Laboratorium programistyczne: Sumator i zastosowania automatów

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2019-05-09

1 Systemy liczbowe

Liczby mogą być zapisywane w różny sposób. Istnieją systemy addytywne (np. rzymski), w których istotna jest ilość powtórzeń danego elementu oraz systemy pozycyjne o różnych podstawach (np. system dziesiętny), w których istotne jest miejsce w którym znajduje się dany element.

W życiu codziennym najczęściej spotykamy się z zapisem dziesiętnym, funkcjonującym następująco: $5731 = 10^0 \cdot 1 + 10^1 \cdot 3 + 10^2 \cdot 7 + 10^3 \cdot 5$.

1.1 System dwójkowy

Ze względu na sposób budowy elektroniki cyfrowej i komputerów w informatyce dużo częściej spotykamy się z systemem dwójkowym (oraz systemami łatwo rozkładającymi się na dwójkowy, np. szesnastkowym).

Pojedynczą cyfrę systemu dwójkowego (przybierającą wartość 0 albo 1) określa się mianem *bit*u, liczby reprezentowane są jako ciągi takich cyfr. Terminem *bajt* określa się zazwyczaj ciąg o długości 8 bitów (ale w niektórych systemach ciąg o innej długości).

Podstawowym sposobem zapisy liczb całkowitych nie ujemnych w systemie dwójkowym jest *naturalny kod binarny* (*NKB*), w którym np. 4 bitowy ciąg $\mathtt{a_3a_2a_1a_0}$ reprezentuje liczbę $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + 2^3 \cdot a_3$. Zwróć uwagę na podobieństwo do systemu dziesiętnego.

Podstawowym sposobem zapisy liczb całkowitych (ze znakiem) jest *kod uzupełnień do dwóch* (*U2*) w którym n-bitowa liczba reprezentowana przez ciąg $\mathtt{a_{n-1}} \ldots \mathtt{a_3a_2a_1a_0}$ będzie miała wartość $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + ... + 2^{n-2} \cdot a_{n-2} - 2^{n-1} \cdot a_{n-1}$. Jako że najstarszy bit wchodzi z ujemną wagą, jego ustawienie na 1 oznacza liczbę ujemną (ale nie jest to kod znaku). Warto zauważyć kompatybilność z NKB.

Liczby zapisywane w tych kodowaniach systemu dwójkowego oznacza się często przy pomocy prefiksu ”0b” albo sufiksu ”b” (w Pythonie możemy stosować jedynie zapis z prefiksem), np. `0b101 = 101b` reprezentuje liczbę 5 w systemie dziesiętnym ($2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 = 5$).

2 Sumator

Cyfrowe układy elektroniczne (także te składające się na nasze komputery) każdą liczbę traktują jako ciąg logicznych jedynek i zer (wartości True i False). Spróbujemy zasymulować takie działanie w Pythonie i zastanowić się nad tym jak można zrealizować dodawanie takich liczb.

Jeżeli znamy i pamiętamy jeszcze metodę dodawania „w słupku” („pod kreską”) to wiemy, że dodawanie dwóch n cyfrowych liczb możemy potraktować jako ciąg n dodawań dwóch liczb jednocyfrowych z obsługą przeniesienia. Podobnie można postąpić przy sumowaniu liczb binarnych. Potrzebujemy zatem elementu, który będzie sumował 3 wartości logiczne (dwie pochodzące z sumowanych liczb, jedna z przeniesienia z poprzedniego elementu) i generował wynik sumy oraz wartość przeniesienia dla następnego elementu.

Zadanie 2.0.1

Zastanów się w jaki sposób, korzystają z poznanych funkcji logicznych (and, or, xor, not) można obliczyć wartość sumy dwóch cyfr binarnych a i b oraz wartość przeniesienia.
Wskazówka: zapisz tablicę prawdy dla tej operacji

Zadanie 2.0.2

Spróbuj rozszerzyć poprzednie rozwiązanie, tak aby uwzględniać w sumowaniu przeniesienie z poprzedniego elementu.
Wskazówka: rozszerz tablicę prawdy o jedną kolumnę wejściową

Zadanie 2.0.3

Napisz funkcję realizującą sumator. Funkcja powinna przyjmować 3 argumenty logiczne (wartości dwóch bitów do zsumowania oraz wartość przeniesienia) i zwracać wartość sumy i przeniesienia do następnego elementu.
Wskazówka: zwracanie dwóch elementów z funkcji najprościej zrealizować poprzez zwracanie dwu elementowej listy.

