



AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Translacja – wprowadzenie

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Literatura

- 1) Aho A. V., Sethi R., Ullman J. D.: *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1986 (jest tłumaczenie polskie: Kompilatory. Reguły, metody i narzędzia, WNT, 2002).
- 2) Aho A. V., Ullman J. D.: *The Theory of Parsing, Translation and Compiling*, vol. 1, Prentice-Hall, 1972 (tłumaczenie rosyjskie: Ахо А., Ульман Дж.: Теория синтаксического анализа, перевода и компиляции, Издательство „Мир”, 1978)
- 3) Gries D.: *Compiler Construction for Digital Computers* (jest tłumaczenie polskie).
- 4) Waite W. M., Goos G.: *Konstrukcja kompilatorów*, WNT, 1989.

Translator, definicje

- (1) Translator to program, który umożliwia wykonanie programów napisanych w języku różnym od języka komputera.
- (2) Translator to specjalny program komputerowy, dokonujący tłumaczenia (translacji) programu napisanego w języku programowania, z postaci źródłowej do postaci wynikowej, zrozumiałej dla maszyny.

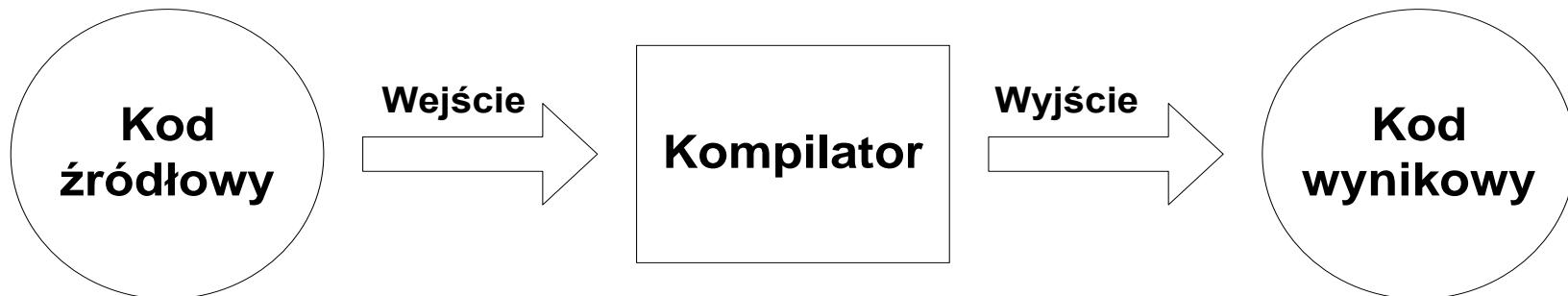
Rozróżnia się dwa rodzaje translatorów oraz odpowiednio dwie podstawowe techniki translacji:

- kompilatory (kompilacja)
- interpretery (interpretacja)

Kompilator, komplikacja

- Kompilator: program, który tłumaczy tekst (kod) źródłowy programu na równoważny tekst (kod) wynikowy. Program źródłowy jest napisany w języku źródłowym, a program wynikowy należy do języka wynikowego. Wykonanie programu kompilatora następuje w czasie tłumaczenia.
- Cecha zasadnicza: program po przetłumaczeniu nie da się zmienić, jest statyczny.
- Kompilator jako dane wejściowe otrzymuje „cały” program źródłowy i przekształca go na postać wynikową.

Kompilator, komplikacja



Zalety komplikacji:

- program skompilowany wykonuje się szybciej niż program interpretowany
- do wykonania programu wynikowego nie jest potrzebny kompilator

Interpreter, interpretacja

- Interpreter można nazwać dynamicznym translatorem. Tłumaczy on oraz na bieżąco wykonuje program źródłowy. Działanie interpretera polega na wyodrębnieniu niewielkich jednostek programu źródłowego, tłumaczeniu ich na pewną postać wynikową i natychmiastowym ich wykonywaniu. Proces jest cykliczny. W czasie interpretacji przechowywany jest program źródłowy. Wynik tłumaczenia nie jest dostępny.
- Przykład:

for i:=1 to 5 do s := s + i ;

