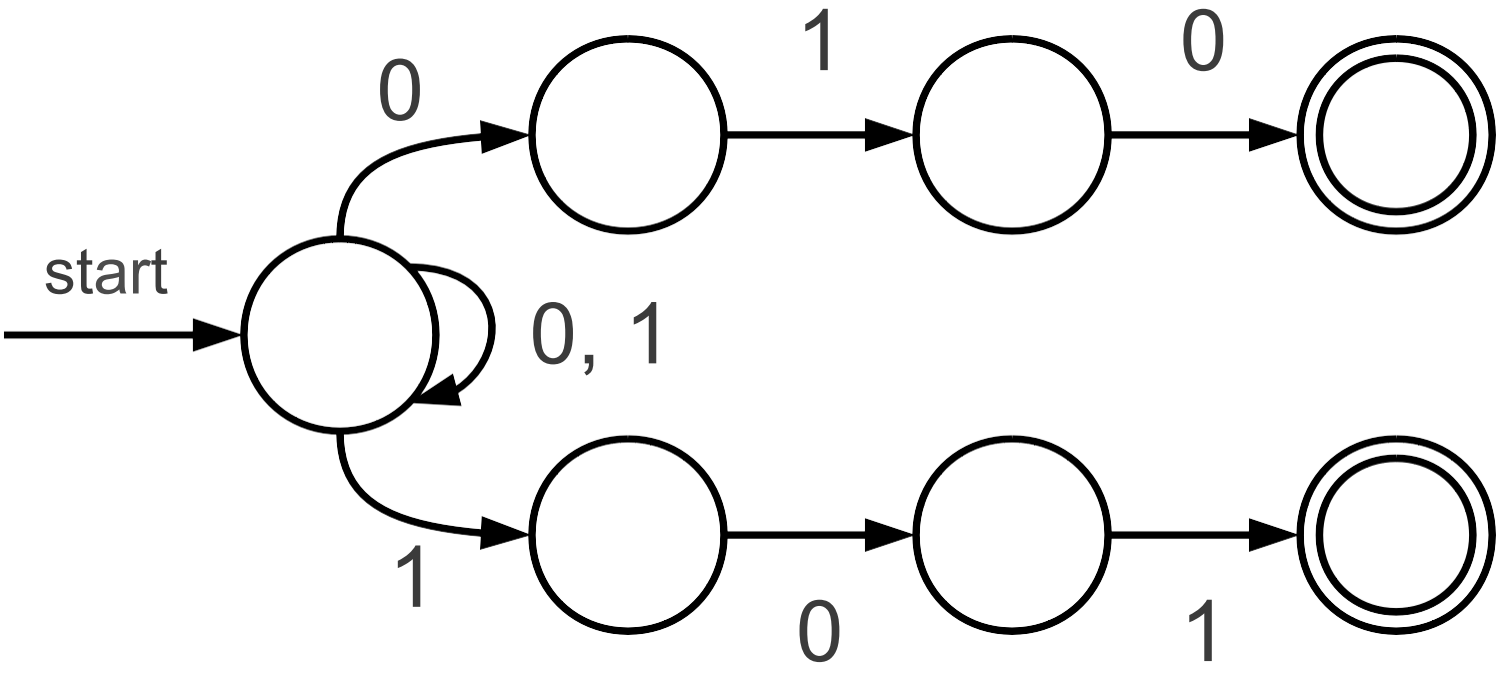


Prosty automat



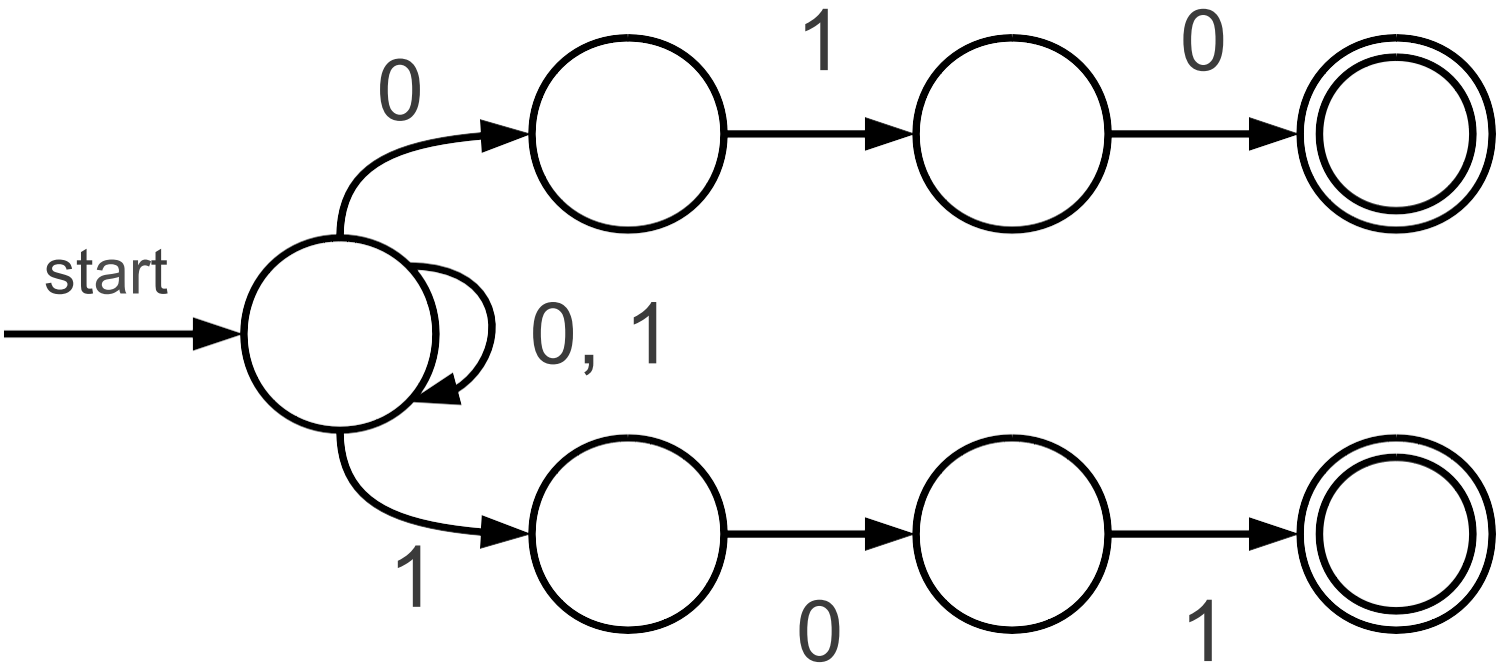
''AUTOMAT''

A. złożony, NAS



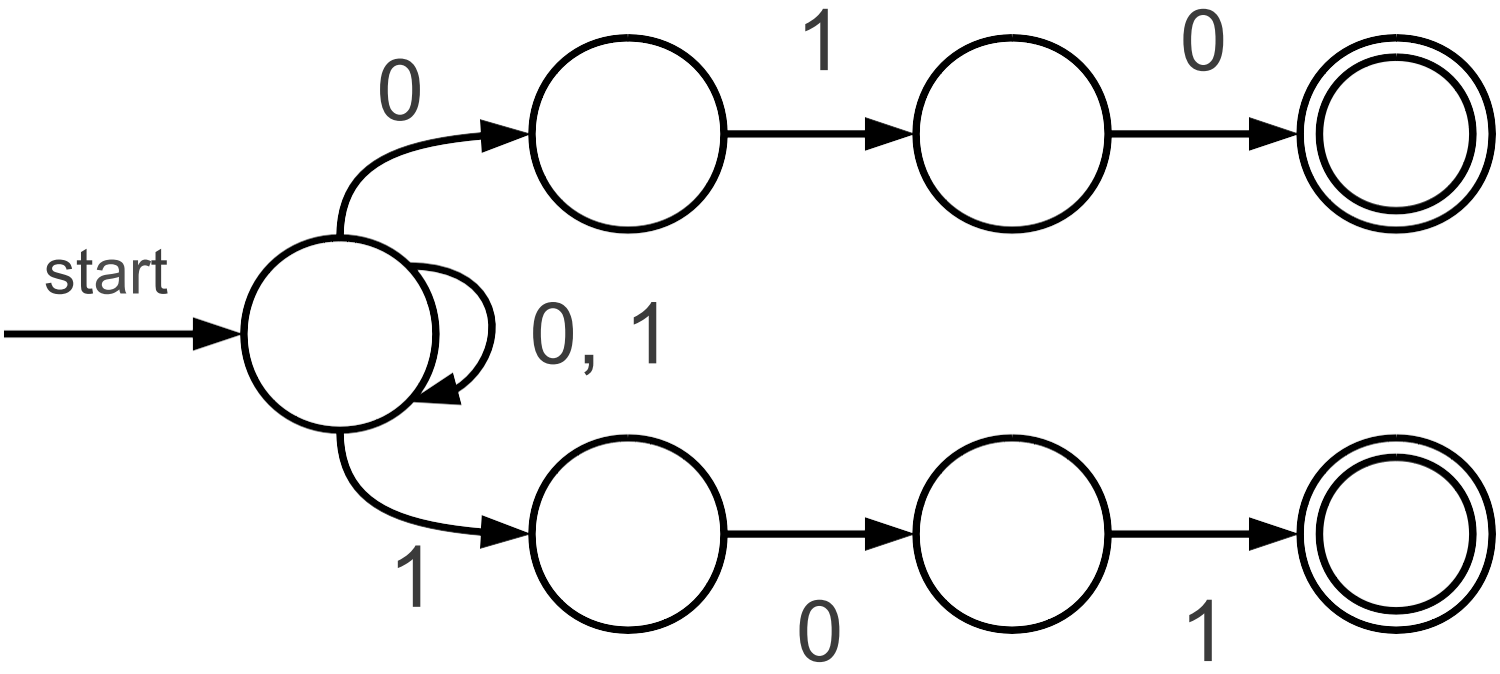
(Przykład pożyczony z kursu cs143, Stanford)