Liczba 6 posiada zapis binarny 0b110. Czyli jeżeli chcielibyśmy przedstawić ją w postaci listy wartości logicznych byłoby to [0, 1, 1]. Zwróć uwagę na zmianę kolejność bitów: wynika ona z tego że w zapisie list element o indeksie zero podajemy jako pierwszy a w zapisie binarnym liczby bit zerowy (wchodzący z wagą 2^0) jest jako ostatni. Dzięki takiemu zapisaniu liczb możemy użyć naszego sumatora do dodania wielo-bitowych liczb.

Zadanie 2.0.4

Napisz funkcję wykorzystującą sumator stworzony w zadaniu 2.0.3 do obliczania sumy dwóch liczb reprezentowanych jako listy wartości logicznych. Na przykład dla argumentów `[True, False, True]`, `[False, True, True]` (odpowiadających liczbom 5 i 6) wynikiem powinna być lista `[True, True, False, True]` (odpowiadająca liczbie 11).
Rozwiązanie powinno działać poprawnie dla liczb o dowolnej ilości bitów, także w przypadku gdy liczba bitów poszczególnych liczb jest różna. Rozwiązania działające tylko dla liczb o równej lub ustalonej liczbie bitów mogą otrzymać maksymalnie 2pkt.
Wskazówka: dodawanie elementu na koniec listy możliwe jest z użyciem metody `append`*, można też zedefiniować listę o zadanej ilości elementów np.* `5 * [False]` *i tylko modyfikować wybrane elementy.*

3 Automaty skończone

Już najprostszy z poznanych typów automatów, czyli automat skończony, znajduje szerokie zastosowania praktyczne m.in. w informatyce i elektronice. Model automatu używany jest m.in. do opisu działania niektórych protokołów komunikacyjnych, czy też algorytmów postępowania, a także do opisu i realizacji elektronicznych układów logicznych.

3.1 opisy algorytmów

W zastosowaniach informatycznych najczęściej:

- przejścia związane są z reakcją na działania użytkownika lub klienta
- wykonaniu przejścia do nowego stanu towarzyszy też wykonanie jakiejś akcji (np. wysłanie odpowiedzi do klienta, zapamiętanie jakiś dodatkowych informacji)
- stan automatu związany jest z stanem wewnętrznym realizowanego algorytmu

3.1.1 Typowe sposoby implementacji automatów

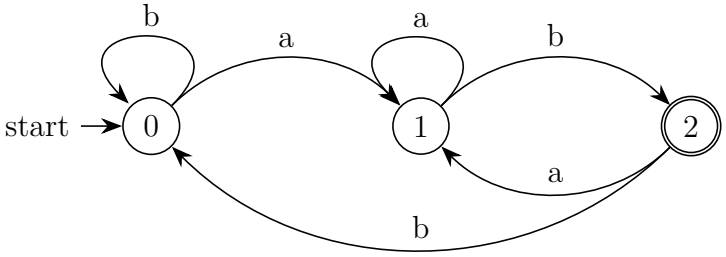
Najbardziej klasycznym przykładem implementacji algorytmu w postaci automatu jest (zazwyczaj nieskończona) pętla w ramach której:

- wczytujemy kolejne porcje danych odpowiedzialne za wywoływanie kolejnych przejść pomiędzy stanami automatu,
- z użyciem instrukcji warunkowej `if / else` (lub nie występującej w Pythonie instrukcji wyboru `switch / case`) w oparciu o aktualny stan automatu wybieramy stosowne postępowanie,
- w wyniku wykonanych operacji ustalamy nowy stan automatu.

```
while True:
    x = pobierzDane()
    if stan == "A":
        stan = dzialaniaA(x)
    elif  stan == "B":
        stan = dzialaniaB(x)
    ...
```

