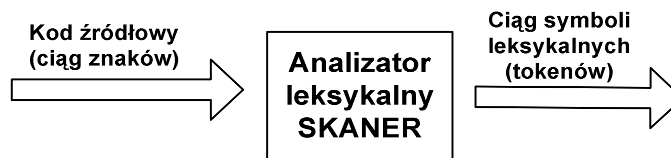
AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

# Analiza leksykalna – 2

## Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

## Zadanie analizy leksykalnej




- wyodrębnianie symboli leksykalnych (tokenów)
- ignorowanie komentarzy
- ignorowanie białych znaków (spacji, tabulacji, znaków nowej linii…)
- korelowanie błędów zgłaszanych przez kompilator z numerami linii
- tworzenie kopii zbioru wejściowego (źródłowego) łącznie z komunikatami o błędach
- czasami realizacja funkcji preprocessingu, rozwijanie makrodefinicji

# Specyfikacja analizatora leksykalnego (1)

| | |
|---|---|
| wyrażenie_regularne_1 | {akcja_1} |
| wyrażenie_regularne_2 | {akcja_2} |
| ………. | …….. |
| wyrażenie_regularne_n | {akcja_n} |

Zasady interpretacji specyfikacji:

- Zawsze rozpoznawany i uzgadniany jest token odpowiadający możliwie najdłuższemu leksemowi.
- W przypadku, gdy ten sam leksem odpowiada więcej niż jednemu tokenowi, uzgadniany jest token odpowiadający najwcześniejszej z „pasujących” pozycji specyfikacji.

# Specyfikacja analizatora leksykalnego (2)

Przykład:

| | |
|---|---|
| a | {akcja_1} |
| abb | {akcja_2} |
| $a^*b^+$ | {akcja_3} |

Analizowany łańcuch: aabba

Możliwe interpretacje: **a|abb|a**, **a|a|bb|a, a|a|b|b|a, aabb|a**

**1 2 1 1 1 3 1 1 1 3 3 1 3 1**

Poprawna interpretacja: **aabb|a**

**3 1**

gdyż:

- Zawsze rozpoznawany i uzgadniany jest token odpowiadający możliwie najdłuższemu leksemowi.

# Specyfikacja analizatora leksykalnego (3)

Przykład:

| | |
|---|---|
| a | {akcja_1} |
| abb | {akcja_2} |
| $a^*b^+$ | {akcja_3} |

Analizowany łańcuch: abba

Niektóre możliwe interpretacje: **abb|a**, **abb|a**
**2 1   3 1**

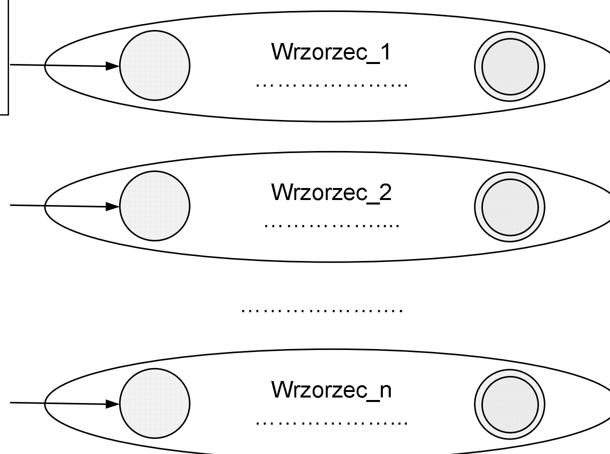
Poprawna interpretacja: **abb|a**
**2 1**

gdyż:

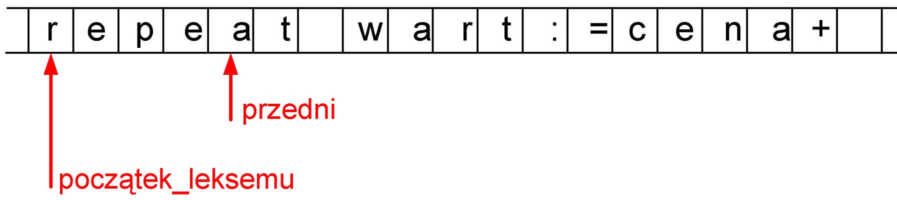
- W przypadku, gdy ten sam leksem odpowiada więcej niż jednemu tokenowi, uzgadniany jest token odpowiadający najwcześniejszej z „pasujących" pozycji specyfikacji.