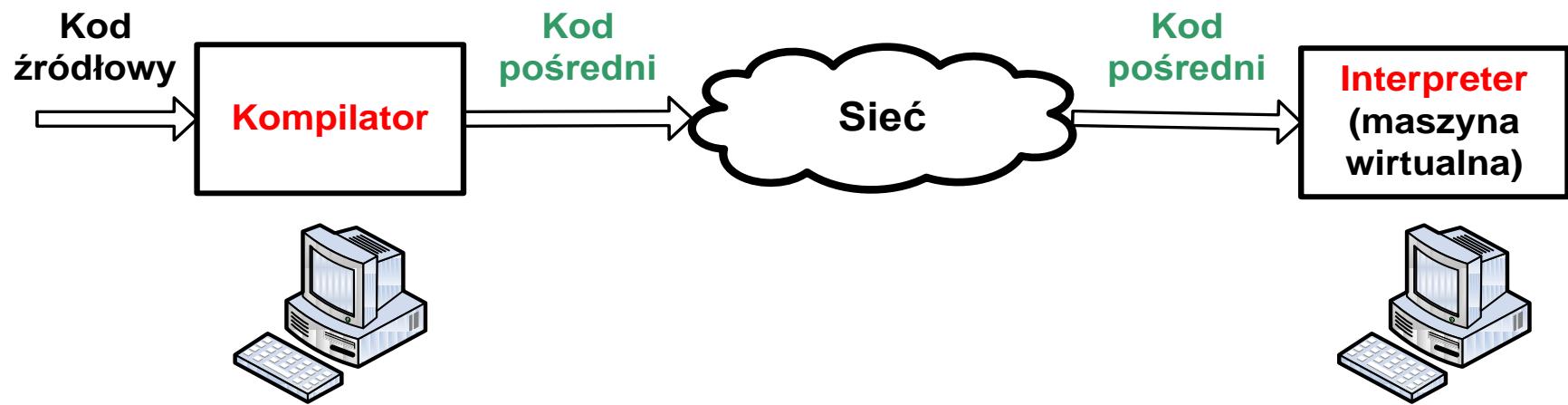
s:=s+i – w przypadku interpretera ten fragment jest 5-krotnie tłumaczony

Interpreter, interpretacja

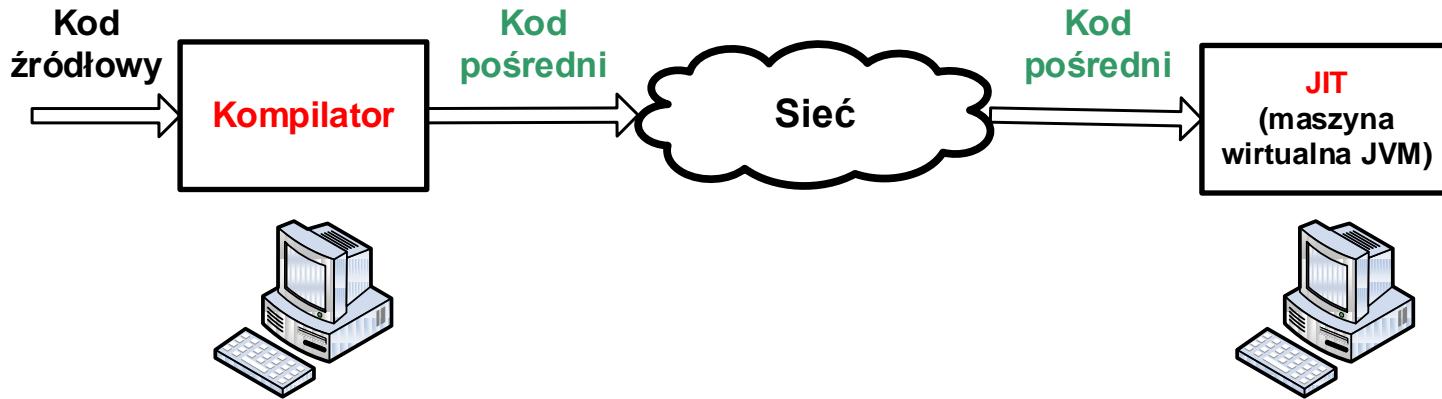
Zalety interpreterów:

- łatwość zmian programu
- mniejsza zajętość pamięci zewnętrznej (tylko tekst źródłowy)
- możliwość pracy konwersacyjnej (zatrzymanie wykonania, zmiana wartości zmiennych, modyfikacja kodu, kontynuacja wykonania)
- przenośność, niezależność od platformy systemowo-sprzętowej, wykorzystanie w zastosowaniach sieciowych (tylko tekst źródłowy)