Zadanie 3.1.1

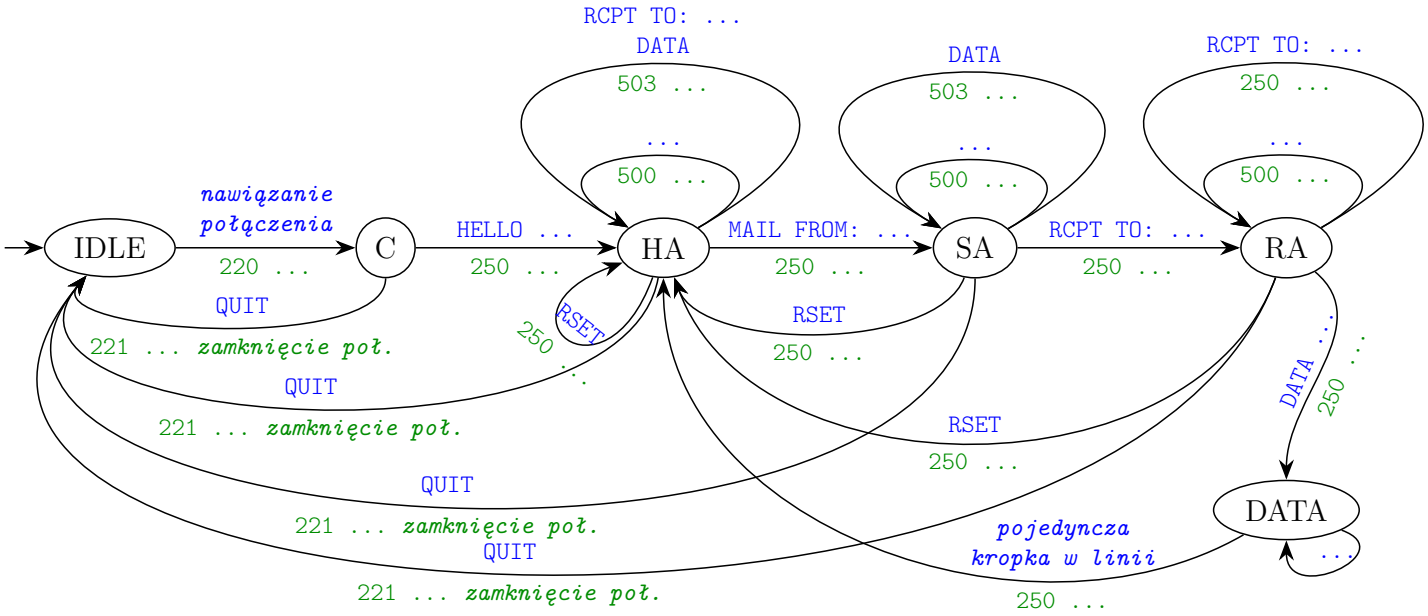
Zaimplementuj automat sprawdzający czy podane słowo należy do języka nad alfabetem słów kończących się ciągiem `ab`:



Wskazówka: zamiast ***`while True`*** *możesz użyć pętli iterującej po literach słowa.*

3.1.2 SMTP

Oficjalna specyfikacja protokołu przesyłu poczty elektronicznej SMTP (Simple Mail Transfer Protocol) czyli RFC 821 nie definiuje tego protokołu bezpośrednio z użyciem automatu skończonego. Jednak możliwe jest opisanie go w taki sposób. Poniżej znajduje się graf automatu reprezentujący typową sesję protokołu SMTP (nie jest to pełen opis protokołu ani jego działania - pominięte zostały niektóre z obowiązkowych komend protokołu oraz zagadnienia weryfikacji).



Poszczególne stany oznaczają:

- IDLE – oczekiwanie na połączenie
- C – nawiązane połączenie w warstwie niższej (TCP)
- HA – otrzymano i zapamiętano informację o adresie klienta przekazanym w HELO (jest to nazwa

hosta lub nazwa domenowa, ale nie adres poczty)

- SA – otrzymano i zapamiętano informację o adresie e-mail nadawcy
- RA – otrzymano i zapamiętano informację o adresach e-mail odbiorców (kolejne wejścia do tego stanu powodują dodanie kolejnego odbiorcy do listy)
- DATA – odbieranie treści maila (zakończenie przy pomocy linii złożonej wyłącznie z kropki, skutkuje zakolejkowaniem listu do wysyłki lub jego wysłaniem)

Powyżej strzałek reprezentujących przejścia pomiędzy stanami (kolor niebieski) podane zostały działania i polecenia wysyłane przez klienta wywołujące dane przejście, zaś poniżej (kolor zielony) podane zostały odpowiedzi serwera. W miejscu ... występuje tekst stanowiący dane (np. adres nadawcy lub odbiorcy), komunikat związany z podanym kodem odpowiedzi lub dowolną, inną od wymienionych komendę. Zapis przykładowej sesji protokołu SMTP opisywanej powyższym automatem:

```
220 dragon.icm.edu.pl ESMTP Exim 4.89 Fri, 26 Apr 2019 13:14:42 +0000
HELO test.example.com
250 dragon.icm.edu.pl Hello test.example.com [2001:6a0:0:21::60:13]
MAIL FROM: rrp@dragon.icm.edu.pl
250 OK
RCPT TO: rrp@dragon.icm.edu.pl
250 Accepted
DATA
354 Enter message, ending with "." on a line by itself
to jest e-mail, ale bez standardowych naglowkow ...
wiec bedzie dziwnie wygladal w programie klienckim ...
.
250 OK id=1hK0hn-0008FU-OV
QUIT
221 dragon.icm.edu.pl closing connection
```

Zadanie 3.1.2

Napisz symulator serwera SMTP w postaci wyżej przedstawionego automatu. Program powinien przyjmować kolejne komendy od użytkownika i stosownie na nie reagować.

