Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

# Analiza leksykalna – 1

## Teoria kompilacji

**Dr inż. Janusz Majewski**
**Katedra Informatyki**

# Zadanie analizy leksykalnej

Przykład:

*We:* COST := ( PRICE + TAX ) * 0.98

*Wy:* $\underline{id_1}$ $\underline{:=}$ ( $\underline{id_2}$ + $\underline{id_3}$ ) * $\underline{num_4}$

*Tablica symboli:*

| Adres | Nazwa/wartość | Charakter | Dodatkowa informacja |
|---|---|---|---|
| 1 | COST | zmienna | …. |
| 2 | PRICE | zmienna | …. |
| 3 | TAX | zmienna | …. |
| 4 | 0.98 | stała | …. |

# Zadanie analizy leksykalnej

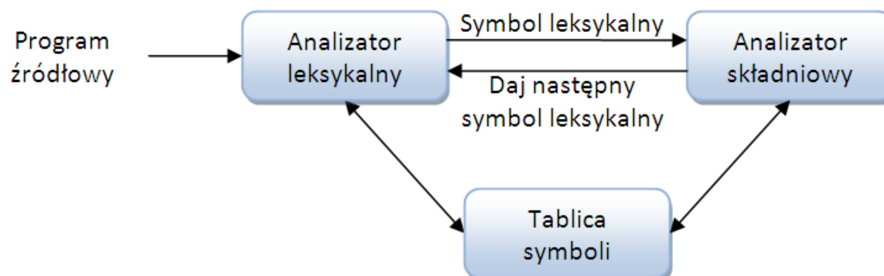
Przykład:

*We:* COST:= ( PRICE + TAX ) * 0.98

*Wy:* $id_1$ := ( $id_2$ + $id_3$ ) * $num_4$

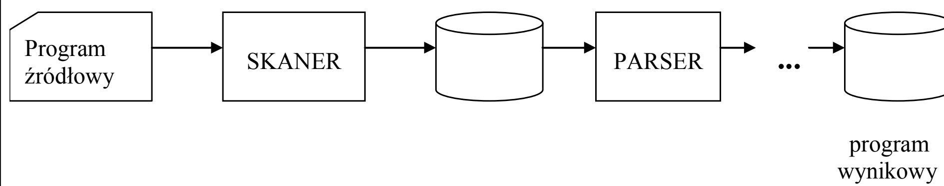
| **Wyjście skanera:** | <= | **Wejście skanera:** |
|---|---|---|
| (id, 1) | | COST |
| (equ, ) | | := |
| (left-par, ) | | ( |
| (id, 2) | | PRICE |
| (plus, ) | | + |
| (id, 3) | | TAX |
| (right-par, ) | | ) |
| (mult, ) | | * |
| (num, 4) | | 0.98 |

# Współpraca z parserem



# Architektura kompilatora (1)

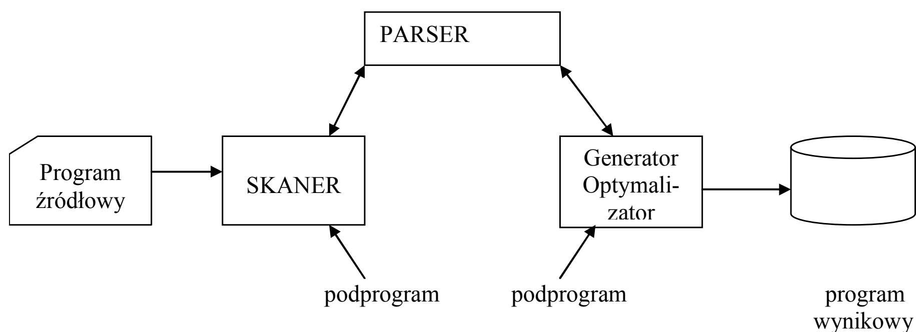
a) Realizacja szeregowa




- translator wieloprzebiegowy
- translator nakładkowy
- mniejsza zajętość PaO
- dłuższy czas kompilacji

# Architektura kompilatora (2)

b) Realizacja równoległa




- translator jednoprzebiegowy
- większa zajętość PaO
- krótszy czas kompilacji

# Zadania analizatora leksykalnego

Zadania analizatora leksykalnego:

- 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

Rozdzielenie etapu analizy na dwie odrębne funkcje: analizę leksykalną i analizę syntaktyczną sprawia, że jedna i druga mogą być wykonywane przy użyciu bardziej efektywnych algorytmów, gdyż algorytmy te istotnie się różnią, wykorzystując inne pryncypia formalne i realizacyjne.