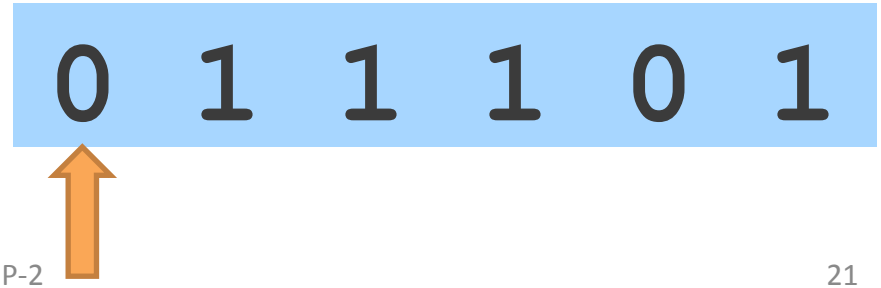
A. złożony, NAS



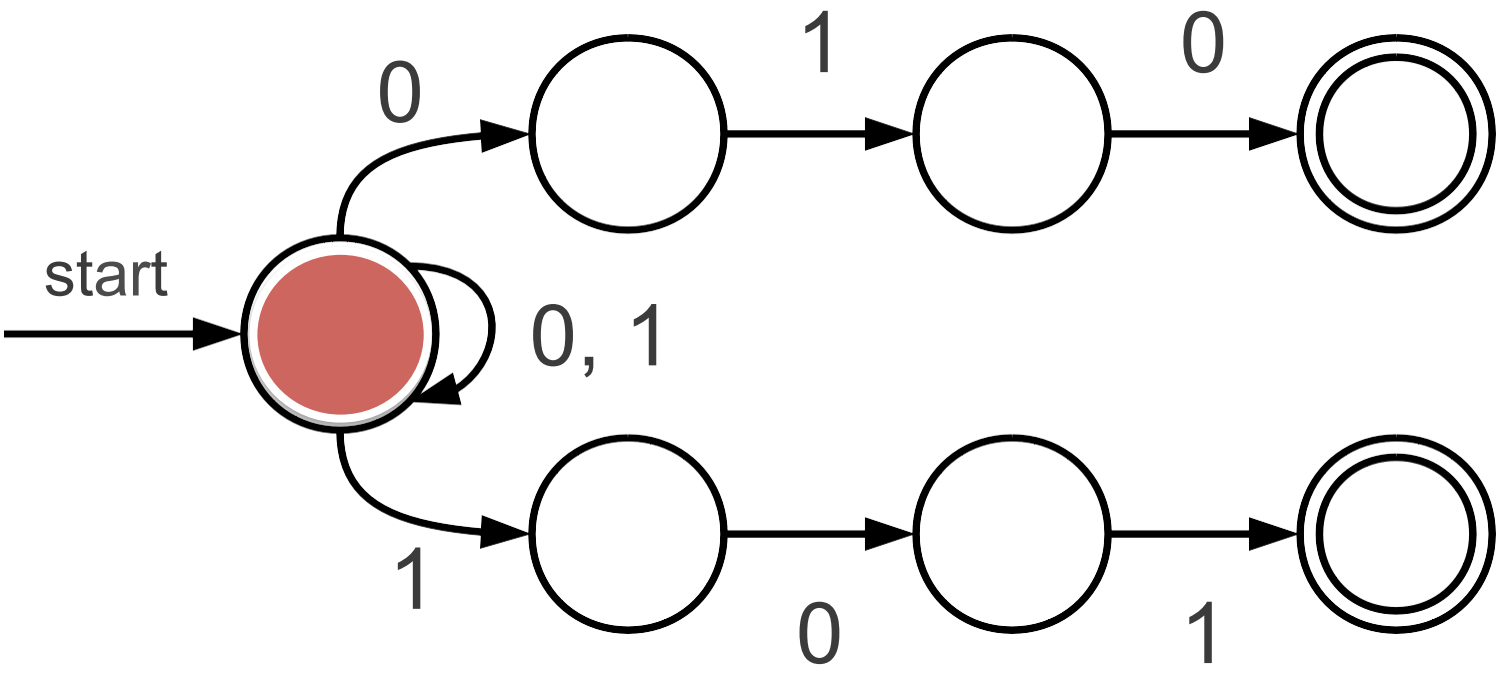
0 1 1 1 0 1

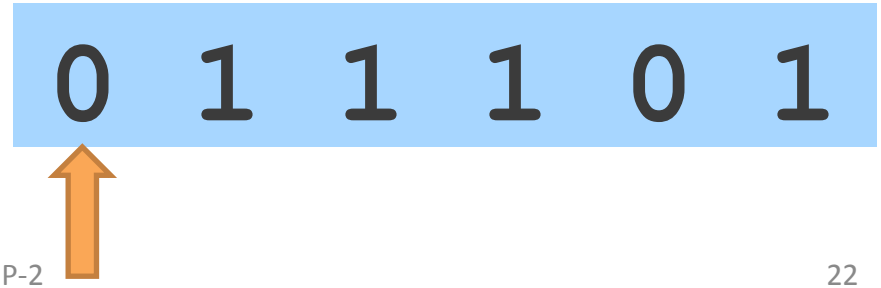
A. złożony, NAS



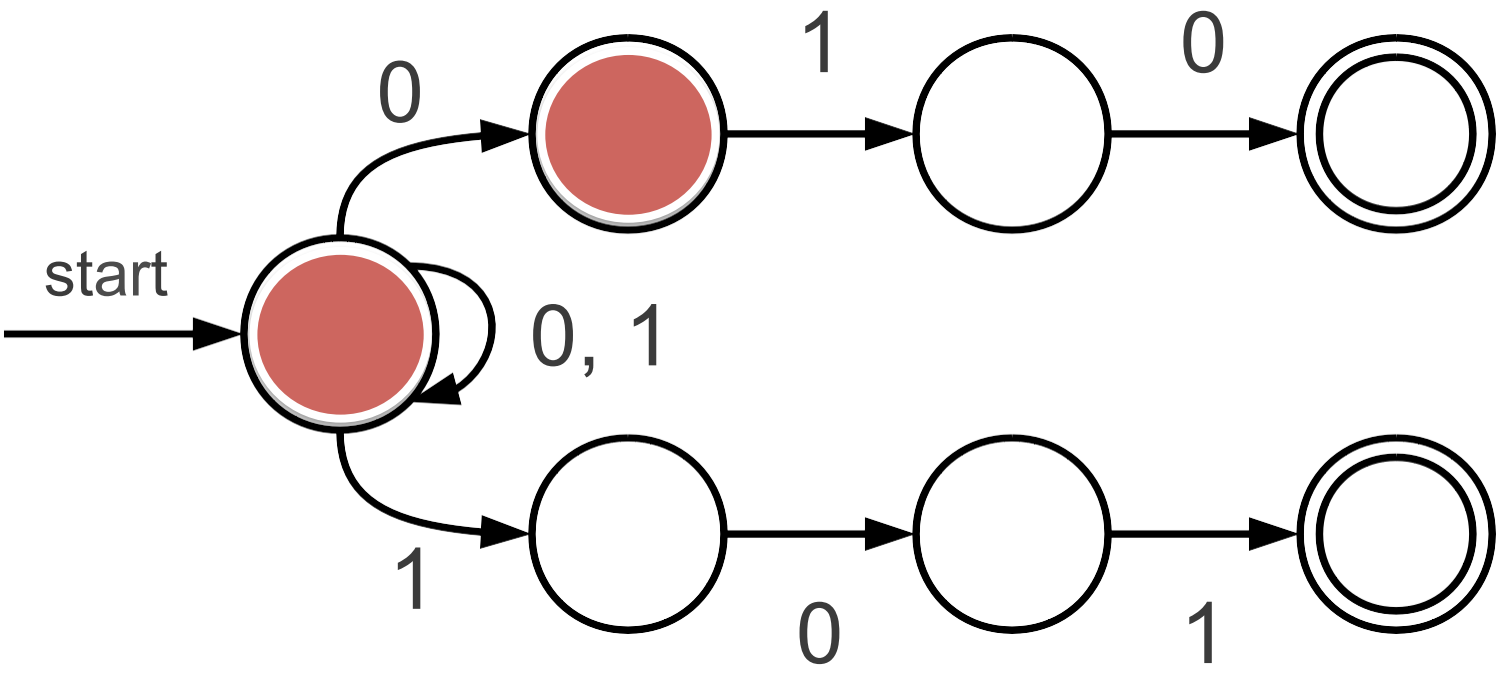


A. złożony, NAS



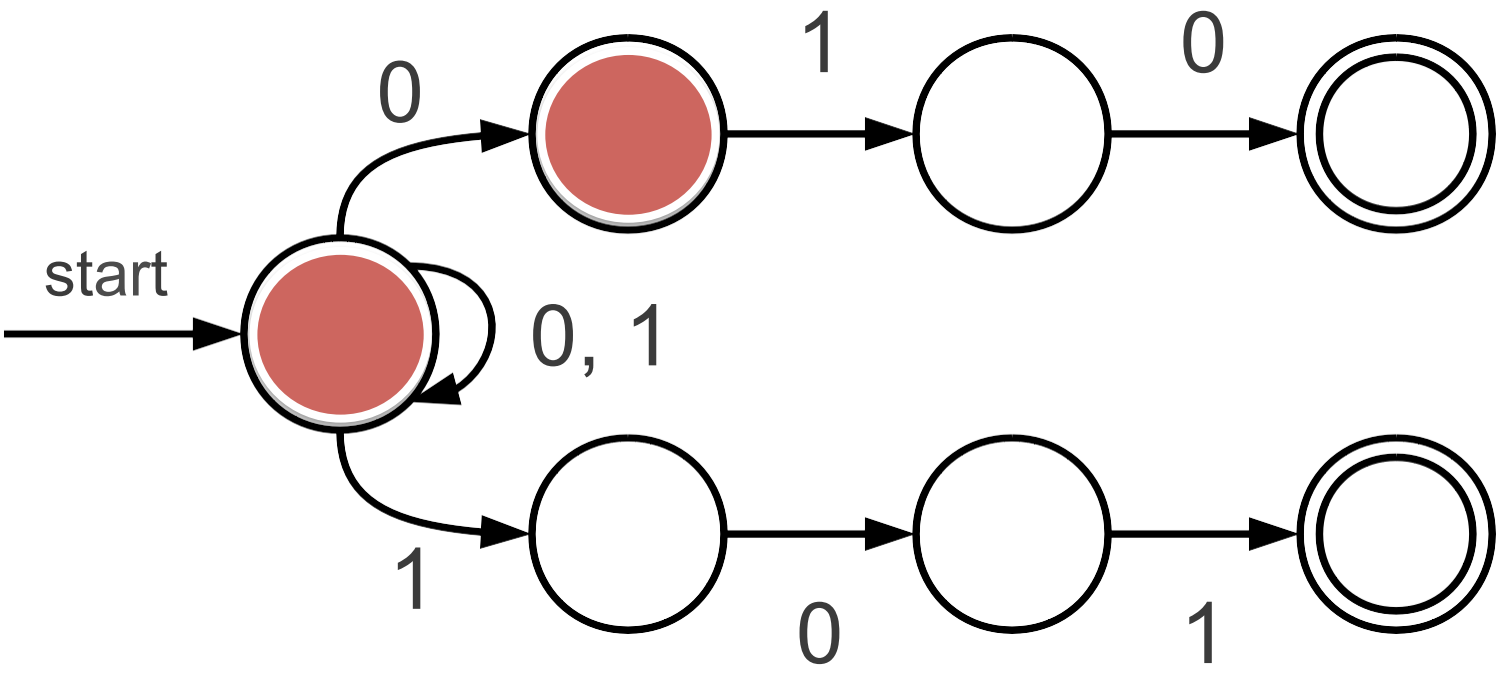


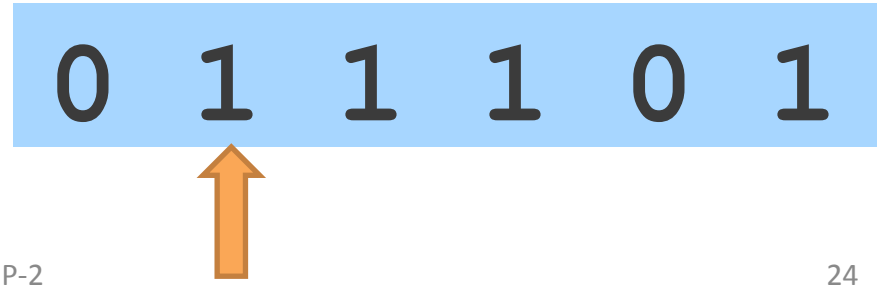