Wskazówka: do wczytywania kolejnych linii w ramach pętli głównej automatu możesz użyć: `linia = sys.stdin.readline().rstrip()`. *Wymaga to wcześniejszego zaimportowania moduły sys poprzez:* `import sys`.
Dzięki użyciu metody `rstrip()` *wczytana linia nie będzie zawierała kończącego ją znaku nowej linii (ani innych białych znaków na końcu).*

3.2 elektronika – układy logiczne

3.2.1 opis automatu przy pomocy tablicy prawdy

Jak pamiętamy z zajęć o sumatorze elektronicy lubią opisywać układy cyfrowe z użyciem tablic prawdy. W taki sposób można opisać także dowolny automat skończony. W tym celu numerujemy (binarnie) stany automatu oraz pobudzenia wywołujące przejścia pomiędzy stanami (elementy alfabetu), numery te będziemy nazywać wektorami stanu i wektorami wejść. Następnie tworzymy tablicę prawdy postaci:

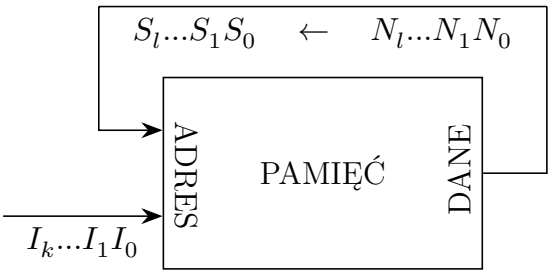
S_l	...	S_1	S_0	I_k	...	I_1	I_0	N_l	...	N_1	N_0

gdzie $S_l...S_1S_0$ to wektor stanu (czyli l bitowy numer stanu), $I_k...I_1I_0$ to wektor wejść (czyli l bitowy numer pobudzenia wywołującego zmianę stanu), a $N_l...N_1N_0$ to l bitowy numer stanu do którego ma przejść automat znajdujący się w stanie określonym w kolumnach $S_l...S_1S_0$, pod wpływem pobudzenia

podanego w kolumnach $I_k...I_1I_0$. Wadą takiego opisu jest bardzo szybko rosnąca liczba wierszy w tabeli. Wynosi ona 2^{l+k}, czyli np. dla 3 stanowego automatu z 2 bitowym wejściem jest ich już 32.

3.2.2 realizacja pamięciowa

Opis taki natomiast umożliwia bardzo prostą implementację w postaci „*pamięciowej*". Używając $l+k$ komórek pamięci, adresowanych ciągiem bitowym $S_l...S_1S_0I_k...I_1I_0$, możemy w każdej z nich przechowywać po prostu numer następnego stanu (w związku z tym muszą to być co najmniej l bitowe komórki pamięci). Ideę takiej realizacji automatu przedstawia poniższy schemat:



Ciekawostka

Pamięć o n bitowym rozmiarze pojedynczej komórki (i szyny danych) oraz o m bitowej przestrzeni adresowej (i nie mniejszej szynie adresowej) stanowi bardzo prostą realizację bramki logicznej mogącej realizować dowolną funkcję logiczną. Dodatkowo funkcja ta może być programowo zmieniana. Własność ta jest powszechnie wykorzystywana w współczesnej elektronice i leży u podstaw działania układów o programowalnej strukturze, takich jak FPGA.

Zadanie 3.2.1

Czy pamięć o wielkości 8kB, złożona z komórek o wielkości 8bitów (i używająca takiej szerokości szyny danych), posiadająca 16 bitową szynę adresową[a] umożliwia realizację (w przedstawiony powyżej sposób) automatu o 8 stanach i 4 bitowym wejściu? Odpowiedź krótko uzasadnij.

a. W przypadku gdy szyna adresowa udostępnia większą przestrzeń adresową niż rozmiar pamięci, najstarsze bity adresu są ignorowane. Oznacza to że w przypadku tej pamięci odwołanie do adresu $8\text{k}+x$ (dokładnie $8*1024+x$), jest odwołaniem do adresu x.

3.2.3 realizacja w postaci funkcji logicznej

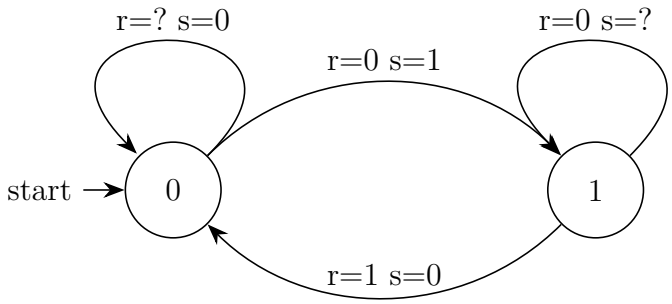
Jak wiemy (z zajęć o sumatorze) opis układu przy pomocy tablicy prawdy pozwala na jego realizację z użyciem standardowych funkcji logicznych (bramek and, or, not, xor). W ten sposób możemy także realizować automaty. Analogicznie jak na powyższym schemacie realizacji *pamięciowej* będzie musiało wystąpić podanie sygnału z wyjścia bramek na ich wejście (sprzężenie zwrotne).