Kompilacja + interpretacja = Java

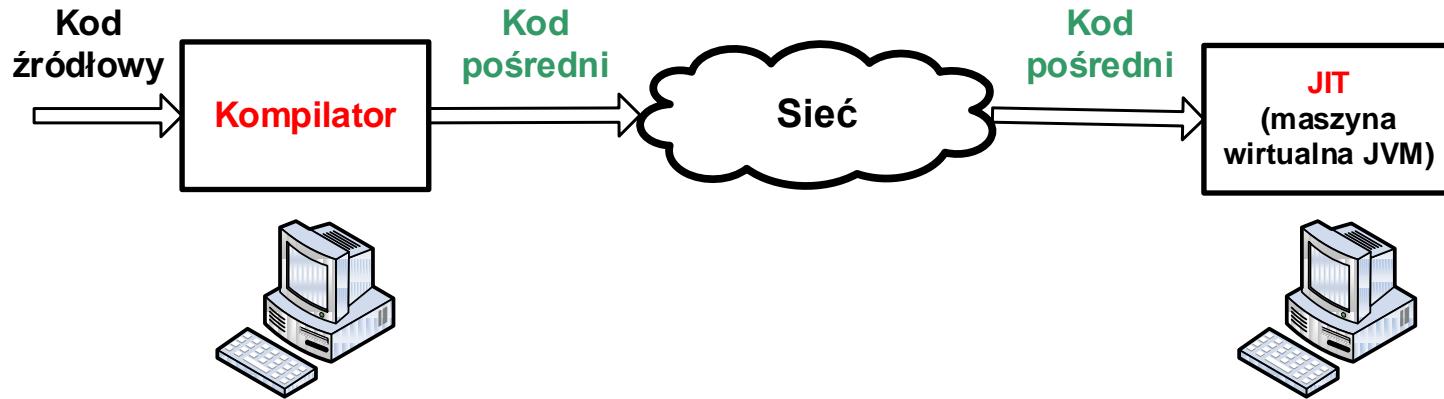


Kompilacja + JIT = Java



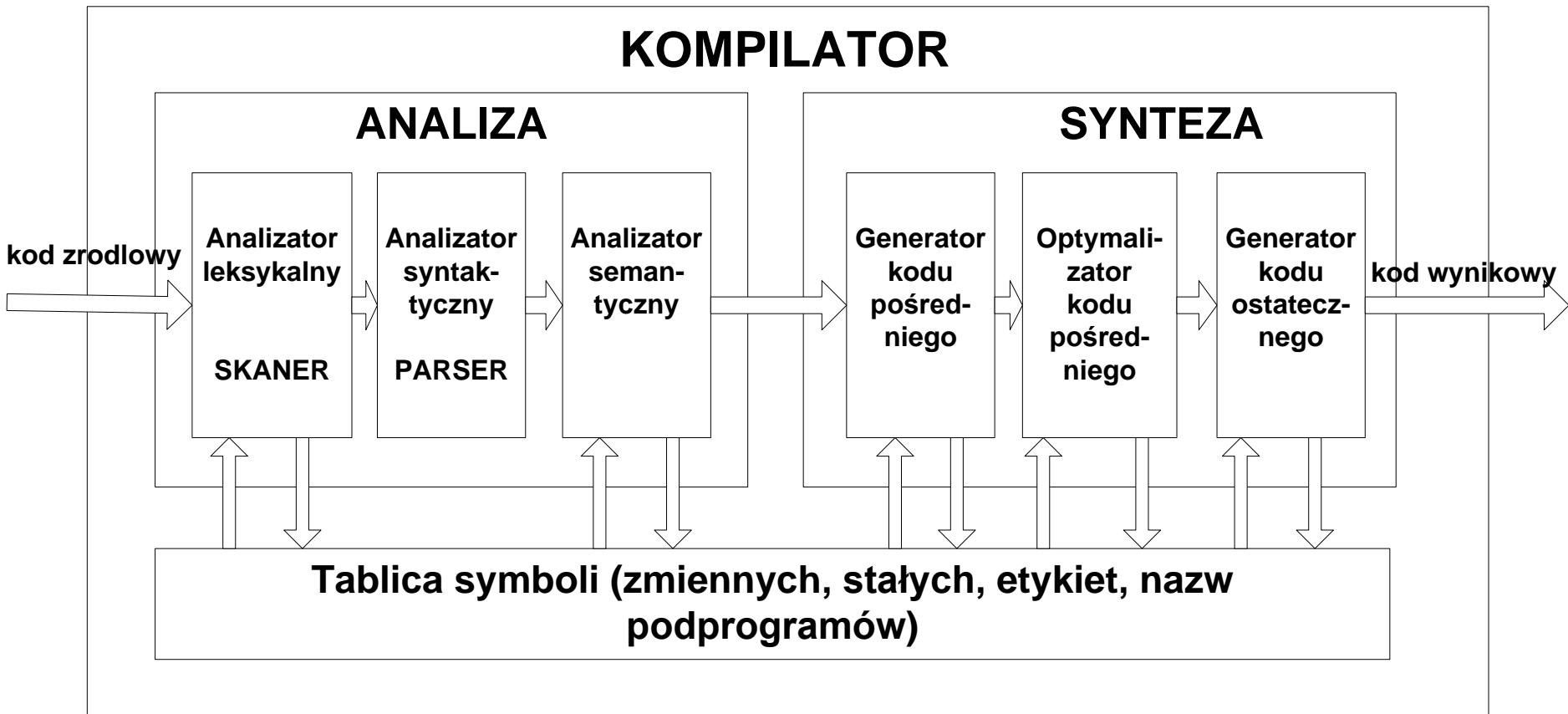
Maszyna wirtualna JVM obecnie to JIT (Just-In-Time, w domyśle kompilator), czyli środowisko uruchomieniowe, które z początku interpretuje kod pośredni, a gdy dany fragment kodu jest wykonywany wiele razy (staje się tzw. „gorącym punktem”, stąd nazwa najpopularniejszej JVM czyli Oracle HotSpot) to ten fragment zostaje skompilowany do natywnego kodu maszynowego i ten kod jest wykorzystywany zamiast interpretowania kodu pośredniego (tzw. bajtowego) Javy.