# Izolowane automaty deterministyczne

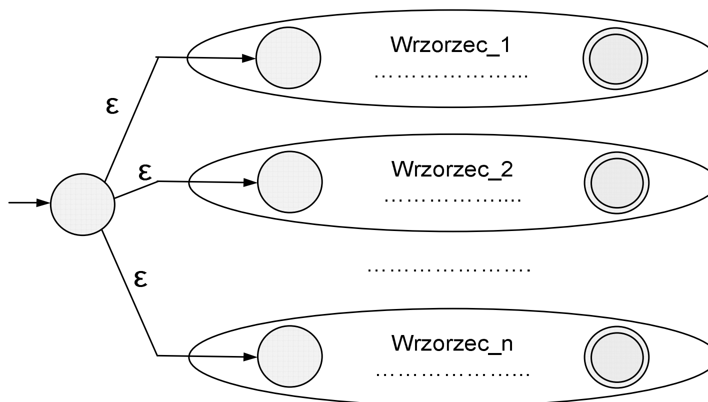
| | |
|---|---|
| wzorzec_1 | {akcja_1} |
| wzorzec_2 | {akcja_2} |
| ………. | ……. |
| wzorzec_n | {akcja_n} |



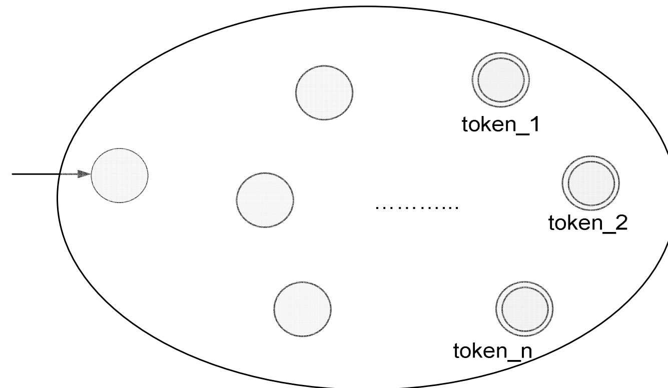
# Najprostsza implementacja – algorytm z powrotami