Realizacja taka jest bardziej oszczędna pod względem zasobów od realizacji *pamięciowej*, jednak bardziej wymagająca na etapie projektowania – trzeba dokonać konwersji tablicy prawdy na funkcje logiczne. Konwersji takiej dokonujemy niezależnie dla każdego bitu nowego wektora stanu ($N_l...N_1N_0$), czyli musimy skonwertować l tablic prawdy na l funkcji logicznych, każda w ogólności[1] o $l+k$ argumentach.

1. część argumentów może okazać się nieistotna dla obliczania danego N_x – nie wpływać na jego wartość

Zadanie 3.2.2

Przyjrzyj się poniższemu automatowi (y=x oznacza "gdy y ma wartość x", ? – dowolna wartość).



Stan	r	s	Nowy Stan
0	0	0	0
0	1	0	0
0	0	1	1
1	0	0	1
1	1	0	0
1	0	1	1

1. Czy zauważyłeś coś nietypowego?
2. Spróbuj znaleźć wyrażenie logiczne realizujące ten automat.
3. Zaimplementuj funkcję realizującą ten automat i użyj je do wypisania tablicy prawdy, celem weryfikacji poprawności działania.

3.2.4 asynchroniczne i synchroniczne

Omówione do tej pory automaty charakteryzowały się brakiem jakiegokolwiek mechanizmu ustalania kiedy powinny odczytać stan wektora wejściowego i wykonać związane z nim przejścia. Układy tego typu (które reagują na zmiany wejścia zachodzące w dowolnym momencie) nazywamy asynchronicznymi.

Niestety taka prosta realizacja napotyka kilka problemów:

- Automat nie uzyskuje żadnej informacji o tym, iż wektor wejściowy jest nowy (podana została nowa litera analizowanego słowa). W efekcie, jeżeli litera v odpowiada za przejścia z A do B i z B do C, to automat na skutek podania jej na wejście może przeskoczyć z A do C zanim zostanie podana inna litera. Aby tego uniknąć każdy wektor wejściowy powodujący wejście do danego stanu powinien powodować pozostawanie w nim.
- Zmiana kilku bitów wektora wejściowego praktycznie nigdy nie będzie równoczesna. W efekcie, jeżeli np. jesteśmy w stanie A (z którego możemy przejść pod wpływem 01 do B, 10 do C, 11 do D) i następuje zmiana wektora wejściowego z 00 na 11 nie możemy przewidzieć do którego stanu przejdziemy (B, C czy może D). Aby uniknąć negatywnych skutków takiego działania automat taki powinien ze stanów B i C przechodzić do D pod wpływem wektora 11.

Ciekawostka

Mianem układów asynchronicznych określane mogą być nie tylko automaty, ale też inne układy elektroniczne. Nie wszystkie z omawianych problemów dotyczą wszystkich układów asynchronicznych. Duża część tych problemów związana jest z faktem sprzężenia zwrotnego, z którym mamy do czynienia w konstrukcji automatu.

Pomimo tych utrudnień w konstrukcji automaty asynchroniczne są spotykane w praktyce (zauważ, że wcześniej także skonstruowaliśmy poprawny automat asynchroniczny – przerzutnik RS). Jednak ze względu na te problemy dużo częściej stosowane są automaty synchroniczne.

W automatach synchronicznych występuje dodatkowy sygnał zegarowy, służący do synchronizacji odczytu wejść i wykonywania przejść. Można powiedzieć, że informuje on o tym, iż na wejściu przygotowana została kolejna litera analizowanego słowa i można wykonać związaną z nią zmianę stanu automatu.

Ciekawostka

W naturalny sposób prowadzi to do stosowania w realizacji automatów synchronicznych, jako komórek pamiętających poszczególne bity wektora stanu przerzutników typu D. Przerzutniki te zapamiętują podawaną na nie informację w momencie narastającego bądź opadającego zbocza zegara i przechowują ją (oraz wystawiają na swoim wyjściu) do nadejścia kolejnego zbocza. Ze względu na sposób działania pamięci, implementacje automatów oparte na niej na ogół w naturalny sposób są automatami synchronicznymi – funkcję zegara pełnią sygnały sterujące wykonaniem operacji odczytu.

Zwróć uwagę, iż nasze implementacje automatów w Pythonie są dużo bliższe automatom synchronicznym niż asynachronicznym – nie mamy typowego okresowego sygnału zegarowego, ale jego funcję pełni zatwierdzenie wprowadzonych danych przy pomocy znaku nowej linii w `readline()`, bądź jawne wywołanie funkcji.