# Podstawowe pojęcia: token, leksem, wzorzec

Przykład:

const pi2=6.2832;

| token | leksem | wzorzec (pattern) |
|---|---|---|
| const (słowo kluczowe) | const | const |
| ws | □ | *biały_znak*$^+$ |
| id | pi2 | *litera* (*litera* \| *cyfra*)* |
| relop | = | < \| > \| <= \| >= \| = \| <> |
| num | 6.2832 | *cyfra*$^+$(. *cyfra*$^+$)? ((E\|e) (+\|–)? *cyfra*$^+$)? |

źródło: `const pi2=6.2832`

leksemy: const □ pi2 = 6.2832

tokeny: const ws id relop num

# Podział gramatyki G opisującej dany język programowania

Gramatyka G języka źródłowego (najczęściej bezkontekstowa):

$G = \langle V, \Sigma, P, S \rangle$

gdzie na ogół:

$\Sigma = \{ A, B, ..., Z, a, b, ..., z, 0, 1, ..., 9, +, -, *, /, ., (, ), ... \}$

Przebudowujemy gramatykę G na rodzinę gramatyk:

$G_S = \langle V_S, \Sigma_S, P_S, S_S \rangle$ - gramatyka syntaktyczna

$\Sigma_S = \{ \Sigma_{S_i} \mid \Sigma_{S_i} \text{jest_tokenem} \}$ - tokeny są terminalami w gramatyce syntaktycznej

$G_i = \langle V_i, \Sigma_i, P_i, S_i \rangle$ - gramatyki leksykalne, jedna dla każdego tokenu

$S_i = \Sigma_{S_i}$

$\Sigma_{S_i} \in V_i$, $\Sigma_i \subset \Sigma$

i-ty token $\Sigma_{S_i}$ jest symbolem początkowym (nieterminalem) i-tej gramatyki leksykalnej.

# Podział gramatyki G opisującej dany język programowania

Podział gramatyki języka źródłowego $G = \langle V, \Sigma, P, S \rangle$

- Gramatyka syntaktyczna: $G_S = \langle V_S, \Sigma_S, P_S, S_S \rangle$
- Rodzina gramatyk leksykalnych $G_i = \langle V_i, \Sigma_i, P_i, S_i \rangle$ po jednej dla każdego tokenu

  $V = V_S \cup \bigcup_{i=1}^{n} V_i$

  $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$

  $P = P_S \cup \bigcup_{i=1}^{n} P_i$

  $S = S_S$

Gramatyka języka źródłowego $G$ jest więc efektem „podstawienia” gramatyk leksykalnych $G_i$ do gramatyki syntaktycznej $G_S$.

# Podział gramatyki G opisującej dany język programowania

| **token** | **leksem** | **wzorzec (pattern)** |
|---|---|---|
| const (słowo kluczowe) | const | const |
| ws | □ | *biały_znak*$^+$ |
| id | pi2 | *litera* (*litera* \| *cyfra*)* |
| relop | = | < \| > \| <= \| >= \| = \| <> |
| num | 6.2832 | *cyfra*$^+$(. *cyfra*$^+$)? ((E\|e) (+\|–)? *cyfra*$^+$)? |

Wzorce są opisami sposobu wyodrębniania tokenów.
Wzorce pełnią rolę produkcji w gramatykach leksykalnych.
Tokeny są symbolami nieterminalnymi w gramatykach leksykalnych.
Tokeny są terminalami w gramatyce syntaktycznej.

# Trudności w budowaniu analizatora leksykalnego

Przykład:

W niektórych językach programowania słowa kluczowe nie są zastrzeżone (FORTRAN, PL/I). W języku PL/I poprawny jest zapis:

**IF** THEN **THEN** THEN = ELSE ; **ELSE** ELSE = THEN;

# Trudności w budowaniu analizatora leksykalnego

Przykład:

W języku FORTRAN spacje (z wyjątkiem spacji wewnątrz łańcuchów) są zawsze ignorowane. Nazwy zmiennych nie wymagają deklaracji. Typ zmiennej ustalany jest na podstawie pierwszej litery nazwy.