Kompilacja + JIT = Java

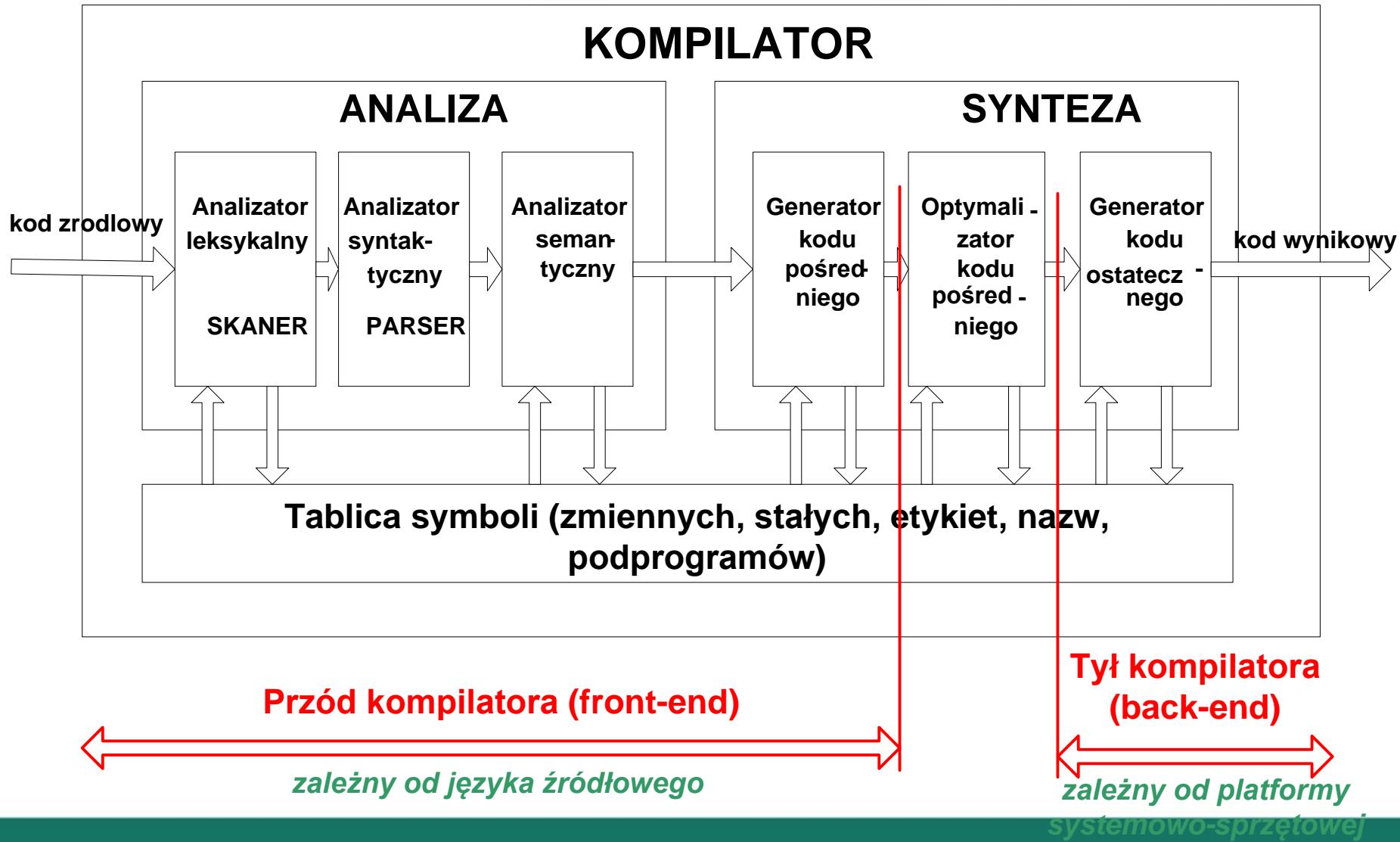


JIT jest zaawansowanym systemem potrafiącym wykonywać programy znacznie szybciej niż interpretery. Fakt zastosowania środowiska uruchomieniowego w postaci JVM pozwala na zastosowanie optymalizacji, które nie są i nigdy nie będą dostępne na etapie kompilacji statycznej, co może doprowadzić do sytuacji, w której aplikacje wykonywane pod kontrolą JIT będą szybsze niż ich natywne odpowiedniki kompilowane statycznie.