3.2.5 automaty Moore'a i Mealy'ego

W praktycznych zastosowaniach chcemy aby działanie układu elektronicznego realizującego dany automat objawiało się nie tylko zmianą stanów wewnętrznych automatu, ale przede wszystkim zmianą stanów jakiś wyjść. Można zostać to zrealizowane na jeden z dwóch sposobów:

- Możemy ściśle powiązać stan wyjścia z stanem w którym się znajduje automat. W tym celu ustalamy funkcję logiczną przekształcającą wektor stanu na stan wyjść z nim związany (wektor wyjściowy). Ten typ automatu określamy mianem automatu Moore'a. W niektórych przypadkach możliwe jest takie zakodowanie automatu w taki sposób aby wektor stanu lub jego fragment stanowił bezpośrednio wektor wyjściowy (lub jego fragment). Jeżeli automat realizujemy w postaci *pamięciowej* to wektor wyjściowy może być przechowywany w tych samych komórkach pamięci co wektor stanu.
- Alternatywnie przy ustalaniu wartości wyjść oprócz bieżącego stanu automatu możemy także uwzględniać wartość wejść (pobudzenia, które spowodowało ostatnie przejście). Ten typ automatu określamy mianem automatu Mealy'ego. Często pozwala on na zmniejszenie ilości stanów automatu, kosztem bardziej rozbudowanej logiki wyjściowej.

Zadanie 3.2.3

Przypomnij sobie automat akceptujący słowa kończące się na ab z zadania 3.1.1. Dodaj do tego automatu jedno bitowe wyjście informujące o tym że aktualnie wprowadzone słowo zostało zaakceptowane. Możesz w tym celu wybrać rozwiązanie automatu Moore'a albo Mealy'ego. Wybór krótko uzasadnij. Zapisz tablicę prawdy dla tego automatu i zasymuluj jego działanie w postaci pamięciowej. *Wskazówka: do zasymulowania pamięci możesz użyć słownika w którym kluczami są stałe listy*[a] *reprezentujące wektor stanu i wektor wejściowy, a wartościami reprezentują nowy wektor stanu.*

a. Stałą listę (nazywaną *krotką*) zapisujemy używając nawiasów okrągłych zamiast kwadratowych, np. `(1, 2, 3)`. Możemy konwertować zwykłe listy na krotki przy pomocy `tuple()` i krotki na listy przy pomocy `list()`