A. złożony, NAS



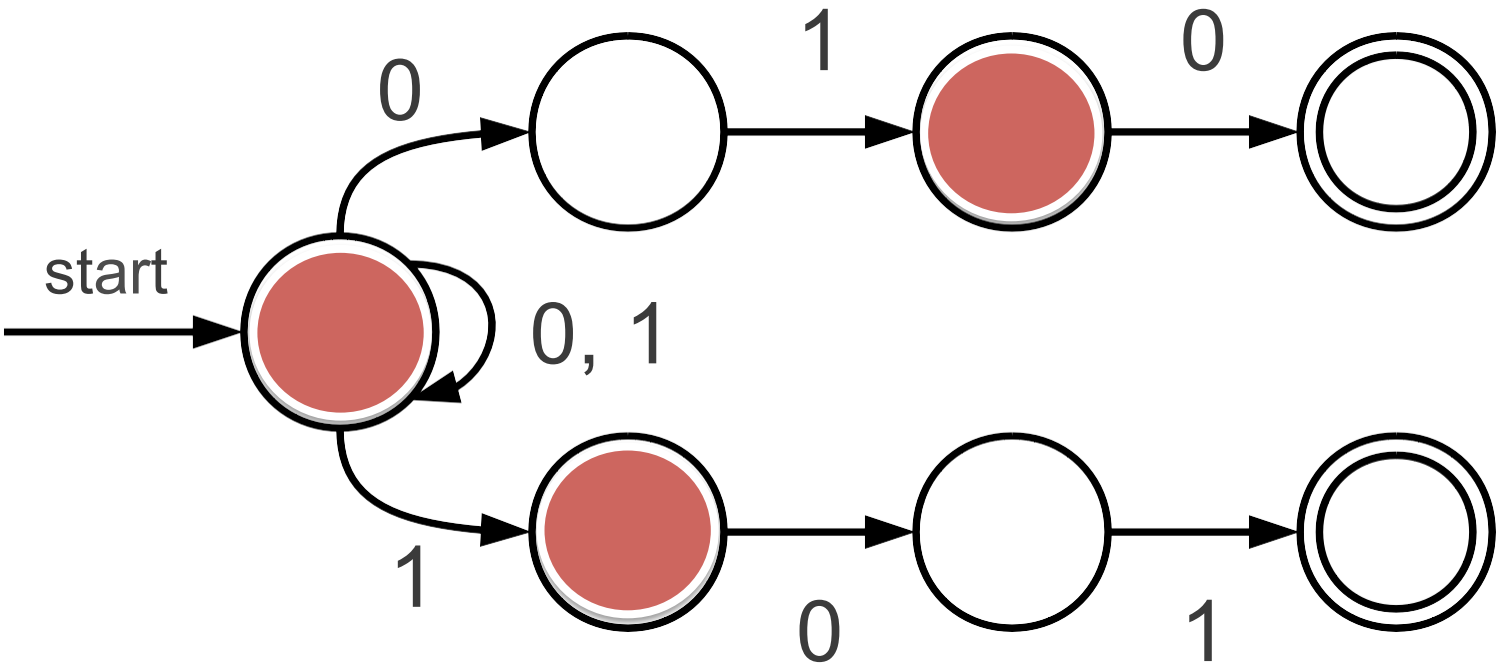
0 1 1 1 0 1

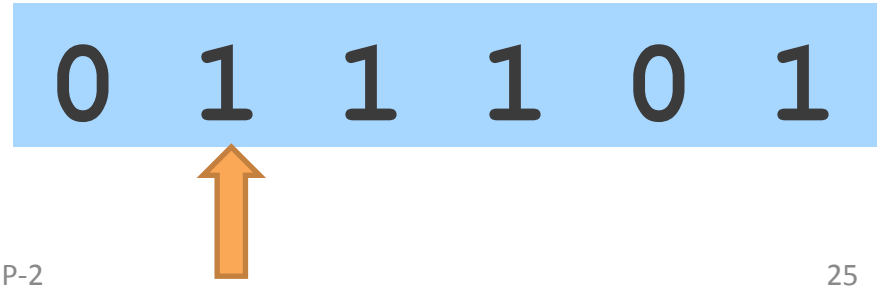
A. złożony, NAS



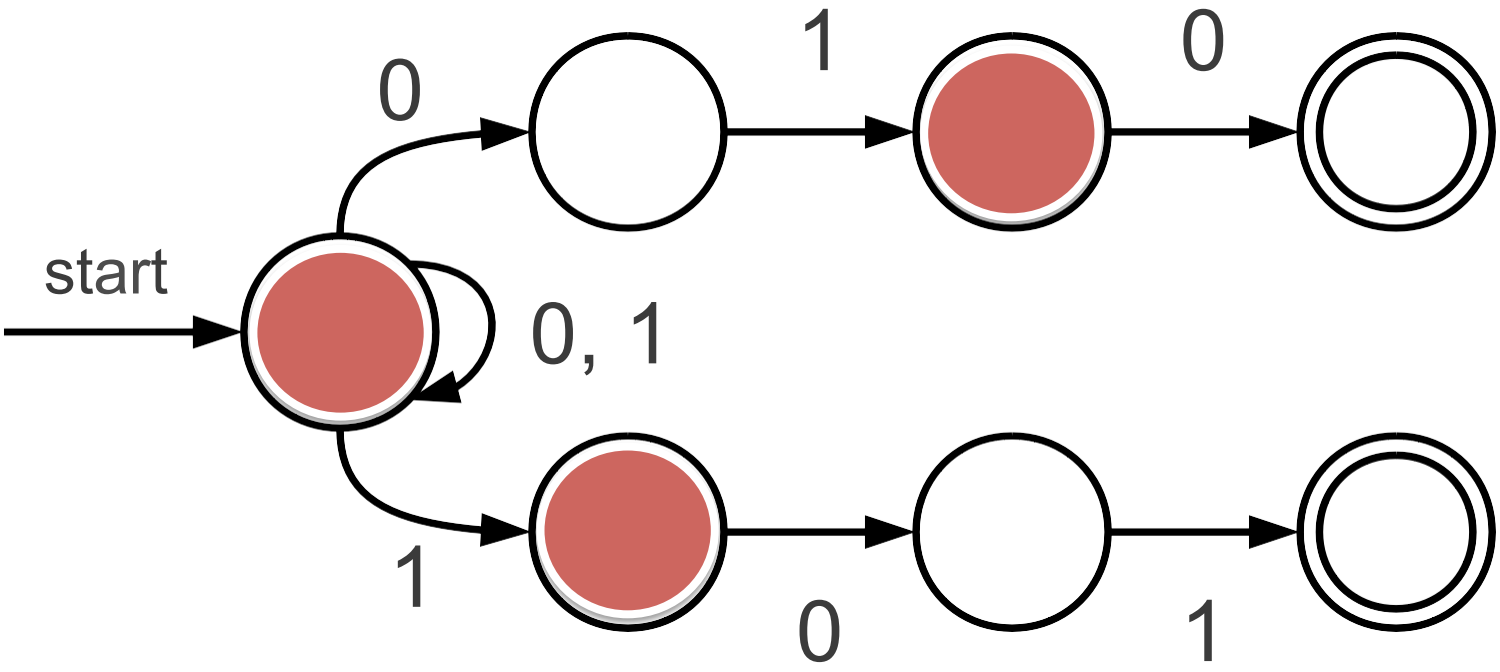


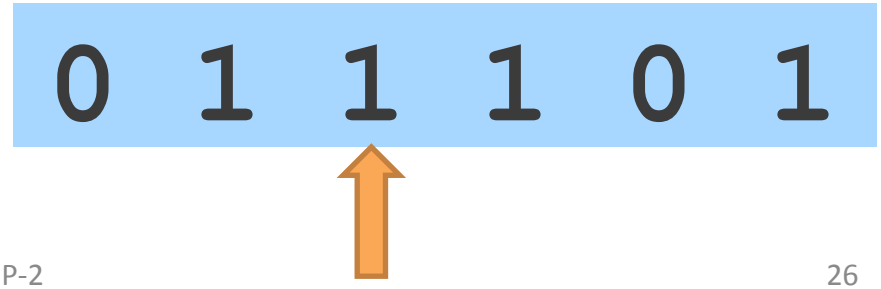
A. złożony, NAS



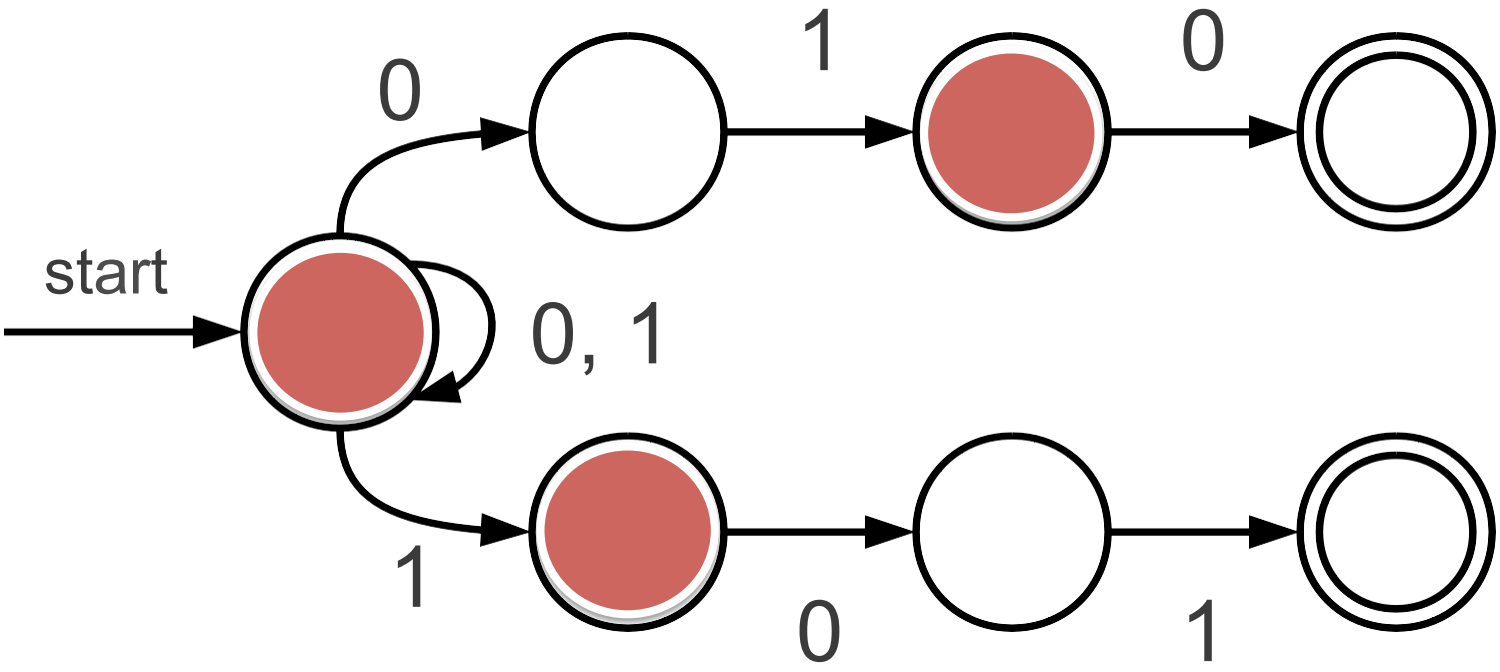