# Kombinowany automat niedeterministyczny



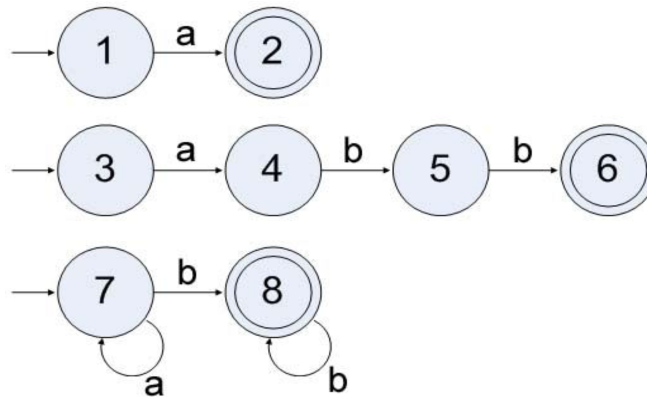
# Automat deterministyczny



# Budowa automatu skończonego (1)

a {akcja 1}
abb {akcja 2}
$a^*b^+$ {akcja 3}

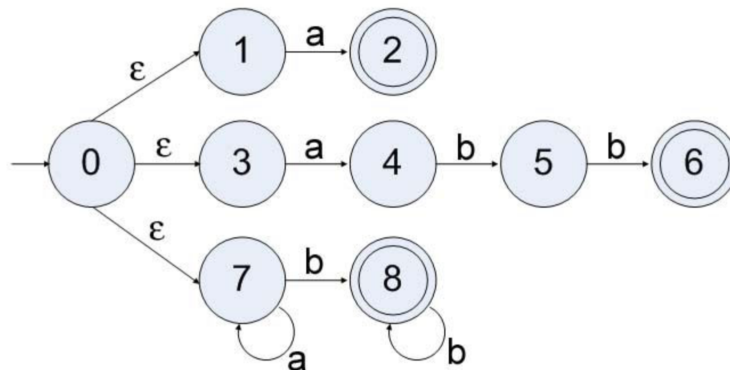
pojedyncze NFA



# Budowa automatu skończonego (2)

a {akcja 1}
abb {akcja 2}
$a^*b^+$ {akcja 3}

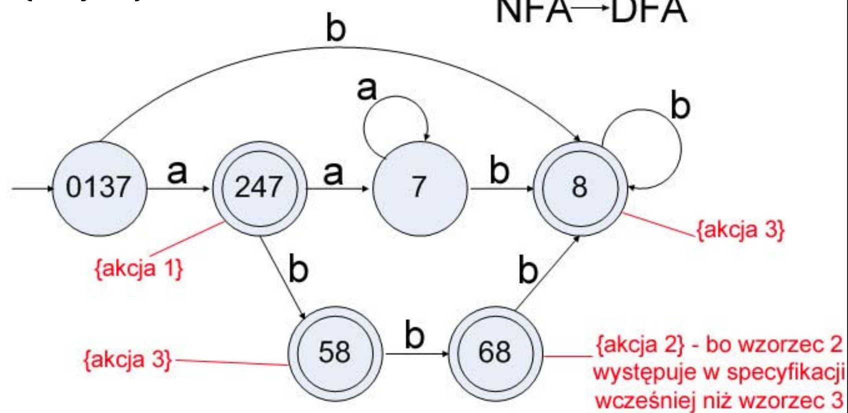
kombinowany NFA



# Budowa automatu skończonego (3)

a {akcja 1}
abb {akcja 2}
$a^*b^+$ {akcja 3}

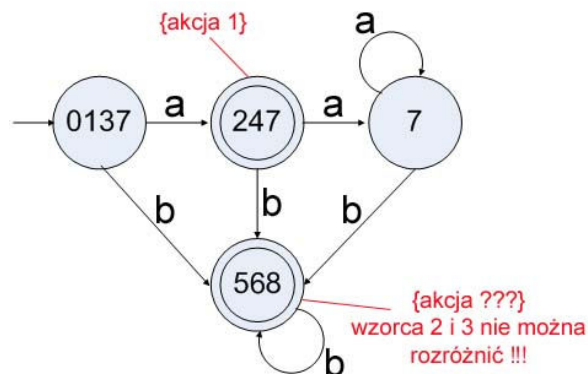
NFA→DFA



# Budowa automatu skończonego (4)

a {akcja 1}
abb {akcja 2}
$a^*b^+$ {akcja 3}

optymalizacja DFA



# Implementacja skanera (1)

Wymagania:

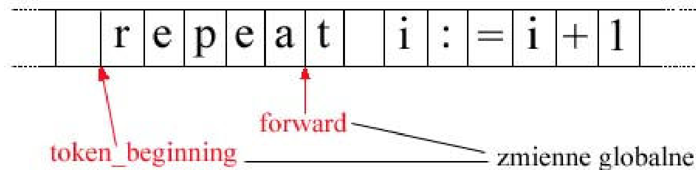
- \- minimalna ilość operacji przypadających na jeden znak czytany z wejścia
- \- rozsądna zajętość pamięci

Sposoby realizacji:

(a) schemat blokowy analizatora ≡ graf automatu (stan jest wtedy miejscem w programie, np. etykietą; dużo if-ów, case-ów, ewentualnie skoków; skomplikowany schemat blokowy działań; utrudniona automatyczna generacja tekstu źródłowego programu analizatora; organizacja czasochłonna)

# Implementacja skanera (2)

- **Organizacja bufora wejściowego**




- **Zmienne globalne**

state - aktualny stan analizatora
start - aktualny stan początkowy wskazujący, który diagram jest aktualnie przeglądany
lexical_value - drugi „składnik" tokenu, jego atrybut

- **Funkcje pomocnicze**

input() - zwraca kolejny znak tekstu analizowanego, przesuwa *forward* jeden znak do przodu
unput() - „oddaje" znak do bufora, przesuwa *forward* jeden znak do tyłu
err_recover() - sygnalizacja błędu, poszukiwanie początku następnego tokenu