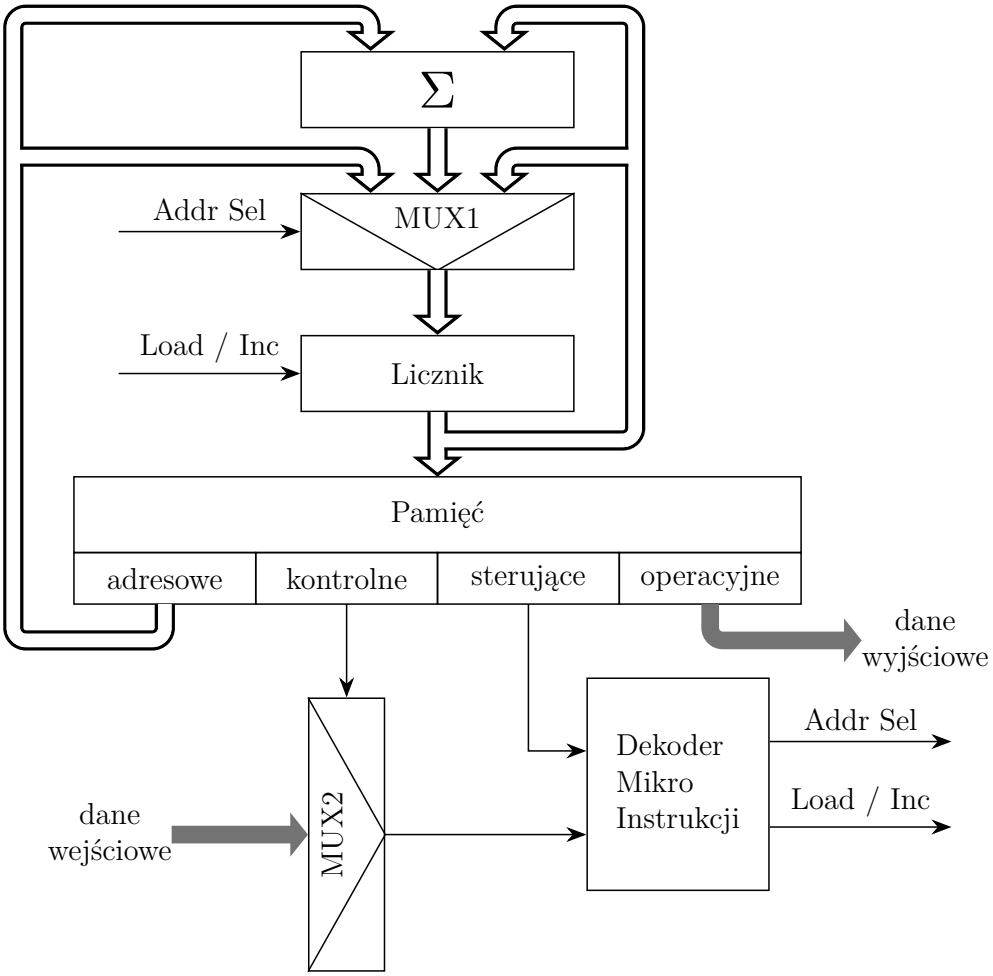
Zadanie 3.2.4

Zaimplementuj automat z zadania 3.2.2 w postaci pamięciowej, jako automat z jednobitowym wyjściem. Możesz w tym celu wybrać rozwiązanie automatu Moore'a albo Mealy'ego. Wybór krótko uzasadnij.

3.2.6 realizacja w postaci układu mikro-programowalnego

Jeszcze innym podejściem do realizacji automatu w oparciu o pamięć jest układ mikro-programowalny. Jego działanie opiera się na przechowywaniu w pamięci instrukcji złożonych z pól: adresowych, kontrolnych, sterujących i operacyjnych. W oparciu o aktualne dane wejściowe (z *układu operacyjnego*) i aktualny stan (czyli wartość tych pól) wystawiane są dane wyjściowe (do *układu operacyjnego*) oraz podejmowana jest

decyzja o wyborze stanu następnego. Poniżej znajduje się przykładowy schemat działania układu mikro-programowalnego.



W przedstawionym układzie:

1. MUX2 dokonuje selekcji informacji wejściowych w oparciu o zawartość pola kontrolnego.
2. Dekoder Mikro Instrukcji steruje MUX1 (sygnał *Addr Sel*) i licznikiem (sygnał *Load / Inc*) w oparciu o dane przekazane przez MUX2 i zawartość pola sterującego (zawierającą informacje o typie instrukcji), w ten sposób podejmuje on decyzję odnośnie wyboru następnego stanu automatu.
3. Licznik w zależności od sygnału *Load / Inc* zwiększa adres (numer) obecnego stanu (w rezultacie czego automat przechodzi do stanu następnego) lub ładuje adres z multiplexera MUX1.
4. Multiplexer MUX1 w zależności od sygnału *Addr Sel* wystawia do licznika:
   - wartość pola adresowego (co pozwala na skok bezwzględny do innego stanu),
   - wartość aktualnego stanu licznika (co pozwala na pozostanie w obecnym stanie),
   - sumę tych wartości (co pozwala na skok względny do innego stanu).
5. Pole operacyjne używane jest jako wartość wyjścia, jeżeli realizowany jest automat typu Mealy'ego to w celu ustalenia właściwego stanu wyjść może zostać funkcja logiczna na tych danych i danych wejściowych

Zauważ, że model ten nakłada pewne ograniczenia na implementowany automat – w każdym stanie mamy ograniczony zbiór przejść które możemy wykonać, zazwyczaj są to: pozostanie w aktualnym stanie, przejście do stanu następnego, skok bezwarunkowy lub warunkowy do innego stanu (numer stanu do którego przechodzimy lub wartość skoku są na ogół na stałe powiązane z obecnym stanem).

Zadanie 3.2.5

Zaimplementuj automat z zadania 3.1.1 w postaci układu mikro-programowalnego

4 Automaty ze stosem

Najpowszechniejszym zastosowaniem automatów ze stosem są różnego rodzaju analizatory składniowe (parsery), mogą one także posłużyć np. do obliczania wartości wyrażeń artmetycznych.

Wyrażenia arytmetyczne zazwyczaj zapisujemy w postaci infixowej, czyli z operatorem pomiędzy argumentami $(2 + 3 * 5)$. Taka notacja wymaga wiedzy o priorytetach poszczególnych operatorów oraz stosowania nawiasów celem wymuszenia innej niż standardowa kolejności wykonywania działań.

Jeżeli spojrzeć na operacje arytmetyczne jako na dwuargumentowe funkcje moglibyśmy wyrażenia zapisywać jako zagnieżdżony ciąg wywołań funkcji: $+(2, *(3, 5))$. Zauważ, że zapis taki dodatkowo także jednoznacznie określa kolejność wykonywania operacji.

Obydwie te notacje umożliwiają stworzenie automatu ze stosem obliczającego wartość wyrażenia, jednak w tym drugim wypadku jest to znacznie prostsze. Wystarczy żeby automat odkładał na stos kolejne czytane znaki z wejścia aż do napotkania), a następnie zdejmował ze stosu argumenty funkcji aż do napotkania funkcji którą ma wykonać. Po wykonaniu funkcji automat powinien odłożyć wynik na stos i kontynuować czytanie słowa wejściowego.