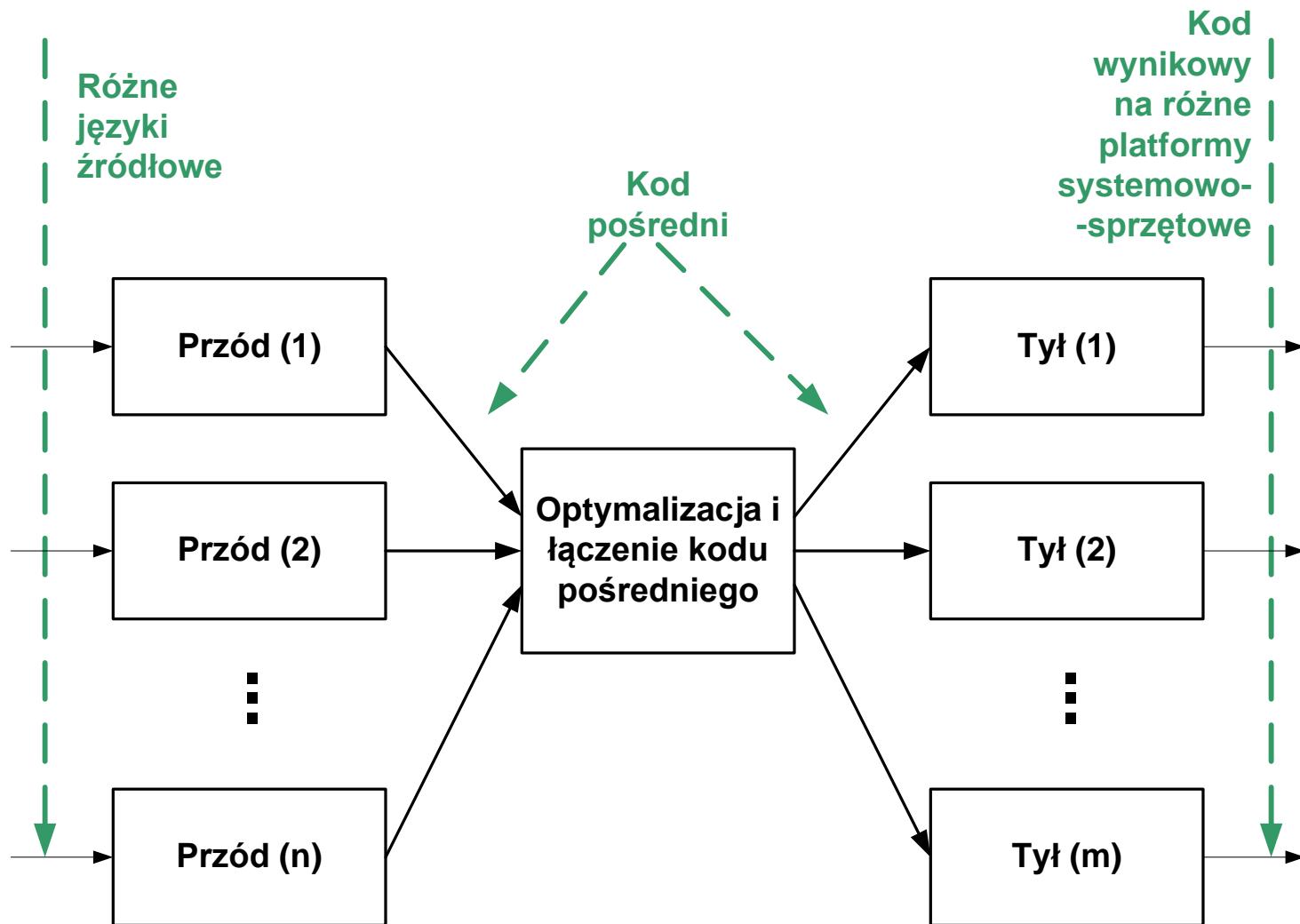
Struktura kompilatora



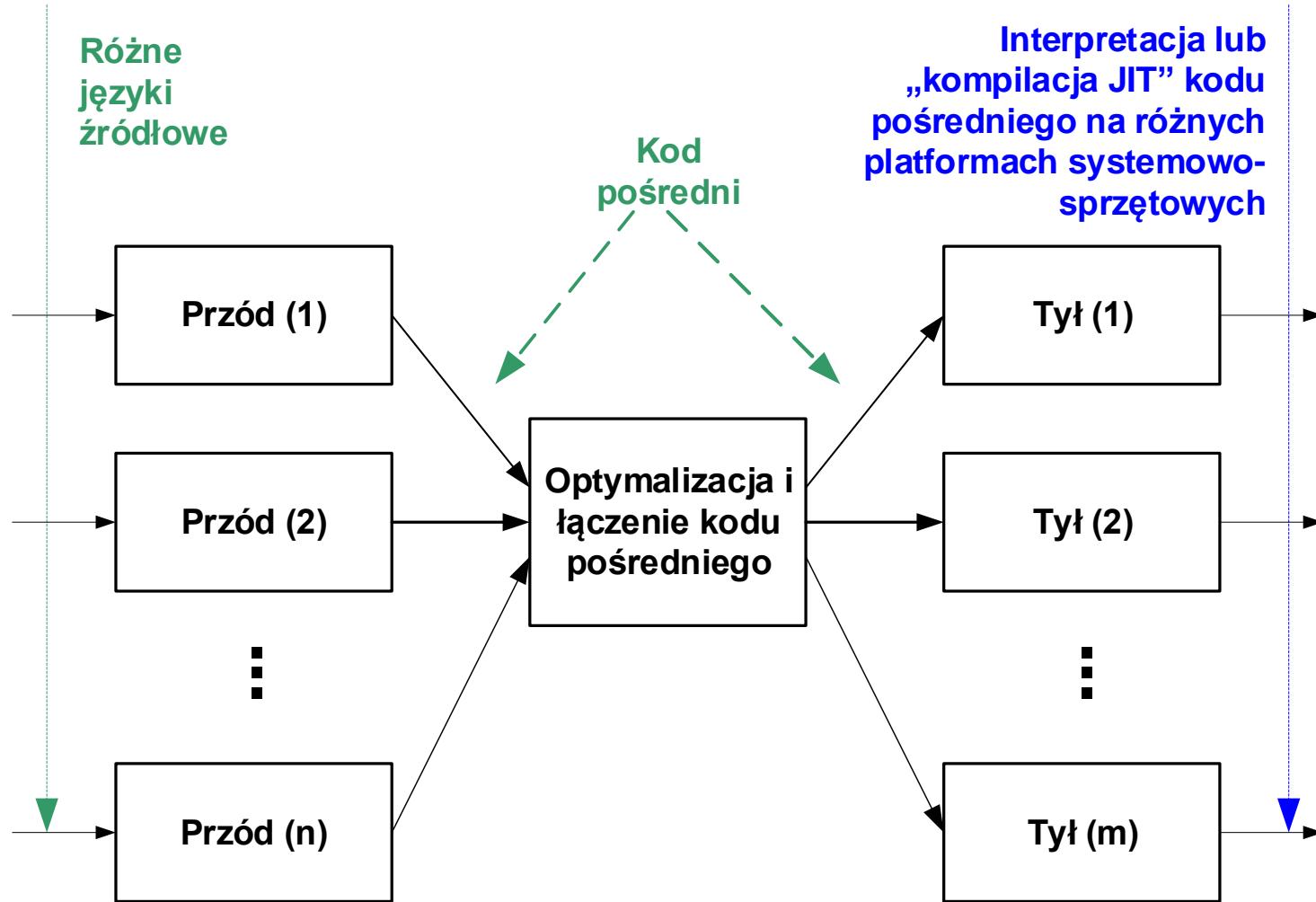
Przód i tył kompilatora



Znaczenie kodu pośredniego, środowiska zintegrowane



Znaczenie kodu pośredniego, środowiska zintegrowane





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: $\underline{id}_1 := (\underline{id}_2 + \underline{id}_3) * \underline{num}_4$

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: id₁ := (id₂ + id₃) * num₄

Wyjście skanera:

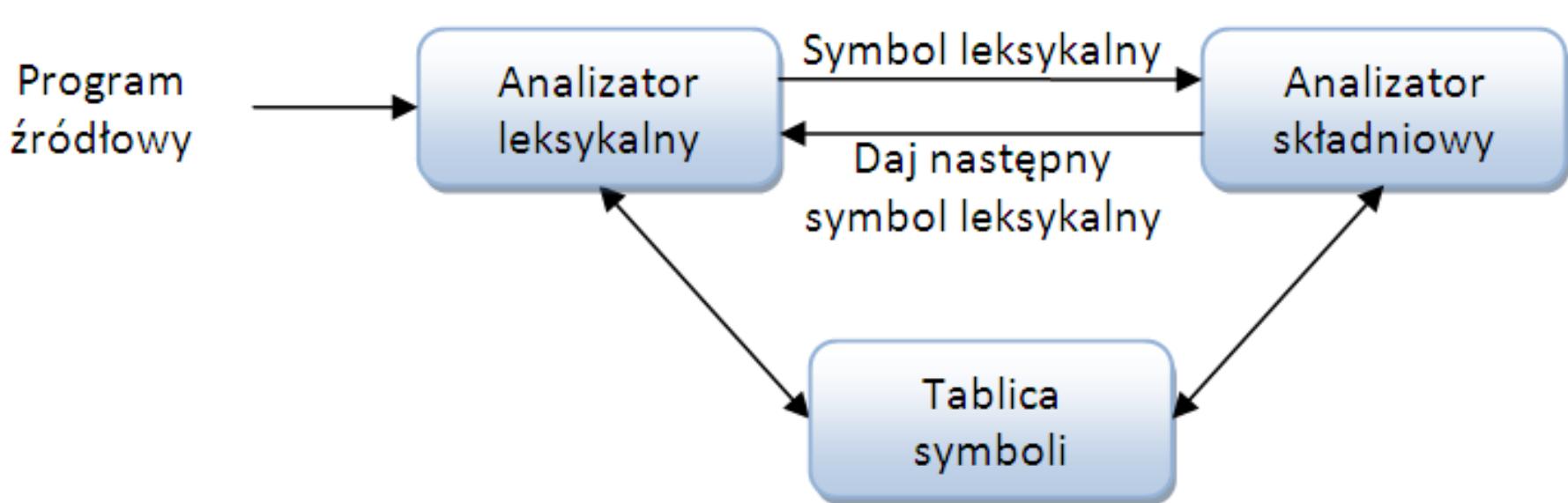
(id, 1)
(equ,)
(left-par,)
(id, 2)
(plus,)
(id, 3)
(right-par,)
(mult,)
(num, 4)

<=

Wejście skanera:

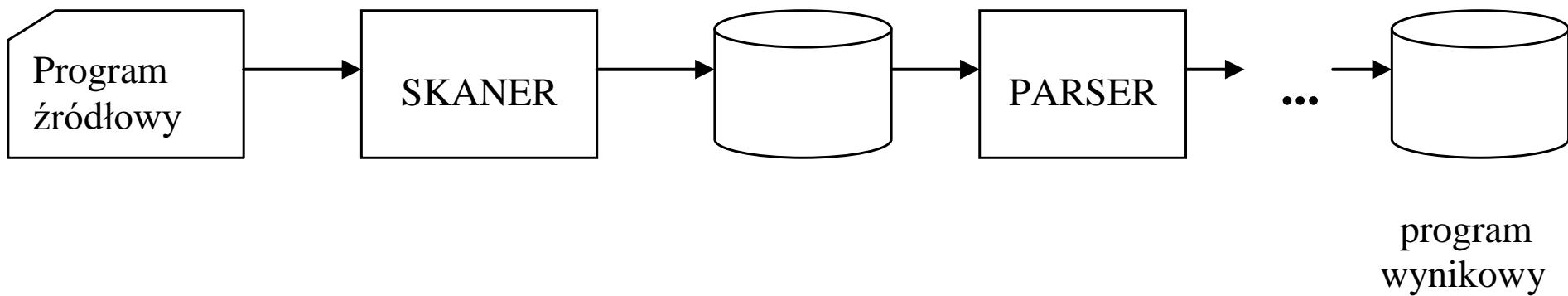
COST
:=
(
PRICE
+
TAX
)
*
0.98

Współpraca z parserem



Architektura kompilatora (1)

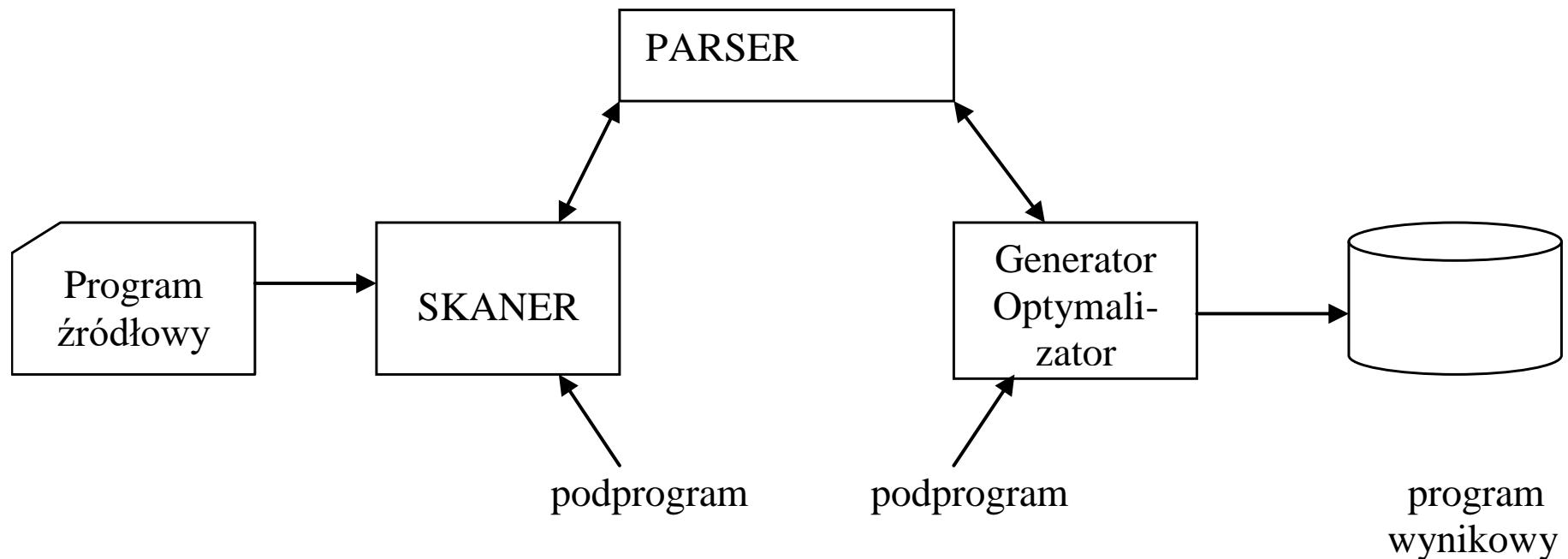
a) Realizacja szeregową



- 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)
<u>const</u> (słowo kluczowe)	const	const
<u>ws</u>	□	<u>biały znak</u> ⁺
<u>id</u>	pi2	<u>litera</u> (<u>litera</u> <u>cyfra</u>) [*]
<u>relop</u>	=	< > <= >= = <>
<u>num</u>	6.2832	<u>cyfra</u> ⁺ (. <u>cyfra</u> ⁺)? ((E e) (+ -)? <u>cyfra</u> ⁺)?

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 ogólnie:

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

Przebudujemy gramatykę G na rodzinę gramatyk:

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

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

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

$$S_i = \Sigma_{S_i}$$

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

i-ty token Σ_{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)
<u>const</u> (słowo kluczowe)	const	const
<u>ws</u>	□	<u>biały znak</u> ⁺
<u>id</u>	pi2	<u>litera</u> (<u>litera</u> <u>cyfra</u>) [*]
<u>relop</u>	=	< > <= >= = <>
<u>num</u>	6.2832	<u>cyfra</u> ⁺ (. <u>cyfra</u> ⁺)? ((E e) (+ -)? <u>cyfra</u> ⁺)?

Wzorce są opisami sposobu wyodrębniania tokenów.

Wzorce pełnią rolę produkcji w gramatykach leksykalnych.