Porównujemy:

1) DO 5 **I** = 1 **.** 25 równoznaczne DO5**I** = 1.25
(instrukcja przypisania, DO5**I** – zmienna rzeczywista)

2) DO 5 **I** = 1**,** 25 instrukcja pętli wyliczanej typu "for", odpowiednik:

```
        for I := 1 to 25 do
         begin
            ...
            ...
etykieta    ...
       \    …
        5: end;
```

# Trudności w budowaniu analizatora leksykalnego

Przykład c.d.:

1)

| DO5I | = | 1.25 |
|---|---|---|
| identyfikator | operator przypisania | stała numeryczna |

2)

wyrażenia (tutaj stałe) → 1, 25

| DO | 5 | I | = | 1 | , | 25 |
|---|---|---|---|---|---|---|
| słowo kluczowe DO | etykieta | identyfikator zmiennej sterującej pętli | operator przypisania | wyrażenie | przecinek | wyrażenie |

## Trudności w budowaniu analizatora leksykalnego

### Przykład c.d.:

**DO 5 I = 1** $\left\{ \begin{matrix} . \\ \backslash n \\ , \end{matrix} \right\}$

Po przeczytaniu znaków DO nie można dokonać uzgodnienia tokenu "Słowo kluczowe DO" dopóki nie zbada się prawego kontekstu i nie znajdzie się przecinka (wtedy rzeczywiście uzgadnia się "DO") lub kropki bądź znaku nowej linii (wtedy mamy instrukcje podstawienia).

## Definicje regularne

Do opisu wzorców dla skanera stosujemy definicje regularne:

$d_1 \to r_1$

$d_2 \to r_2$

............

$d_n \to r_n$

gdzie:

$d_i$ - unikalna nazwa

$r_i$ - wyrażenie regularne nad symbolami alfabetu $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$

## Przykład definicji regularnych (1)

Stałe bez znaku w Pascal'u:

*cyfra* → *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

*cyfry* → *cyfra cyfra* *

*część-ułamkowa* → **.** *cyfry* | $\varepsilon$

*wykładnik* → *( E | e ) ( + | – | $\varepsilon$ ) cyfry* | $\varepsilon$

*num* → *cyfry część-ułamkowa wykładnik*

## Rozszerzenie zbioru operatorów

Dla ułatwienia wprowadza się nowe operatory w wyrażeniach regularnych, np.:

$w^+$ - oznacza jedno lub więcej wystąpień wzorca *w*

$$w^+ = w\, w^*$$

*w?* - oznacza zero lub jedno wystąpienie wzorca *w*

$$w? = w \mid \varepsilon$$

# Przykład definicji regularnych (2)

Stałe bez znaku w Pascal'u zapisane po rozszerzeniu zbioru operatorów w wyrażeniach regularnych:

*cyfra* → *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

*cyfry* → *cyfra* ⁺

*część-ułamkowa* → *( . cyfry ) ?*

*wykładnik* → *( ( E | e ) ( + | – ) ? cyfry ) ?*

*num* → *cyfry część-ułamkowa wykładnik*

# Rozpoznawanie tokenów

**Przykładowa gramatyka syntaktyczna:**

*stmt* → *if expr then stmt* | *if expr then stmt else stmt* | *ε*
*expr* → *term relop term* | *term*
*term* → *id* | *num*

**Definicje regularne:**

*delim* → □ | *\t* | *\n*
*ws* → *delim* ⁺
*letter* → *a* | *b* | ... | *z* | *A* | *B* | ... | *Z*
*digit* → *0* | *1* | ... | *9*
*if* → *if*
*then* → *then*
*else* → *else*
*relop* → < | <= | <> | > | >= | =
*id* → *letter* ( *letter* | *digit* )*
*num* → *digit* ⁺ ( . *digit* ⁺ ) ? ( *E* ( + | – ) ? *digit* ⁺ ) ?

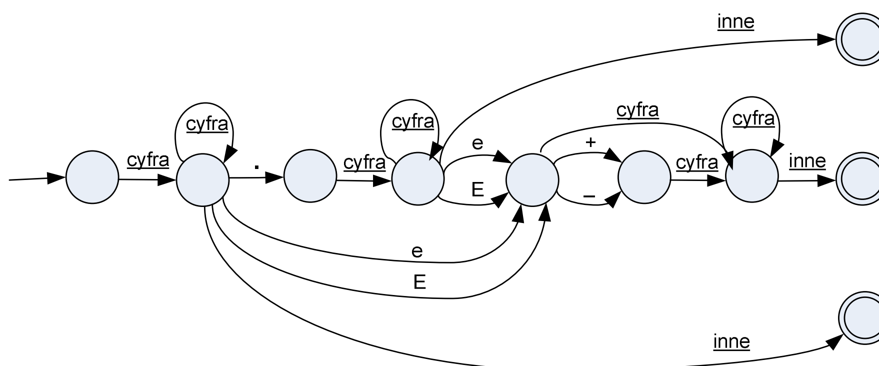
# Skaner ma rozpoznawać…

| | token | atrybut tokenu |
|---|---|---|
| ws | - | - |
| if | if | - |
| then | then | - |
| else | else | - |
| id | id | wskaźnik do tablicy symboli |
| num | num | wskaźnik do tablicy symboli |
| < | relop | LT |
| <= | relop | LE |
| <> | relop | NE |
| = | relop | EQ |
| > | relop | GT |
| >= | relop | GE |