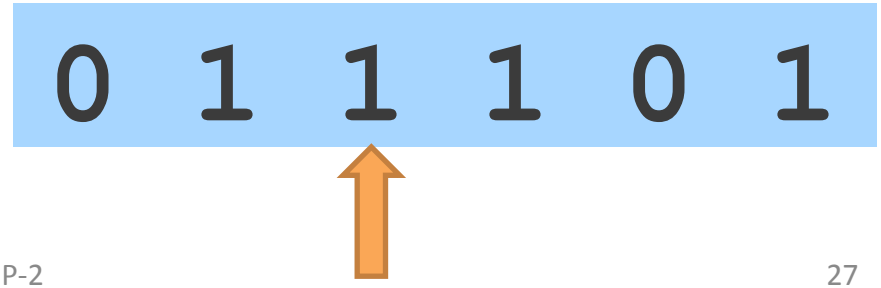
A. złożony, NAS



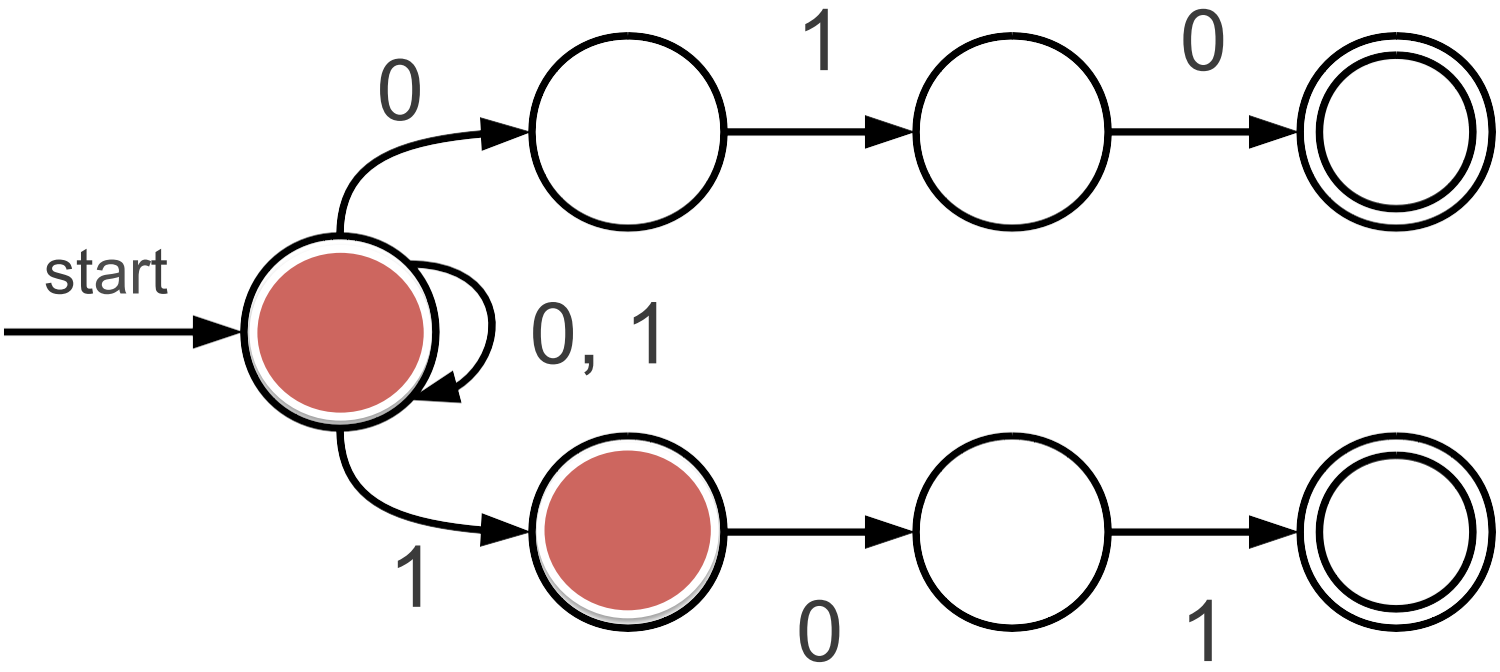


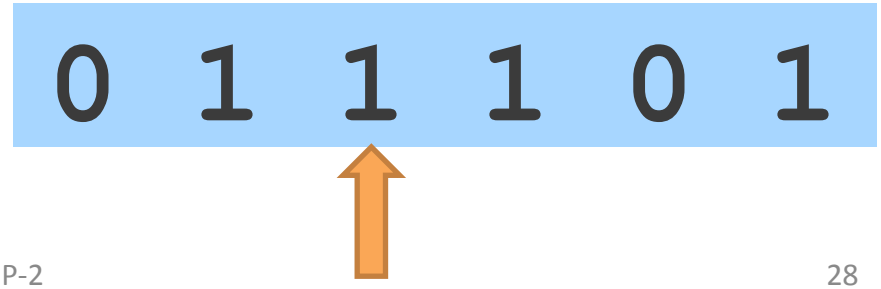
A. złożony, NAS



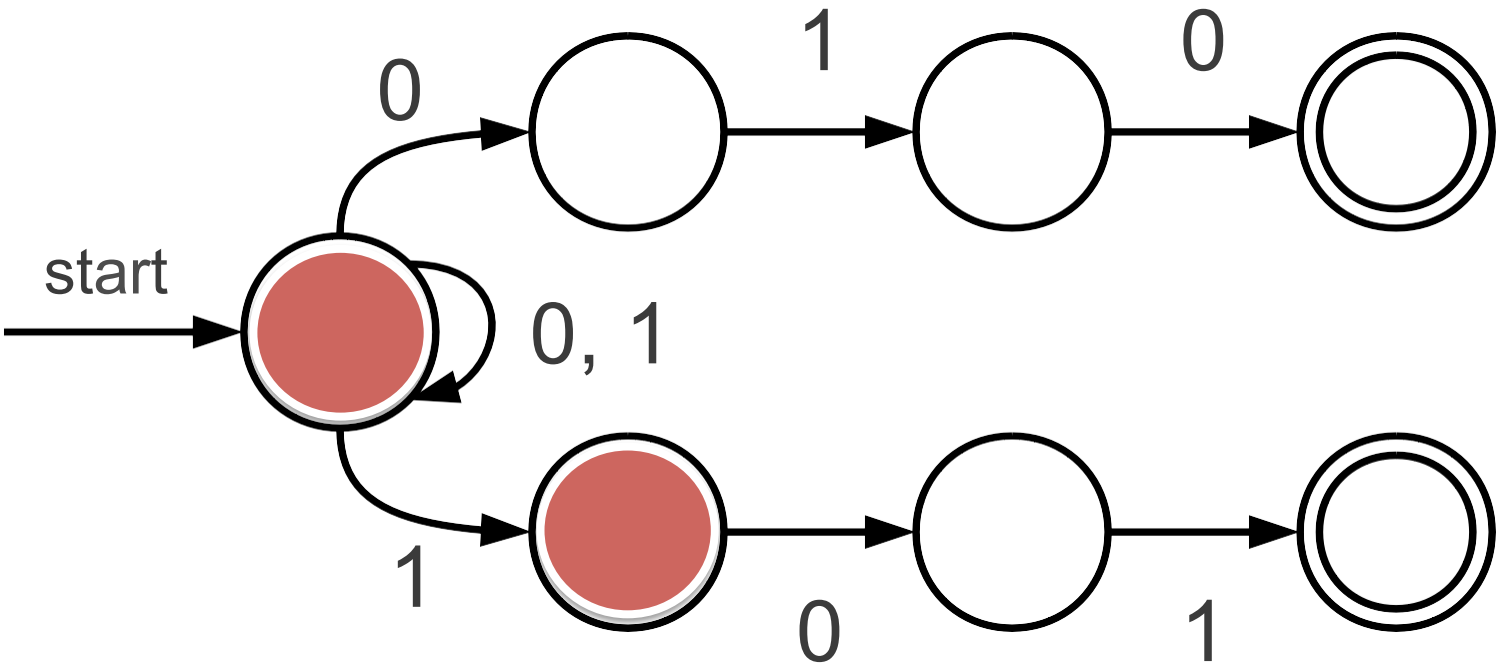


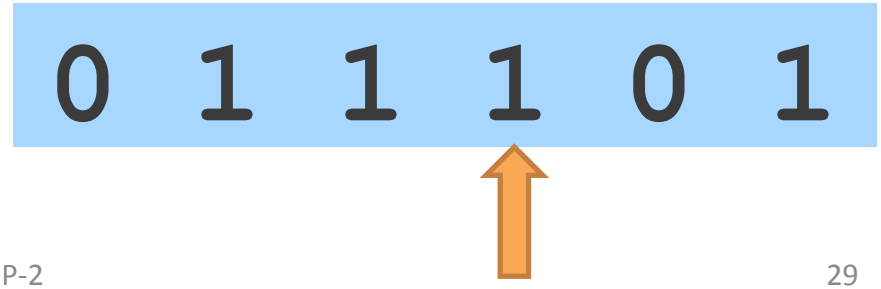
A. złożony, NAS



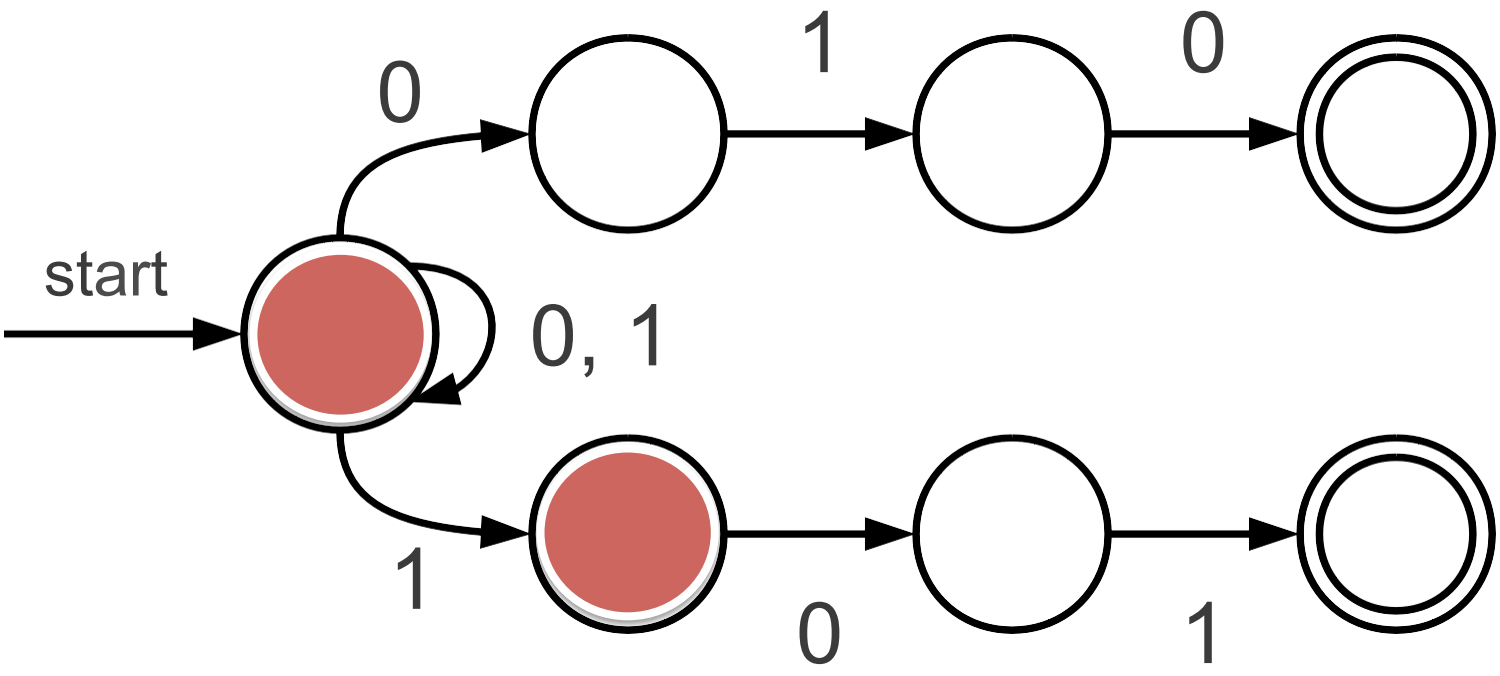