Zadanie 4.0.1

Napisz program symulujący działanie automatu ze stosem obliczającego wartości wyrażeń w przedstawionej notacji funkcyjnej. Automat powinien obsługiwać następujące dwuargumentowe funkcje: $+()$, $-()$, $*()$ i $/()$. Dla uproszczenia przyjmij że wszystkie liczby w danych wejściowych są jednocyfrowe.

Zadanie 4.0.2

Zmodyfikuj rozwiązanie zadania 4.0.1 tak aby poprawnie obsługiwać liczby wielocyfrowe.

Zwróć uwagę iż przy zastosowaniu tej notacji niektóre (te dla których ma to sens) funkcje mogą posiadać dowolną ilość argumentów. Np. zapis $+(*(4, 2), /(8, 2), 13, +(1, 3))$ jest sensowny i jednoznacznie interpretowany.

Zadanie 4.0.3

Zmodyfikuj rozwiązanie zadania 4.0.1 tak aby funkcje $+()$ i $*()$ mogły przyjmować dowolną ilość argumentów.

Wskazówka: Wieloargumentowe operacje dodawania i mnożenia możesz uzyskać np. w następujący sposób (dla argumentów w postaci listy w zmiennej `lista_argumentow`):

```
import functools, operator
wynik_dodawania = functools.reduce(operator.add, lista_argumentow, 0)
wynik_mnożenia = functools.reduce(operator.mul, lista_argumentow, 1)
```

Zadanie 4.0.4

Zastanów się czy możemy wyeliminować także znaki nawiasów (i) w tym zapisie, gdy nadal chcemy korzystać z automatu ze stosem i:

1. każda funkcja przyjmuje dokładnie dwa argumenty (jak w zadaniu 4.0.1)
2. ilość argumentów jest zależna od funkcji, ale stała dla danej funkcji
3. występują funkcje które mogą przyjmować dowolną ilość argumentów (jak w zadaniu 4.0.3)

Okazuje się, że przy przetwarzaniu tego typu wyrażeń z użyciem automatów większe znaczenie ma informacja o tym, że właśnie zakończyły się argumenty jakiejś funkcji, niż informacja że się zaczynają – pozwala to na konstrukcję prostszych automatów. W związku z tym często stosowana jest Odwrotna Notacja Polska, polegająca na podawaniu argumentów przed operatorem działania (funkcją). Np. `2 3 5`

$*$ + oznacza $3 * 5 + 2$. Wyrażenia takie mogą być łatwo obliczane z użyciem automatu ze stosem, który pobiera dane z wejścia i odkłada je na stos do momentu napotkania operatora, wtedy zdejmuje ze stosu wymaganą przez niego liczbę argumentów i odkłada na stos wynik działania, a następnie kontynuuje pobieranie danych z wejścia.

Zadanie 4.0.5

Napisz program symulujący działanie automatu ze stosem obliczającego wartości wyrażeń w Odwrotnej Notacji Polskiej. Automat powinien obsługiwać dwuargumentowe operatory: $+$, $-$, $*$ i $/$.

Zadanie 4.0.6

Automat ze stosem może zostać także użyty do konwersji standardowej notacji infixowej na Odwrotną Notację Polską. Napisz program symulujący działanie takiego automatu.

5 Maszyna Turinga

Zadanie 5.0.1

Na wykładzie o językach rozstrzygalnych w okolicy 26 slajdu przedstawione było działanie Maszyny Turinga w roli sumatora dwóch liczb binarnych.

Działanie Maszyny Turinga można zasymulować w postaci programu komputerowego. W tym celu nieskończoną taśmę można zasymulować z użyciem listy i funkcji, która dla odwołań poza listą odpowiednio ją rozszerzy wstawiając `'#'`:

```
def get(l, i):
  if i >= len(l):
    l += ['#'] * (i-len(l)+1)
  return l[i]
```

Napisz program symulujący działanie Maszyny Turinga sumującej dwie liczby. Program dla taśmy wejściowej w postaci listy: `{'#', 0, 1, '+', 1, 1, '#'}` powinien zwrócić taśmę z wynikiem obliczeń czyli: `{'#', A, A, '+', B, B, '=', 1, 0, 1, '#'}`.

Zwróć uwagę, iż listy zostały zapisane w pozornie odwrotnej kolejności niż na wykładzie. Jest to efektem tego że przy takiem zapisie pierwszy element listy jest podawany po lewej, a naturalne wydaje się podawanie operacji do wykonania na początku nieskończonej taśmy. Zadanie możesz rozwiązać także z odwróconą kolejnością list.