# Diagramy przejść (1)

*Liczba bez znaku w Pascalu – token* ***num***

cyfra$^+$ (. cyfra$^+$ )? ( (e|E) (+|−)? cyfra$^+$ )?

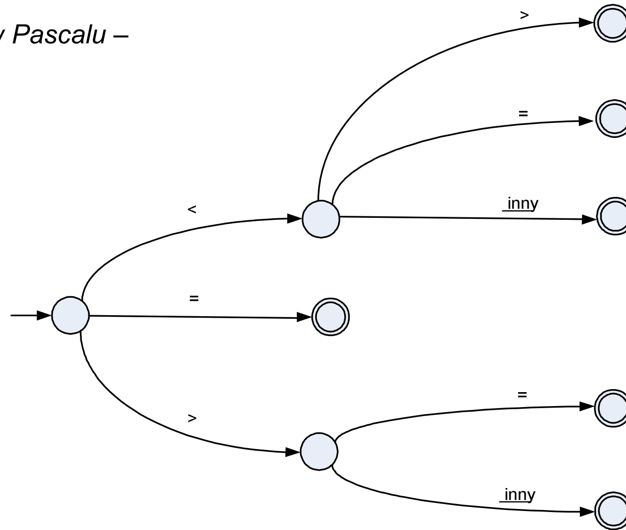


AGH

# Diagramy przejść (2)

*Operatory relacyjne w Pascalu –*
*token **relop***

< | <= | <> | = | >= | >

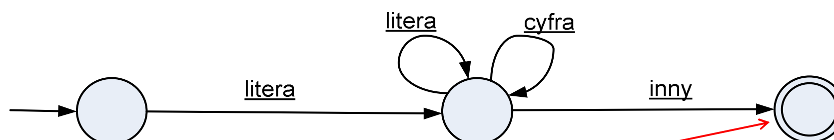



AGH

# Diagramy przejść (3)

*Identyfikator – token **id***

litera ( litera | cyfra )*

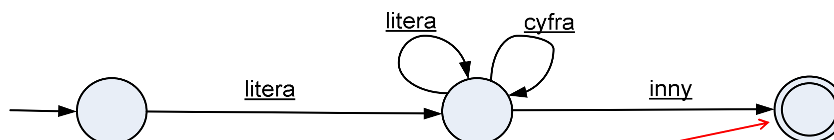



„oddaj" ostatni przeczytany symbol na wejście;
sprawdź, czy leksem to słowo kluczowe;
jeśli tak – zwróć odpowiednie słowo kluczowe;
jeśli nie – sprawdź, czy identyfikator jest
już w tablicy symboli;
jeśli jest – zwróć adres jego pozycji;
jeśli nie ma – utwórz nową pozycję, wpisz
identyfikator do tablicy symboli i zwróć wskaźnik.

# Diagramy przejść (4)

*Identyfikatory*

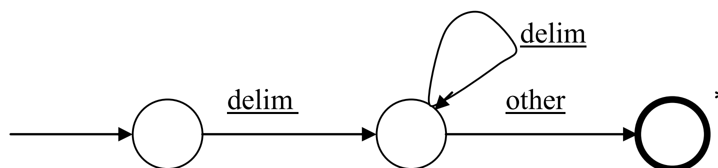
litera ( litera | cyfra )*




„oddaj" ostatni przeczytany symbol na wejście; sprawdź, czy leksem to słowo kluczowe...

Nie zawsze stosuje się metodę umieszczania słów kluczowych w tablicy i sprawdzania każdego identyfikatora, czy nie jest on słowem kluczowym. Można budować dla słów kluczowych odrębne wzorce i konstruować automaty rozpoznające. Wówczas jednak liczba stanów analizatora gwałtownie rośnie.

# Diagramy przejść – ignorowanie białych znaków




cofnij ostatni przeczytany znak na wejście;
żadna inna akcja analizatora nie jest podejmowana