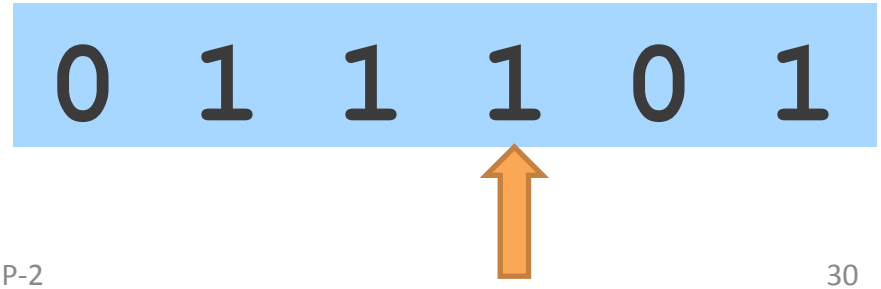
A. złożony, NAS



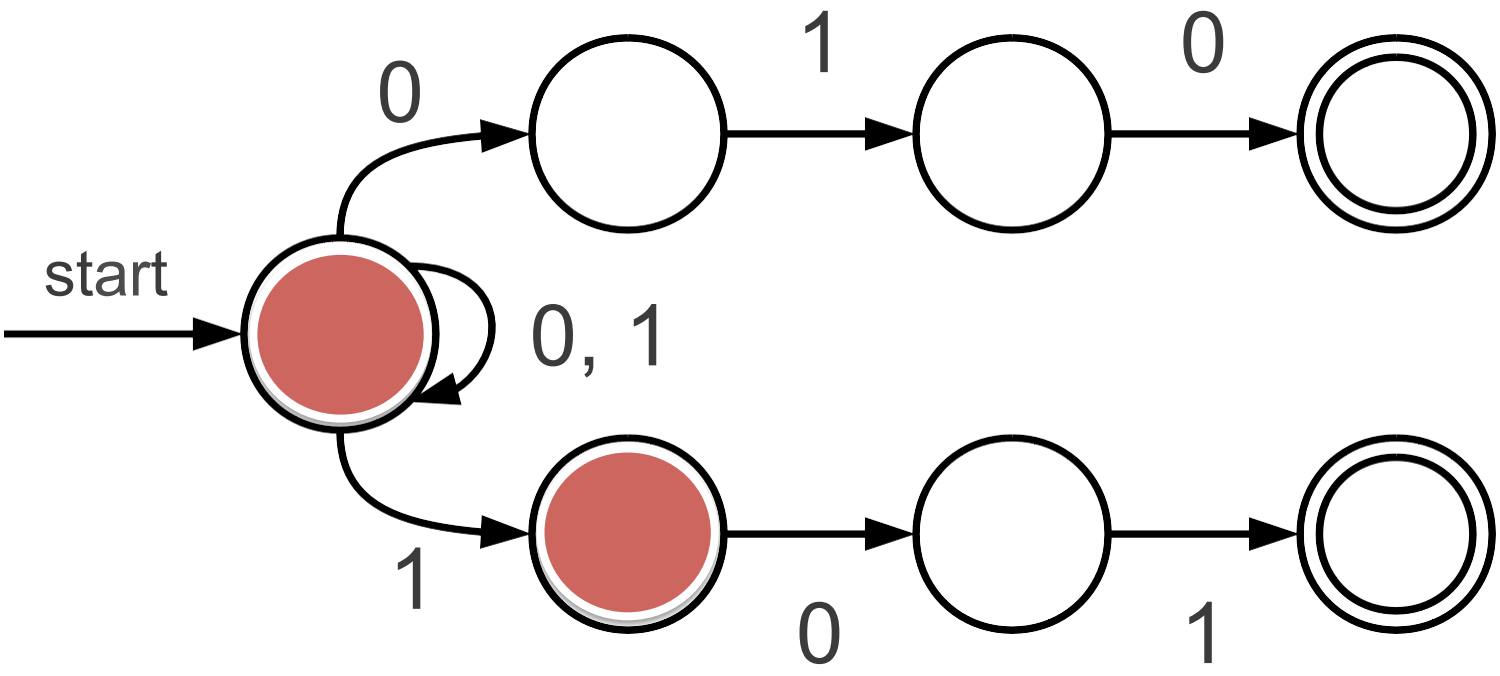


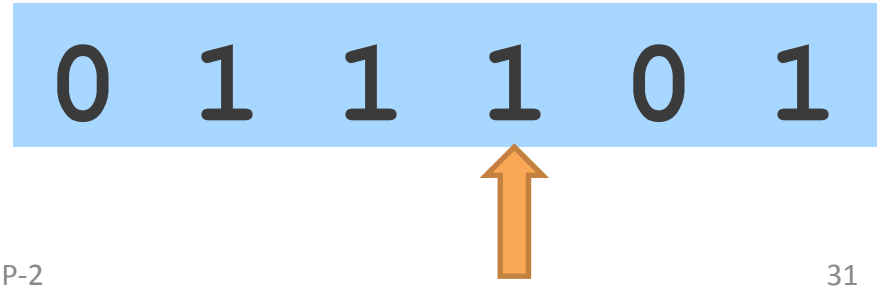
A. złożony, NAS



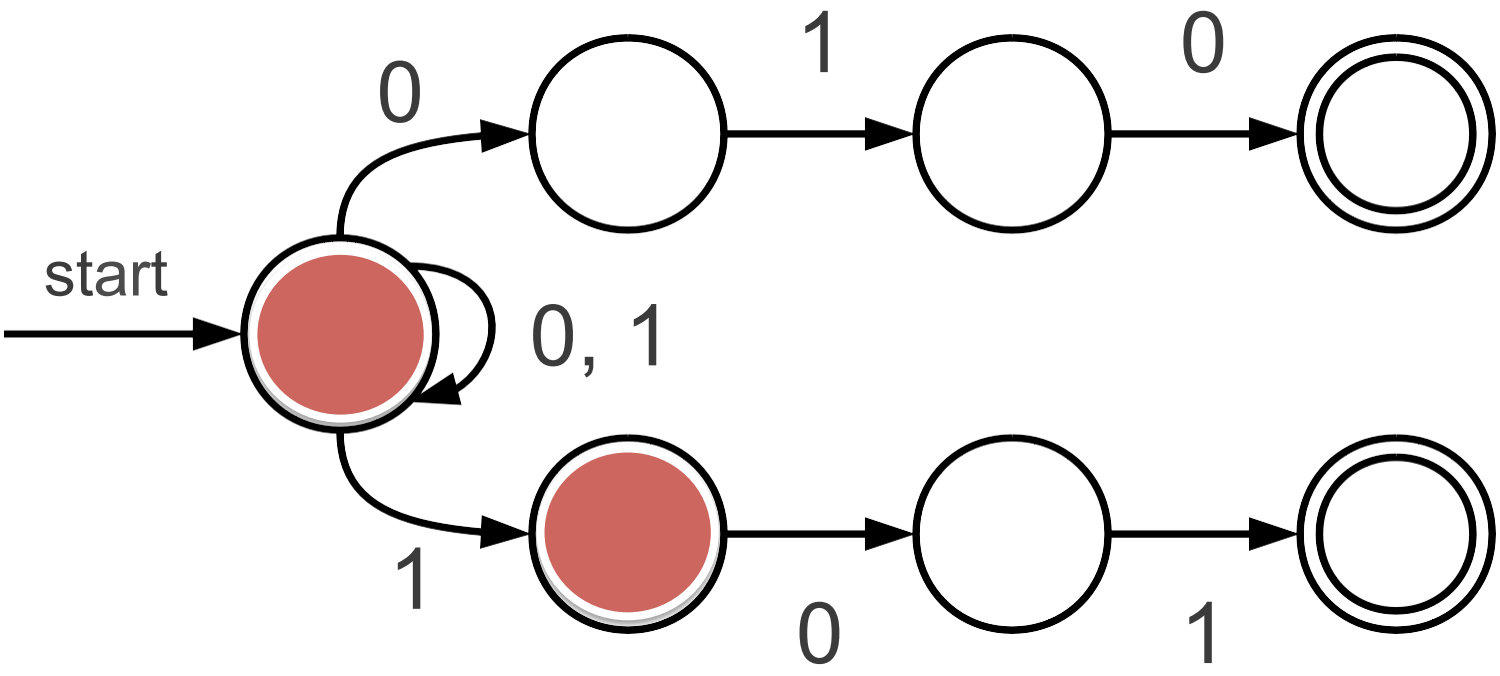


A. złożony, NAS



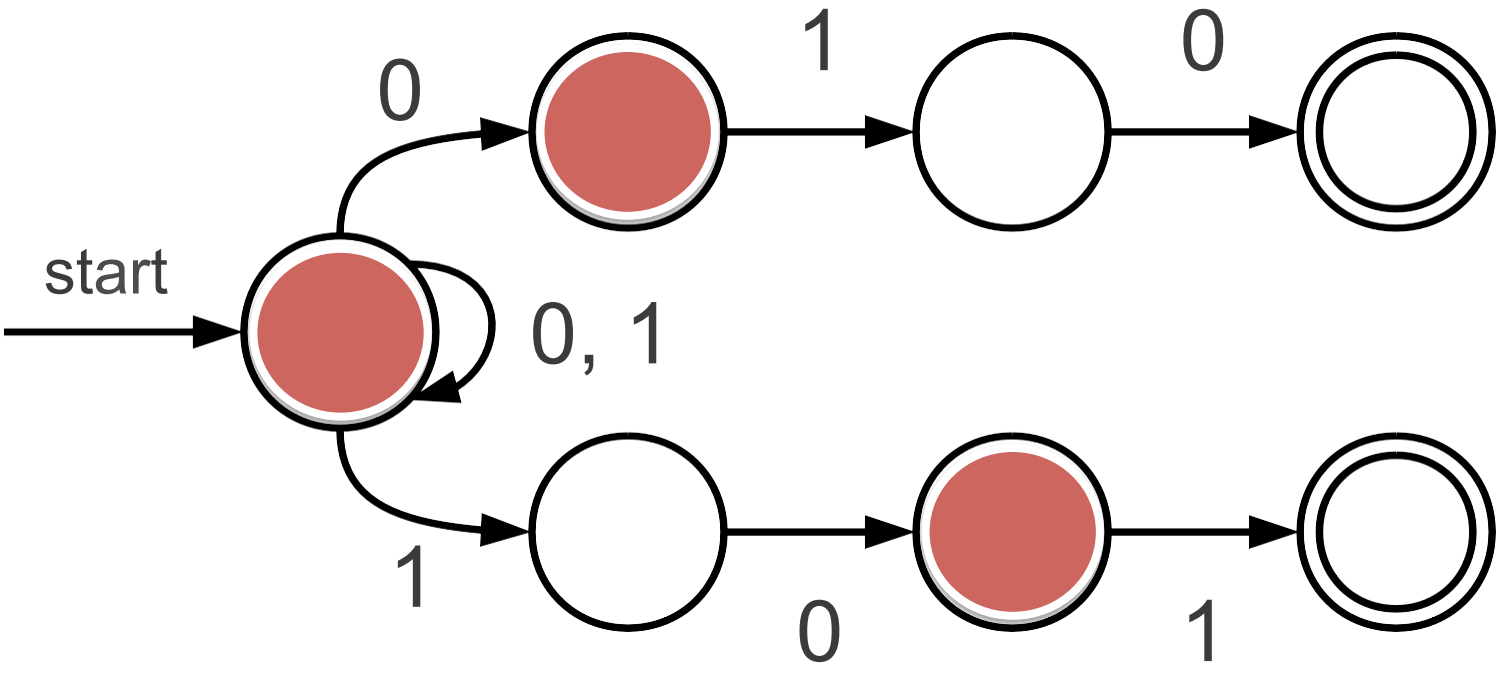


A. złożony, NAS



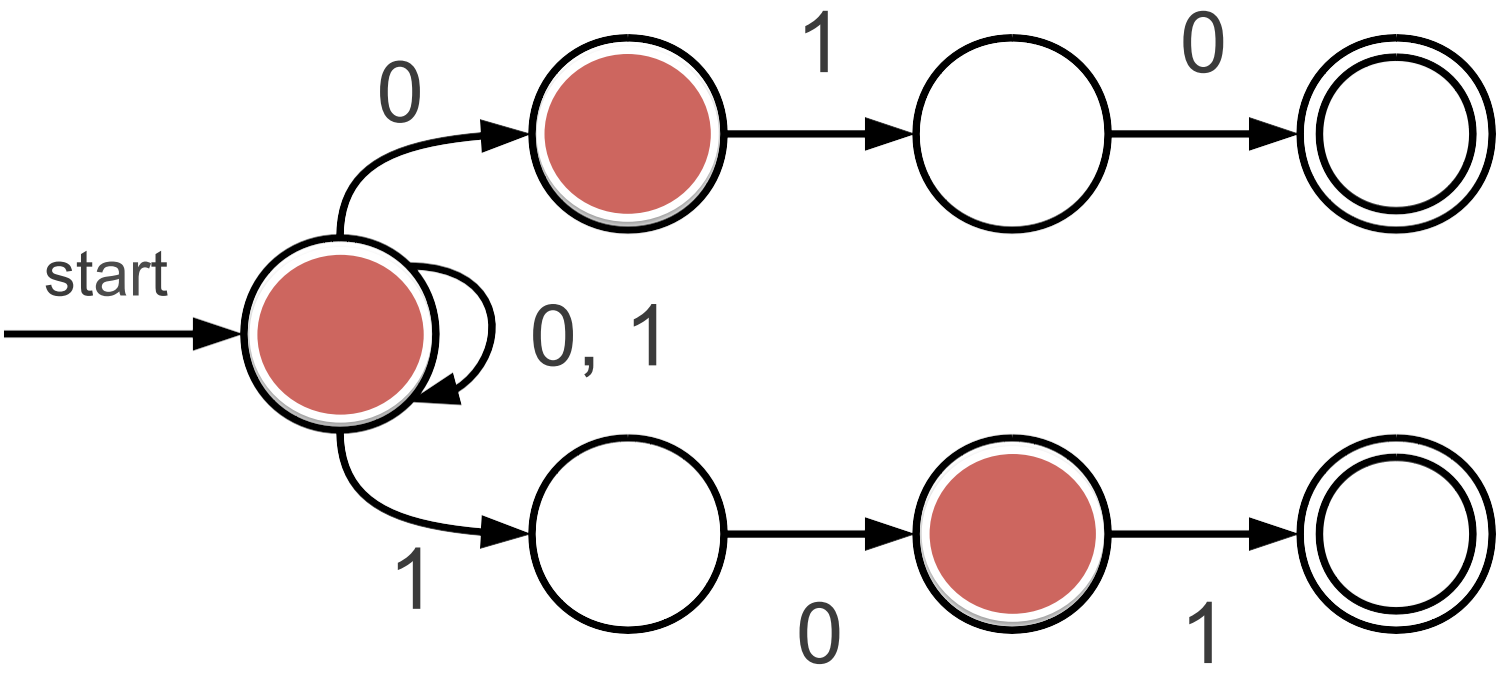