# Implementacja skanera (3)

```
token nexttoken()
{
    while(1)
    {
        switch(state)
        {
            case 0: c = input();
                if (c==blank || c==tab || c==newline)
                {
                    state=0;
                    token_beginning++;
                }
                else if (c=='<') state = 1;
                else if (c=='=') state = 5;
                else if (c=='>') state = 6;
                else state = fail();
                break;
```

# Implementacja skanera (4)

```
case 1: c = input();
        if (c=='=') state = 2;
        else if (c=='>') state = 3;
        else state = 4;
        break;
case 2: token_beginning = forward;
        lexical_value=LE;
        return (REL_OP);
        …/*cases 3-8 here*/
```

# Implementacja skanera (5)

```
case 9: c = input();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
case 10: c = input();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
case 11: unput();
        token_beginning = forward;
        install_id();
        return (gettoken());
        …/*cases 12-21 here*/
        }
    }
}
```

# Implementacja skanera (6)

Procedura fail()

```
int state = 0, start = 0;
int lexical_value;
int fail()
{
    forward = token_beginning;
    switch (start)
    {
        case 0: start = 9;
                break;
        case 9: start = 12;
                break;
        case 12: err_recover();
                break;
    }
    return (start);
}
```

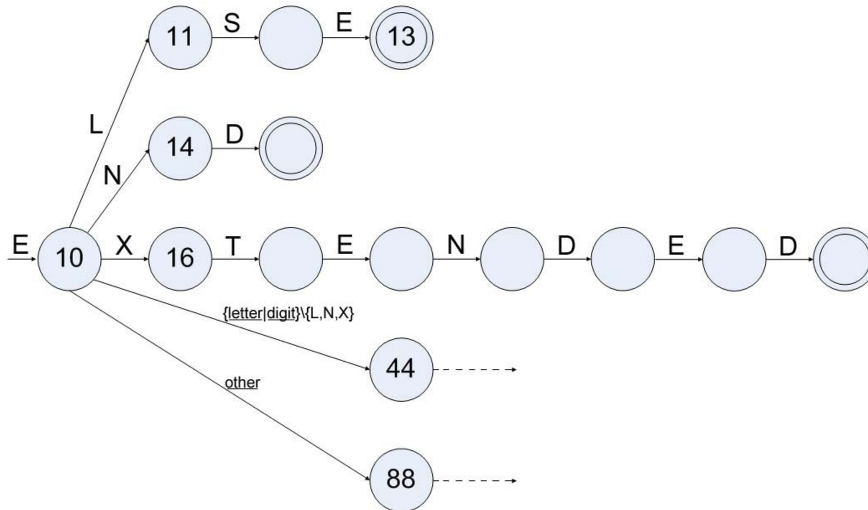
| i | f | ▿ | s | t | < | = | 1 | . | 2 | 5 | E | 4 | ▿ | t | h | e | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

token_beginning forward

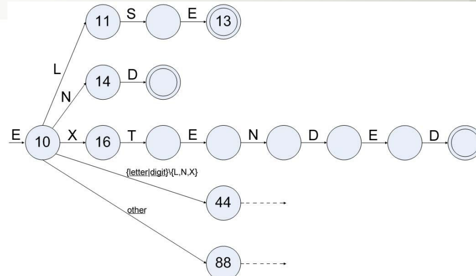
# Implementacja skanera (7)

(b) struktura listowa dla określenia funkcji przejścia (organizacja bardziej zwarta i regularna, łatwiejsza do automatycznej generacji; ale także czasochłonna)

