

<b>SPRAWOZDANIE</b>		<b>Data wykonania:</b> 16/11/2019
<b>Tytuł zadania:</b>	<b>Wykonał:</b>	<b>Sprawdził:</b>
<i>Automat</i>	<i>Kamil Sztandur 307354</i>	<i>dr inż. Konrad Markowski</i>

## Spis treści


Cel projektu .....	1
Teoria .....	1
Szczegóły implementacyjne .....	2
Sposób wywołania programu .....	3
Wnioski i spostrzeżenia .....	3

## Cel projektu

**Zadanie**  
 Napisać program symulujący działanie automatu skończonego  $M=(Q, \Sigma, \delta, q_0, F)$ , gdzie  
 $Q=\{q_0, q_1, q_2, q_3\}$ ,  $\Sigma=\{0, 1\}$ ,  $F=\{q_3\}$

$\delta$	Wejścia	
Stany	0	1
$q_0$	$q_1$	$q_2$
$q_1$	$q_3$	$q_0$
$q_2$	$q_0$	$q_3$
$q_3$	$q_1$	$q_2$

Symulator powinien dla ciągów składających się symboli wejściowych rysować diagram przejść i zaznaczać aktualny stan przy wczytywanych kolejnych symbolach z ciągu. Po wczytaniu całego ciągu program powinien wyświetlić komunikat czy ciąg został zaakceptowany?



Celem tego projektu było napisanie programu symulującego działanie automatu skończonego o określonym języku, podanych możliwych stanach oraz stanie końcowym. Wykonanie tego programu pozwala nie tylko lepiej zrozumieć działanie automatów skończonych, ale także podszkolić swoje umiejętności programowania z wykorzystaniem języka C.

## Teoria

Automat skończony to model matematyczny, system o dyskretnych wejściach i wyjściach, w którym mamy skończoną liczbę sygnałów o ograniczonej wartości, które możemy dać na wejście/wyjście. System taki może znajdować się w skończonej liczbie kontynuacji, czyli stanów. Stan w danym momencie stanowi

podsumowanie informacji dot. aktywnych poprzednich wejść, która jest potrzebna do określenia zachowania automatu przy następnym wejściu.

Oznacza to, że maszyna zaczynając od stanu początkowego czyta kolejne symbole danego słowa, zmieniając kolejno swój stan na stan będący wypadkową wartości przeczytanego symbolu oraz aktualnego stanu. Jeśli po przeczytaniu całego ciągu maszyna znajduje się w stanie oznaczonym jako końcowy, to ciąg zostaje zaakceptowany i oznacza to, że ten ciąg należy do języka regularnego, do którego rozpoznawania zbudowana jest dana maszyna.

Określamy ją poprzez równanie  $M = (Q, \Sigma, \delta, q_0, F)$ , gdzie  $Q$  oznacza zbiór możliwych stanów,  $\Sigma$  alfabet wejściowy,  $\delta$  funkcję przejścia,  $q_0$  stan początkowy, a  $F$  pożądany stan końcowy.

## Szczegóły implementacyjne

W celu wykonania takiej sytuacji należy napisać program, który jako wejście przyjmie skończony ciąg znaków. Następnie program policzy długość ciągu i utworzy odpowiednio duży wektor, do którego zacznie ten ciąg przepisywać, na bieżąco sprawdzając jego poprawność pod kątem występowania niedozwolonych znaków (nieznajdujących się w alfabecie automatu) oraz tego, czy po wykonaniu wszystkich przejść automat znajdzie się w pożądanym stanie końcowym.

W przypadku wykrycia błędu program natychmiast zwróci na ekran informację o przyczynie błędu – wskaże niedozwolony znak wraz z jego numerem na liście lub oznajmi, że po wykonaniu instrukcji automat nie znajdzie się w określonym stanie końcowym (tutaj  $q_3$ ). Następnie przerwie działanie. Natomiast po pozytywnej weryfikacji ciągu program rozpocznie przechodzenie między kolejnymi stanami zgodnie z podanym ciągiem oraz własnymi wewnętrznymi instrukcjami funkcji przejścia.

Logi ze swojego działania będzie wypisywać na ekran w czasie rzeczywistym. Po wykonaniu całej instrukcji program zakończy działanie.

### Weryfikacja kodu:

```
for( int i = 0; i < seriesLength; i++) {
    switch( loadedChar = fgetc( input ) ) {
        case '0':
            code[i] = 0;
            testState = changeCurrentState( 0, testState);
            break;

        case '1':
            code[i] = 1;
            testState = changeCurrentState( 1, testState);
            break;

        default:
            seriesValid = 0;
            return seriesInvalidCheck( argv[0], loadedChar, i+1);
    }
}
if ( testState != finalState )
    return seriesInvalidCheck( argv[0], 0, 0);
else printf("%s: Ciąg został zaakceptowany\n", argv[0]);
fclose( input );
```

Weryfikacja kodu odbywa się za pomocą pętli for oraz instrukcji switch, która przepisuje 0 lub 1 do kolejnych komórek utworzonego wcześniej wektora. W przypadku napotkania jakiegokolwiek innego znaku oznacza ciąg jako nieprawidłowy, a następnie zwraca błędny znak oraz jego pozycję w ciągu do funkcji wypisującej błędy i kończącej program. Instrukcja ta wykonuje także test, czy obecny ciąg znaków doprowadzi automat do stanu końcowego i zwraca błąd, jeżeli tak się nie stanie.