0 1 1 1 0 1

A. złożony, NAS



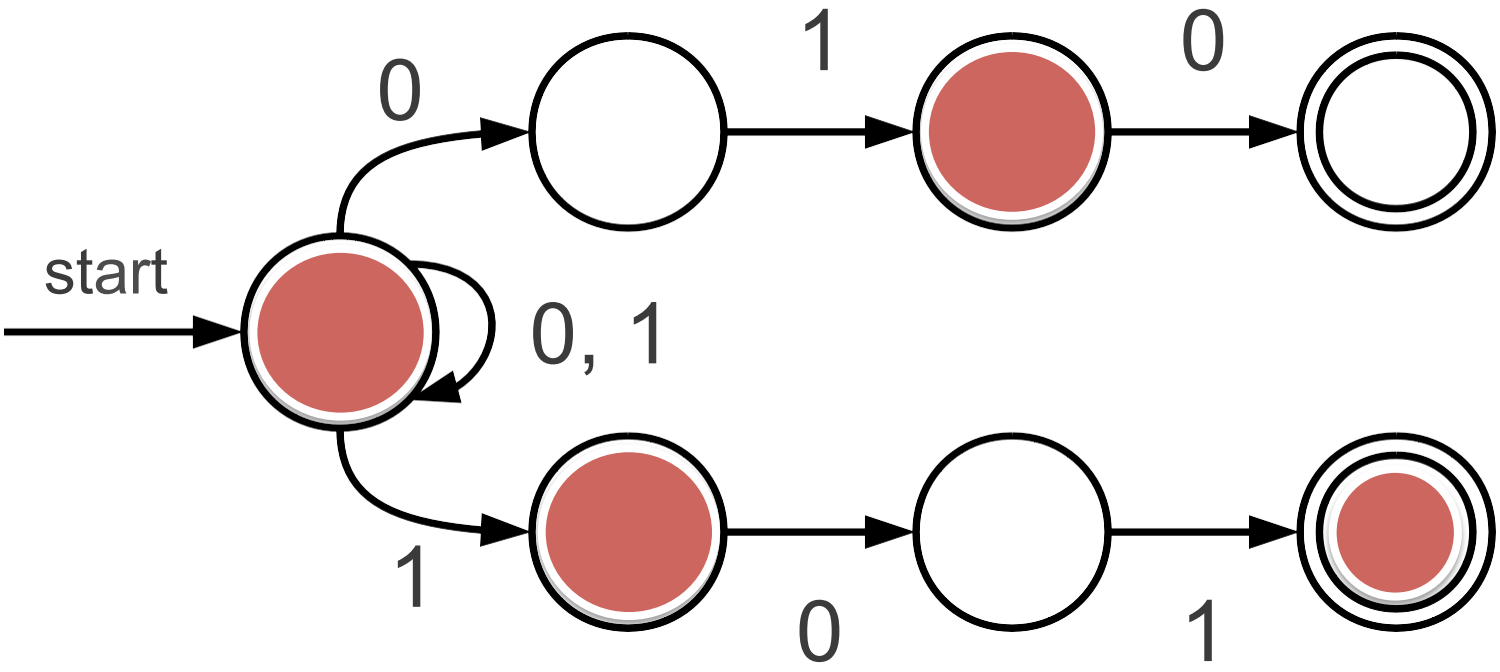
0 1 1 1 0 1

A. złożony, NAS



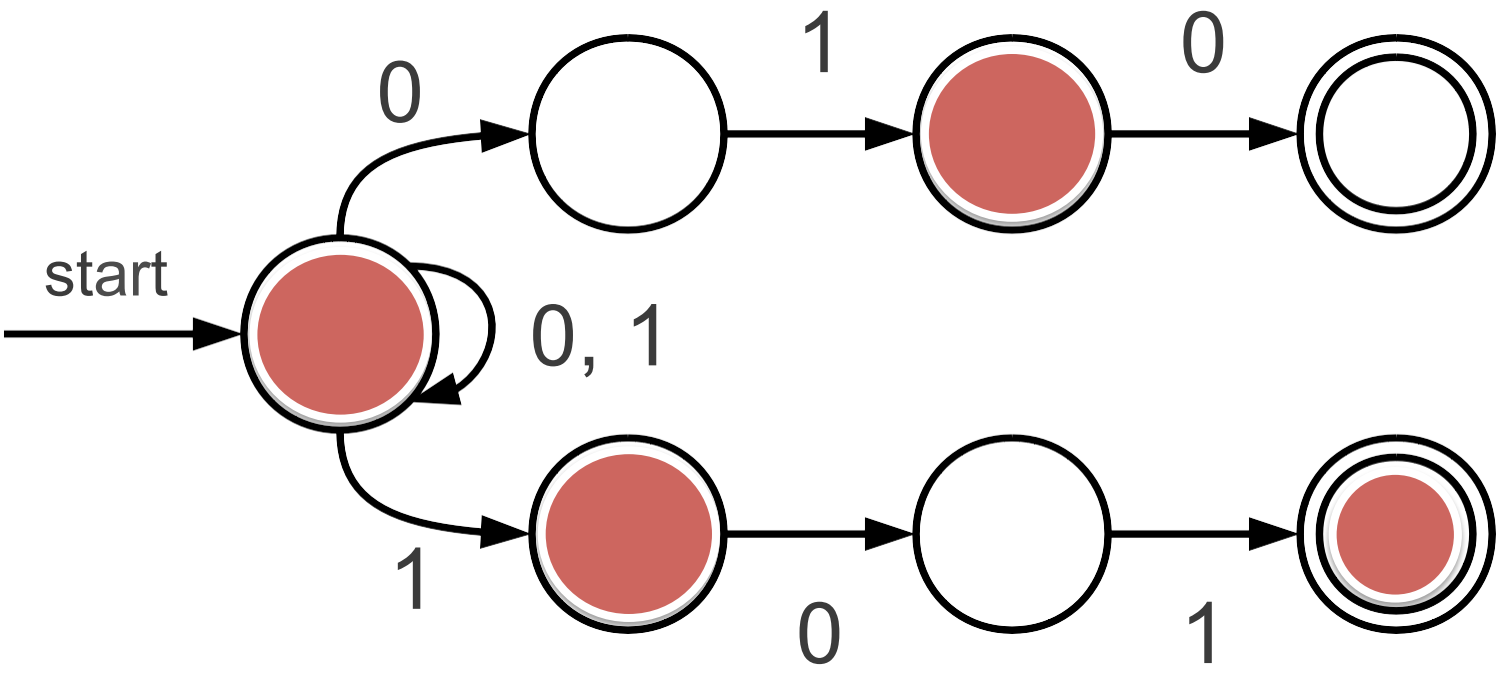
0 1 1 1 0 1

A. złożony, NAS



0 1 1 1 0 1

A. złożony, NAS



Ponieważ automat znalazł się w jednym ze stanów końcowych, więc słowo zostało zaakceptowane