# Implementacja skanera (8)



# Implementacja skanera (9)



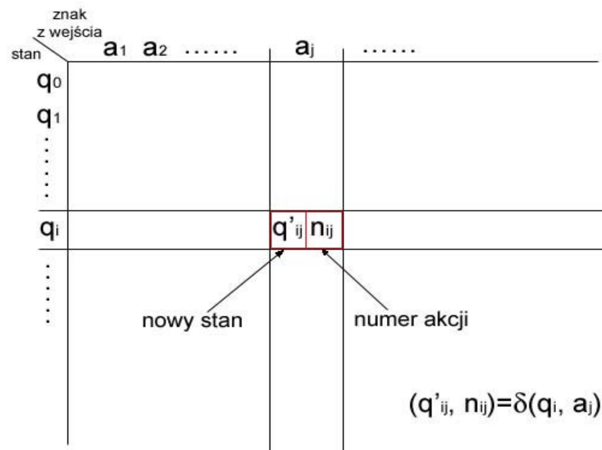

| State | Action | Check |
|---|---|---|
| … | … | … |
| 10 | input() | 1000 |
| … | … | … |
| 13 | return(else) | (start state) |
| … | … | … |

| Check | input char. | yes=next state | no=check |
|---|---|---|---|
| 1000 | L | 11 | 1001 |
| 1001 | N | 14 | 1002 |
| 1002 | X | 16 | 1003 |
| 1003 | letter digit | 44 | 1004 |
| 1004 | other | 88 | |

najczęściej występujące przypadki przejść muszą być umieszczone po kontroli przypadków szczególnych, czyli na końcu listy

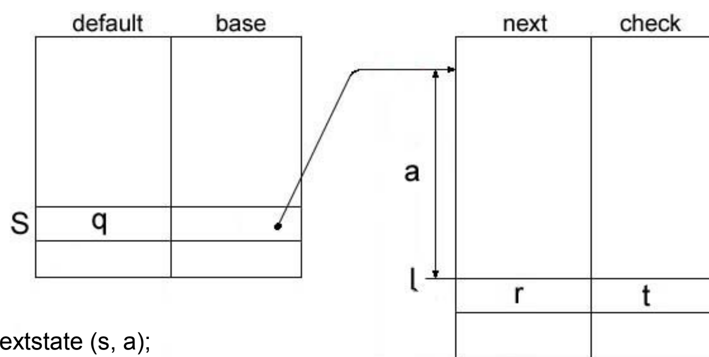
# Implementacja skanera (10)

(c) dwuwymiarowa tablica do realizacji funkcji przejścia
(organizacja efektywna czasowo, ale bardzo pamięciochłonna; duża regularność i łatwość automatycznej generacji)



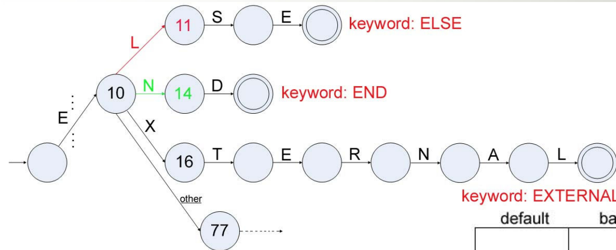
# Implementacja skanera (11)

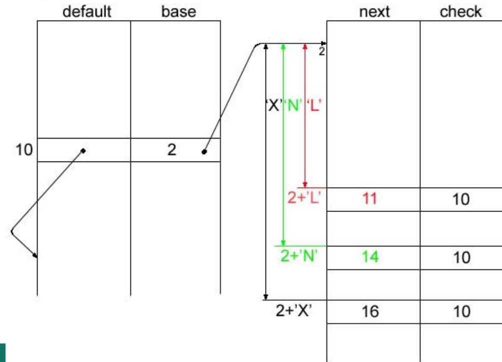
(d) organizacja mieszana




```
procedure nextstate (s, a);
        l:=base[s]+a;
        if check[l]=s then
                return next[l]
        else
                return nextstate (default[s], a);
```

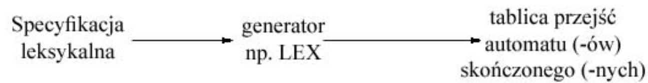
# Implementacja skanera (12)