## Funkcja obsługująca błędy:

```
int seriesInvalidCheck(char *name, char invChar, int position ){
    printf("%s: Ciąg nie został zaakceptowany, ponieważ ", name);

    if( seriesValid != 1 ) printf("pojawily sie w nim niedozwolone znaki:\nZnak nr. %d: '%c'\n", position, invChar);
    else printf("koncowy stan nie jest rowny q3.\n");

    return 1;
}
```

## Przejścia między stanami:

```
int changeCurrentState(int input, int state) {
    int newState;

    switch(state) {
        case 0:
            if( input == 1)
                newState = 2;
            else newState = 1;
            break;
        case 1:
            if( input == 1)
                newState = 0;
            else newState = 3;
            break;
        case 2:
            if( input == 1)
                newState = 3;
            else newState = 0;
            break;
        case 3:
            if( input == 1)
                newState = 2;
            else newState = 1;
    }
    return newState;
}
```

Funkcja przejścia **changeCurrentState** na podstawie aktualnego stanu „state” oraz wejścia „input” (1 lub 0) zwraca informację, na który stan powinna przejść maszyna. W momencie odczytywania już zweryfikowanego kodu z wektora wywołuje się ją poprzez np. po odczytaniu „0”:

```
history[i+1] = currentState = changeCurrentState( 0, currentState);

printf("q0");
for(int i = 1; i < seriesLength+1; i++) {
    if( history[i] < 0 ) break;
    printf(" - %d - q%d", code[j], history[i] );
    j++;
}
printf("\n");

j = 0;
break;
```

Gdzie wektor history[] odpowiada tu za przechowywanie historii stanów i wyświetlanie na ekran logów działania programu, a code[] ciągu wejściowego. Pętla for wypisuje za każdym przejściem dotychczasową historię przejść, tworząc logi w kształcie piramidy. Wcześniej wypełniłem wektor history[] ujemnymi liczbami, aby ułatwić pętli for określenie, czy dotarła już do końca zapisanych kolejnymi stanami komórek na chwilę obecną.

## Sposób wywołania programu

Aby poprawnie wywołać program należy uruchomić go poprzez wpisanie do terminalu wyrażenia „./automat przyklad.txt”, gdzie w miejsce ‘przyklad.txt’ możemy wpisać nazwę dowolnego notatnika tekstowego zawierającego interesujący nas ciąg znaków. Oto reakcja programu na kod ‘110111’ w notatniku przyklad.txt:

```
sztanduk@jimp:~/PINF_proj$ ./automat przyklad.txt
./automat: Ciąg został zaakceptowany
q0 - 1 - q2
q0 - 1 - q2 - 1 - q3
q0 - 1 - q2 - 1 - q3 - 0 - q1
q0 - 1 - q2 - 1 - q3 - 0 - q1 - 1 - q0
q0 - 1 - q2 - 1 - q3 - 0 - q1 - 1 - q0 - 1 - q2
q0 - 1 - q2 - 1 - q3 - 0 - q1 - 1 - q0 - 1 - q2 - 1 - q3
```

Nasz ciąg znaków powinien być słowem składającym się jedynie z zer oraz jedynek. Każdy inny znak (w tym znaki białe) unieważnia nasz ciąg. Program nie obsługuje wejścia stdin i w przypadku próby skorzystania z niego zwraca błąd z odpowiednią informacją. Reakcja programu na błędny kod, zawierający nieobsługiwany znak '5':

```
sztanduk@jimp:~/PINF_proj$ ./automat zlykod1.txt
./automat: Ciąg nie został zaakceptowany, ponieważ pojawiły się w nim niedozwolone znaki:
Znak nr. 5: '5'
```

## Wnioski i spostrzeżenia

Napisanie tego programu było dla mnie interesującym oraz dydaktycznym doświadczeniem. Pozwolił mi on lepiej poznać schemat działania automatu skończonego oraz rozszerzyć swoje umiejętności programowania w języku C m.in. o funkcję goto oraz dokładniejsze i wygodniejsze sposoby obsługi błędów.

Znaczącą trudność w tym programie sprawiła mi implementacja funkcji wejścia stdin, która w obecnej formie programu, niestety, pozostała poza moimi możliwościami. Oczywiście, mogłem zrealizować ją poprzez narzucenie na użytkownika szeregu ścisłych wymagań, ale ostatecznie uznałem, że bezpieczeństwo działania programu oraz czytelność kodu (a więc otwartość na przyszłe rozwiązanie tego problemu) jest w tej chwili najważniejsze. Przez długi czas nie miałem pomysłu na napisanie funkcji wypisującej kolejne przejścia między stanami w formie piramidy. Początkowo zorganizowałem to w formie dopisywania kolejnych słów do jednego długiego ciągu znaków, ale uznałem że jest to bardzo niechlujne rozwiązanie i wpadłem na pomysł napisania tego poprzez kreatywne wypisywanie wartości z dwóch wektorów (history[] oraz code[]) na raz.

Jestem zdecydowanie zadowolony z finalnego działania programu oraz jego wyglądu zarówno od strony użytkownika jak i programisty. Program wyświetla dane użytkownikowi w formie piramidy dokładnie tak jak powinien. Po przeczytaniu książki „Czysty Kod” Roberta C. Martina zdaję sobie sprawę z wagi czytelności i przejrzystości kodu, a zarazem mnóstwa korzyści za nią idących. Do tego uważam, że całkiem sprytnie oraz sprawnie zaprogramowałem obsługę błędów, którą niemal całkowicie załatwia za mnie użycie instrukcji switch wraz z krótką funkcją „seriesInvalidCheck” wypisującą komunikat zależnie od zwróconego błędu i kończącą działanie programu.