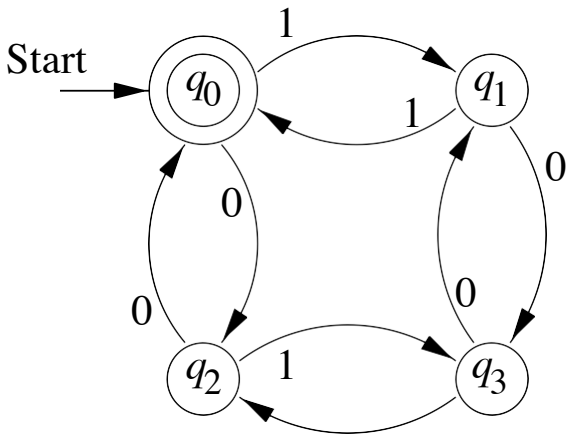
0 1 1 1 0 1

Rozszerzenie δ

- **Podstawa**. $\hat{\delta}(q, \varepsilon) = q$
- **Krok indukcyjny**. Jeśli $w=xa$, to $\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a)$
  Aby obliczyć $\hat{\delta}(q,w)$ obliczamy najpierw $\hat{\delta}(q,x)$ – czyli stan, w którym znajdzie się automat po przetworzeniu wszystkich symboli *w* poza ostatnim. Przypuśćmy, że stanem tym jest *p.* Wtedy dokonując przejścia ze stanu *p* na wejściu *a,* a więc ostatnim symbolu *w,* otrzymamy $\hat{\delta}(q,w)$, a zatem $\hat{\delta}(q,w) = \delta(p,a)$

Przykład DAS

- L={*w* | *w* ma parzystą liczbę zer i parzystą liczbę jedynek}
- Zadaniem DAS jest więc liczenie zer i jedynek modulo 2



	0	1
→*q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Język DAS

- Język DAS A=($Q, \Sigma, \delta, q_0, F$) oznaczamy przez L(A) i definiujemy następująco:

$$L(A) = \{w \mid \delta(q_0, w) \in F\}$$

- Język DAS A, L(A), to zbiór słów w przeprowadzających stan początkowy automatu q_0 w jeden ze stanów akceptujących (końcowych) F.
- Jeśli L=L(A) dla pewnego A to L nazywamy językiem regularnym.

Wyrażenia regularne (RE)

- Definicja: RE = wyrażenie regularne (*regular expression, regex*)
  - RE są to reguły definiujące słowa lub zbiory słów (łańcuchów) należących do języka formalnego Σ^*.
- Reguły składania dla x i **y** należących do reg:
  - konkatenacja (złożenie): **xy**
  - alternatywa: **x|y** lub **x+y**; czytaj **x** lub **y**
  - powtarzanie: **x***, x powtórzone zero lub więcej razy
- RE
  - każdy symbol alfabetu Σ
  - ε, pusty łańcuch
  - jeśli r i p są RE to również **(r), rp, r|p, r*** są RE
  - nic poza tym nie jest RE

Priorytet operatorów RE

- Priorytet = kolejność

(R)
R*
R_1R_2
$R_1|R_2$ lub **$R_1 + R_2$**

- Przykład: **ab*c|d** należy interpretować jako **((a(b*))c)|d**

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera 00 jako podłańcuch

$$(0 \mid 1)^*00(0 \mid 1)^*$$

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera 00 jako podłańcuch

(0 | 1)*00(0 | 1)*

110011
0000
01111011100111

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera tylko trzy symbole

(0|1)(0|1)(0|1)

011

110

111

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera tylko trzy symbole

(0|1){3}

011

110

111

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera przynajmniej jedno 0

1*00*1*

0111

11011

111011

Przykłady prostych RE

- Σ={0,1} – alfabet
- Łańcuch, który zawiera jedno 0

1*0?1*

0111

11011

111011

Zastosowania RE

- Σ={a, @, .}, gdzie a=dowolna litera
- Adresy e-mail

$$aa^* (.aa^*)^* @ aa^*.aa^* (.aa^*)^*$$

Zastosowania RE

- Σ={a, @, .}, gdzie a=dowolna litera
- Adresy e-mail

aa* (.aa*)* @ aa*.aa* (.aa*)*

piotr@gmail.com

piotr.kowalski@umcs.lublin.pl

Zastosowania RE

- Σ={a, @, .}, gdzie a=dowolna litera
- Adresy e-mail

aa* (.aa*)* @ aa*.aa* (.aa*)*

piotr@gmail.com

piotr.kowalski@umcs.lublin.pl

Zastosowania RE

- Σ={a, @, .}, gdzie a=dowolna litera
- Adresy e-mail

a+ (.a+)* @ a+.a+(.a+)*

piotr@gmail.com

piotr.kowalski@umcs.lublin.pl

Zastosowania RE

- Σ={a, @, .}, gdzie a=dowolna litera
- Adresy e-mail

a+ (.a+)* @ a+ (.a+)+

piotr@gmail.com

piotr.kowalski@umcs.lublin.pl

Zastosowania RE

- Σ={+, -, 0,...,9}
- Liczby nieparzyste ze znakiem

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(1|3|5|7|9)

-121

1157

+111115

Zastosowania RE

- Σ={+, -, 0,...,9}
- Liczby nieparzyste ze znakiem

(+|-)?[0123456789]*[13579]

-121

1157

+111115

Zastosowania RE

- Σ={+, -, 0,...,9}
- Liczby nieparzyste ze znakiem

(+|-)?[0-9]*[13579]

-121

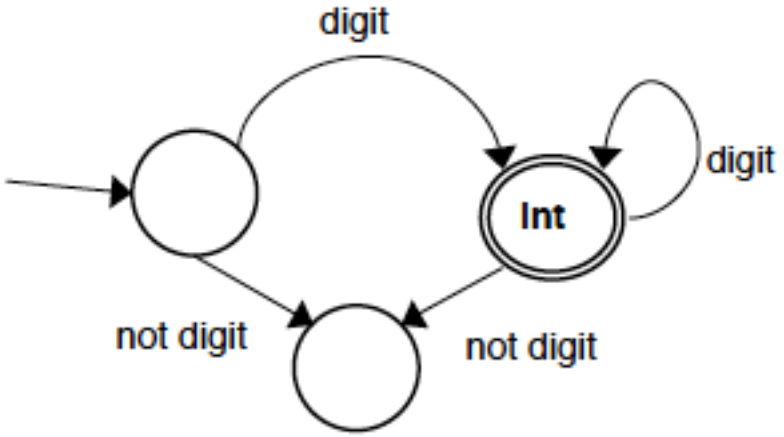
1157

+111115

Zastosowania RE

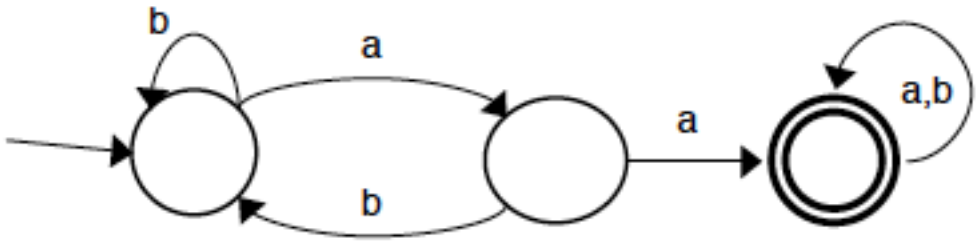
- Przykład. Σ ={a,b} – alfabet
  - łańcuchy zawierające tylko nieparzystą liczbę a: a(aa)*
  - łańcuchy zawierające aa: b*(aa)*b*

– RE [0-9]* realizuje następujący DAS skończony pokazany poniżej



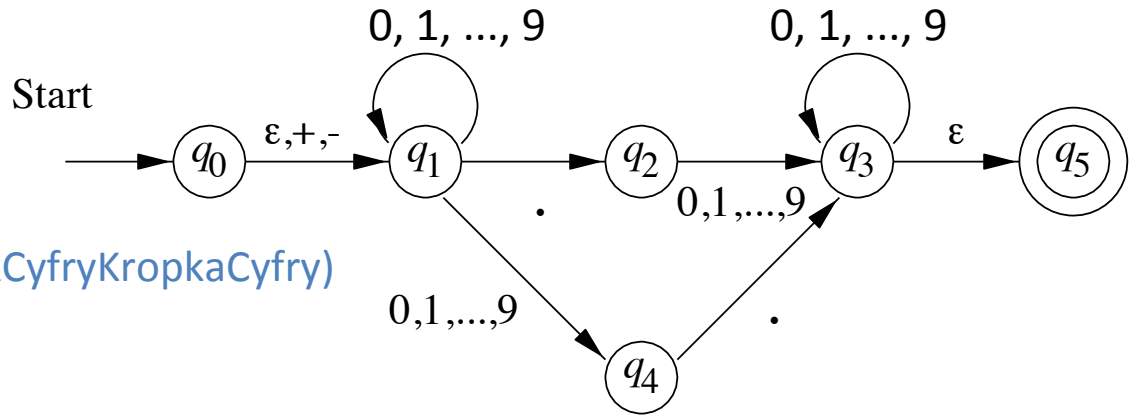
Zastosowania RE

- Zadanie: Podać RE, które realizuje automat skończony pokazany poniżej



Inne automaty skończone

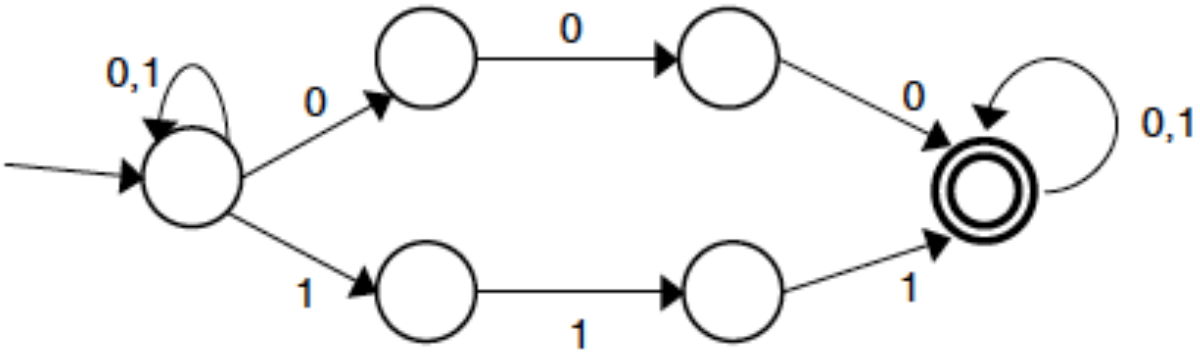
- NAS = niedeterministyczny automat skończony; może znaleźć się jednocześnie w wielu stanach wyjściowych
- ε-NAS = NAS reagujący na słowa puste
  - Przykład. ε-NAS akceptujący liczby dziesiętne postaci:
    - opcjonalny znak + lub -
    - ciąg cyfr
    - kropka
    - ciąg cyfr



Liczba=(ewZnakCyfryKropkaCyfry)