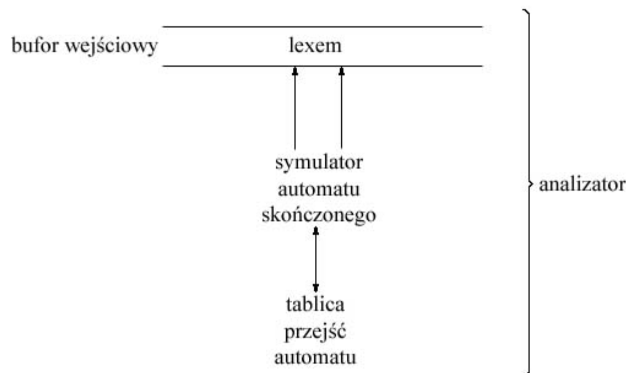


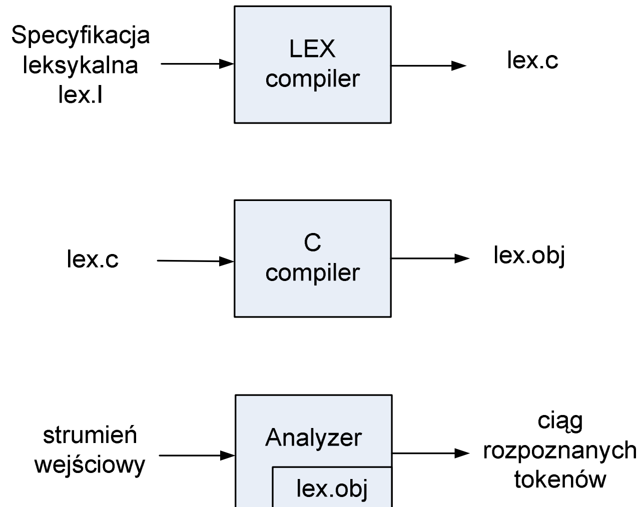

(organizacja stanowiąca kompromis pomiędzy dużą pamięciochłonnością organizacji czysto tablicowej (c), a czasochłonnością metod opartych na listach (b); wadą metody jest skomplikowany algorytm budowania tablic „base", „default", „next" i „check")

# Generatory analizatorów leksykalnych

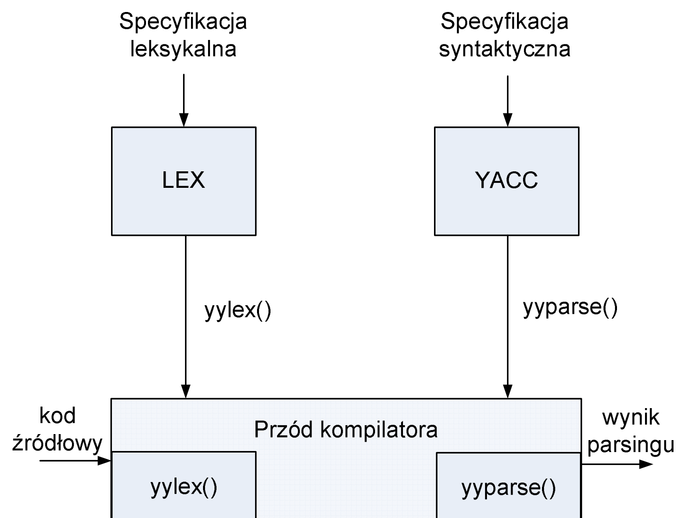


# Generatory analizatorów leksykalnych



# Generatory analizatorów leksykalnych i syntaktycznych



# „Język” tworzenia specyfikacji leksykalnych

| | |
|---|---|
| x | znak „x” |
| „x” | znak „x” nawet gdy x jest operatorem |
| \x | znak „x” nawet gdy x jest operatorem |
| [xy] | znak „x” lub „y” |
| [x-z] | znak „x” lub „y” lub „z” |
| [^x] | dowolny znak z wyjątkiem „x” |
| . | dowolny znak z wyjątkiem „\n” |
| ^x | znak „x” na początku linii |
| <y>x | znak „x” gdy LEX jest w stanie początkowym „y” |
| x$ | znak „x” na końcu linii |
| x? | 0 lub 1 wystąpienie „x” |
| x* | 0, 1, 2,… wystąpień „x” |
| x+ | 1, 2,… wystąpień „x” |
| x\|y | „x” lub „y” |
| (x) | „x” |
| x/y | „x”, ale wtedy, gdy występuje przed „y” |
| {xx} | wyrażenie regularne opisane w sekcji deklaracji |
| x{m,n} | m, m+1,… ,n wystąpień „x” |