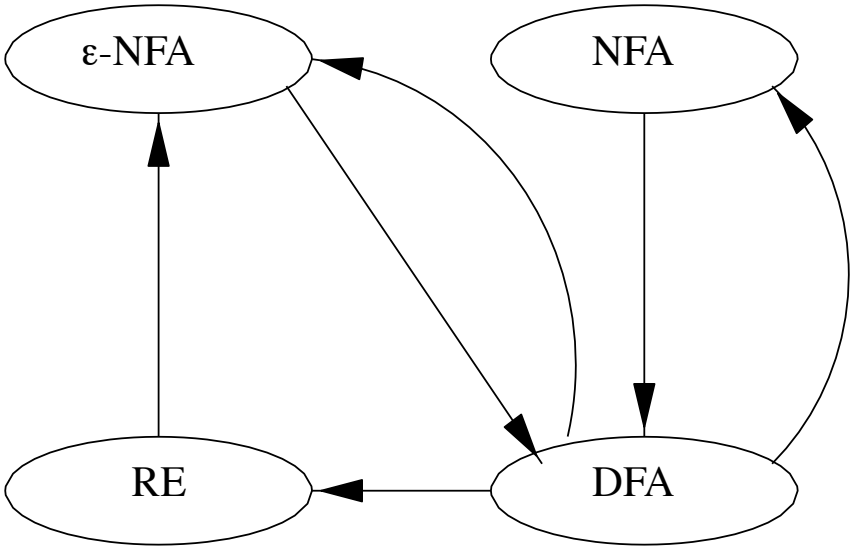
- NAS ze stosem
- inne

Równoważność NAS i RE



NAS akceptujący język RE
(0|1)*(000|111)(01)*

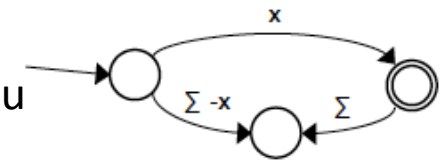
Równoważność automatów i RE



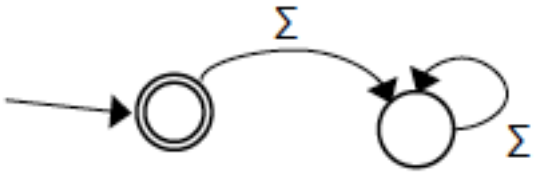
(Hopcroft i in.)

Reguły McNaughtona-Yamady-Thompsona

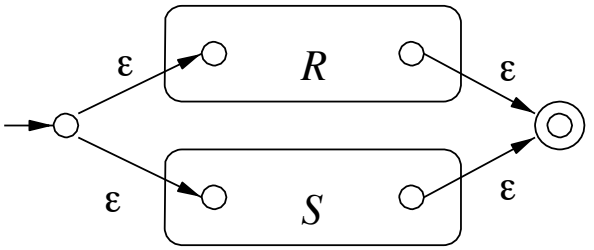
1. Ten NFA akceptuje jeden wybrany symbol alfabetu



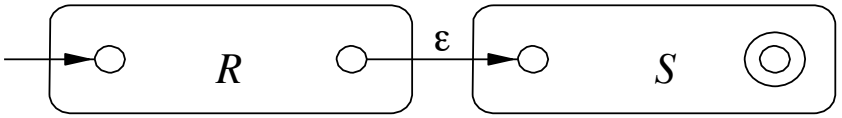
2. Ten NFA akceptuje tylko ε



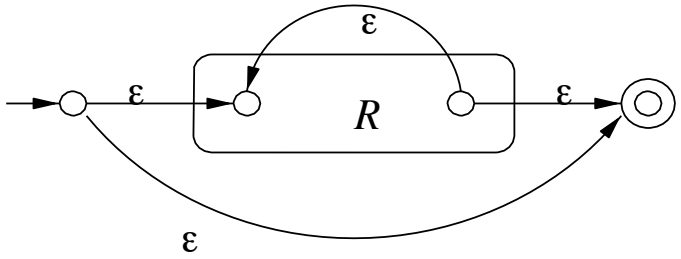
3. Ten ε-NFA akceptuje **r|s**



4. Ten ε-NFA akceptuje **rs**



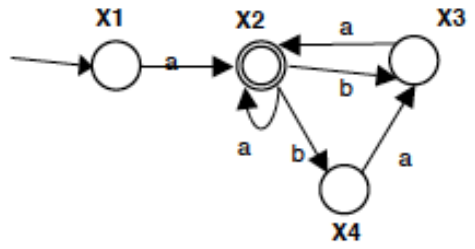
5. Ten ε-NFA akceptuje **r***

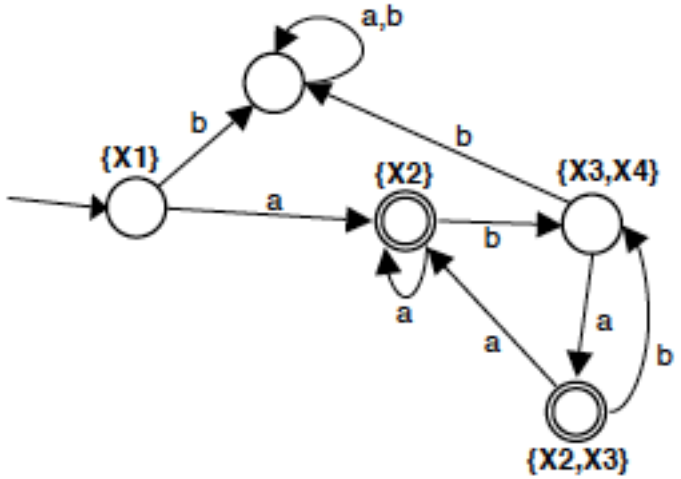


Od RE do DAS

- Używając podanych reguł (McNaughton-Yamada-Thompson; patrz: Aho+) możemy zbudować NAS z danych RE, następnie przekształcić NFA w DAS. Alfabet jest cały czas ten sam. Języki RE i tak zbudowanego DAS są identyczne (*Jak to zrobić dokładnie? Patrz np. Aho+* lub *Hopcroft+*)

Zadanie

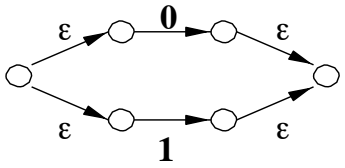




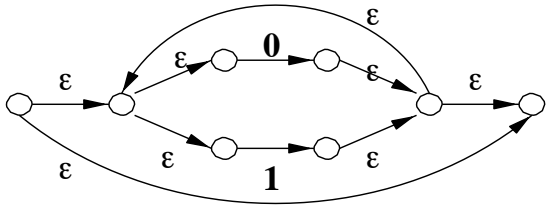
- Pokaż, że NAS pokazany obok (←) można zastąpić przez DAS pokazany obok, poniżej.
- Zapisz RE dla otrzymanego DAS

Przykład

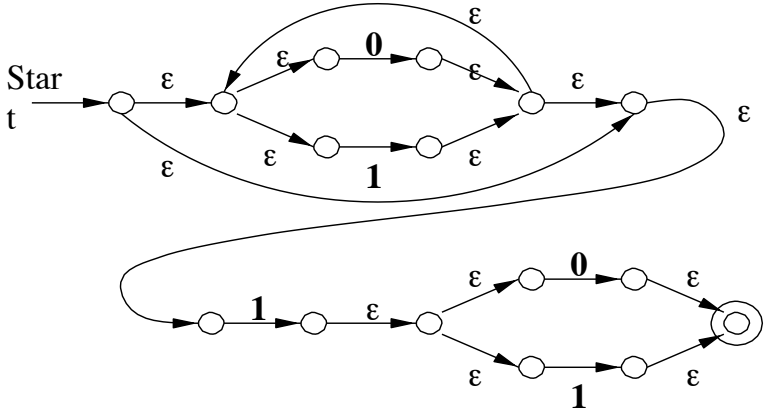
Zamiana wyrażenia regularnego **(0|1)*1(0|1)** na DAS z wykorzystaniem reguł Thompsona.



(a)



(b)



(c)

Wyrażenia regularne UNIXa

- UNIX

  . – dowolny znak

  * - dowolna liczba wystąpień

  ^ - początek wiersza

  $ - koniec wiersza

  \ - znosi znaczenie specjalne; zwykły znak

  [abc] – któryś ze znaków wypisanych

  \(...\) – zapamiętywanie ciągu w buforze

  \<, \> początek, koniec słowa

- Standard POSIX
  - klasy znaków w nawiasach [: :]

    [:alnum:], [:alpha:], [:blank:], [:cntrl:], [:digit:], [:graph:] – drukowalne

    [:lower:], [:print:], [:punct:], [:space:], [:upper:], [:xdigit:]

`lex, flex` – generator skanerów

- Proces przejścia od RE do automatów (pokazany krótko powyżej) jest wykorzystywany przez narzędzie `lex` lub `flex`, służące do budowania skanerów – programów do analizy leksykalnej innych programów, napisanych w jakimś języku, np. C++ lub Java.
- `flex` pobiera ciąg wyrażeń regularnych, opisujących dany język, buduje automat skończony oraz program w języku C służący jako analizator leksykalny programów ...

Przykład języka programowania

- Fortran (pierwszy, prymitywny język wyższego poziomu)
  - zmienne: dł. 5 znaków, pierwszy znak litera; duże litery. Zmienne o nazwach zaczynających się na `i, j, k, l, m, n` mają domyślnie typ całkowity, inne rzeczywisty;
  - Nie potrzeba deklarować zmiennych z wyjątkiem tablic. Tablice mogą być co najwyżej 3 wymiarowe.
  - Instrukcja podstawienia z lewej strony zawiera zmienną, z prawej wyrażenie
  - Struktury kontrolne
    - `bezwzględny skok:` GOTO <numer instrukcji>
    - `skok obliczany:` `GOTO (20,30,40,50) I` -- `I` może mieć wartość 1, 2, 3 ,4 – skok do instrukcji o numerze 20, 30, 40 lub 50
    - `IF (A+B-C/2) 10, 12, 14` -- skok do 10, 12, 14 w zależności od wartości w=A+B+C/2: w<0 -> 10, w=0 -> 12, w>0 -> 14
    - Instrukcja DO: `DO 21 K=1, N, 2` wykonuje się do instrukcji z numerem 21 łącznie; K zmienia się co 2 od 1 do N.
  - Brak podprogramów definiowanych przez użytkownika
  - Ignorowane są odstępy; np. **`DIMENSIONDANE(1234)`** `lub` **`D I M E N S I O N D A N E (1234)`** lub ładniej **`DIMENSION DANE(1234)`** -- deklaracja tablicy o długości 1234
  - Słów kluczowych można używać jako nazw np. `IF = 10` lub `DIMENSION DO(2)`

- Zadanie. NAPISZ schemat skanera☺ Pierwsze skanery to skanery ad hoc. Tutaj nawet RE, czy automaty skończone mają problem.

Literatura

- A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986; jest wydanie polskie.
- J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Reading MA: Addison-Wesley, 1979; jest wydanie polskie.

Co dalej?

- Character stream
- [Lexical Analyzer]
  - token stream
- [Syntax Analyzer]
  - syntax tree
- [Semantic Analyzer]
  - syntax tree
- [Intermediate Code Generator]
  - intermediate representation
- [Machine-Independent Code Optimizer]
  - intermediate representation
- [Code Generator]
  - target-machine code
- [Machine-Dependent Code Optimizer]
  - target-machine code