Tokeny są symbolami nieterminalnymi w gramatykach leksykalnych.

Tokeny są terminalami w syntaktyce 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 DO5I = 1.25
(instrukcja przypisania, DO5I – 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)

DO **5** **I** **=** **1** **,** **25**

słowo kluczowe
DO

etykieta

operator
przypisania

przecinek

identyfikator zmiennej sterującej pętli

Trudności w budowaniu analizatora leksykalnego

Przykład c.d.:

DO 5 I = 1 { . \n }

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 instrukcję podstawienia).

Definicje regularne

Do opisu wzorców dla skanera stosujemy definicje regularne:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.....

$$d_n \rightarrow r_n$$

gdzie:

d_i - unikalna nazwa

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

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 | ε

wykładnik → (E | e) (+ | - | ε) cyfry | ε

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

$$w^? = w \mid \epsilon$$

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:

$stmt \rightarrow if \ expr \ then \ stmt \mid if \ expr \ then \ stmt \ else \ stmt \mid \epsilon$

$expr \rightarrow term \ relop \ term \mid term$

$term \rightarrow id \mid num$

Definicje regularne:

$\underline{delim} \rightarrow \square \mid \backslash t \mid \backslash n$

$\underline{ws} \rightarrow \underline{delim}^+$

$\underline{letter} \rightarrow a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z$

$\underline{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$

$\underline{if} \rightarrow if$

$\underline{then} \rightarrow then$

$\underline{else} \rightarrow else$

$\underline{relop} \rightarrow < \mid <= \mid <> \mid > \mid >= \mid =$

$\underline{id} \rightarrow \underline{letter} (\underline{letter} \mid \underline{digit})^*$

$\underline{num} \rightarrow \underline{digit}^+ (\cdot \underline{digit}^+)? (E (+ / -) ? \underline{digit}^+)?$



AGH

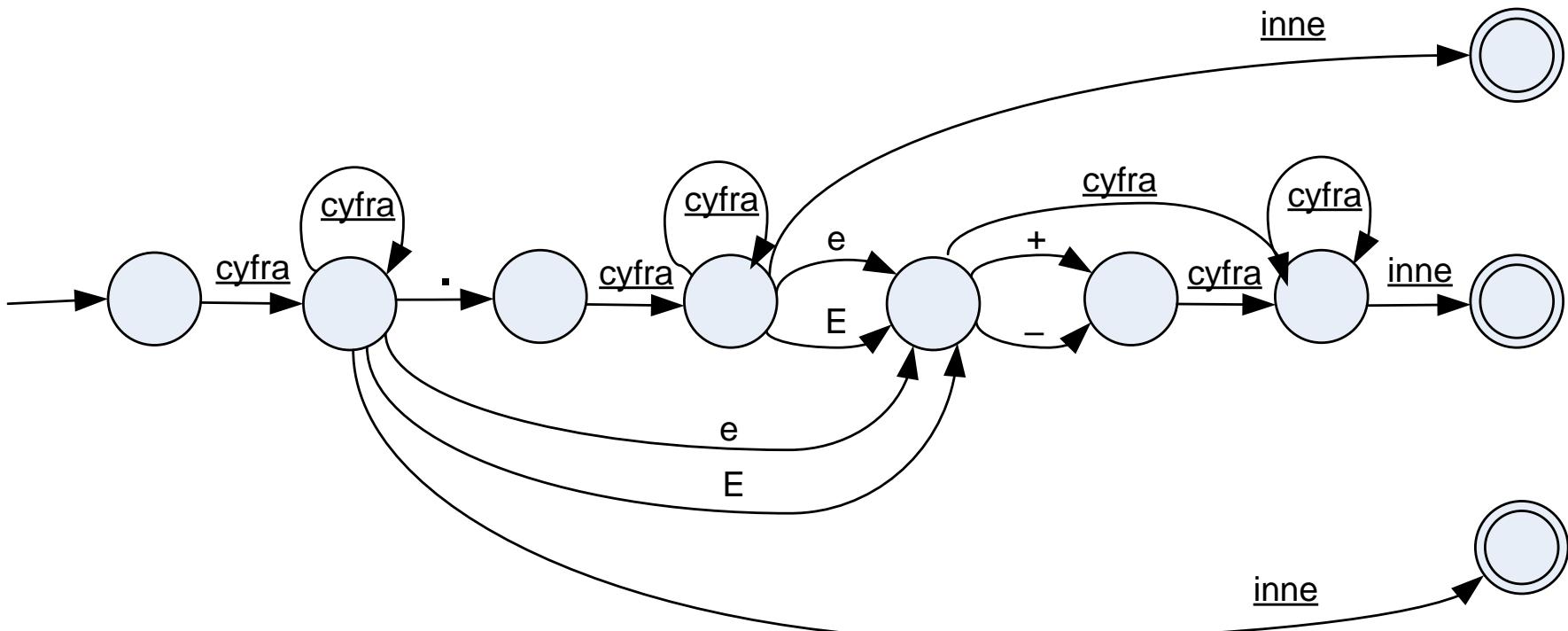
Skaner ma rozpoznawać...

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

Diagramy przejść (1)

Liczba bez znaku w Pascalu – token num

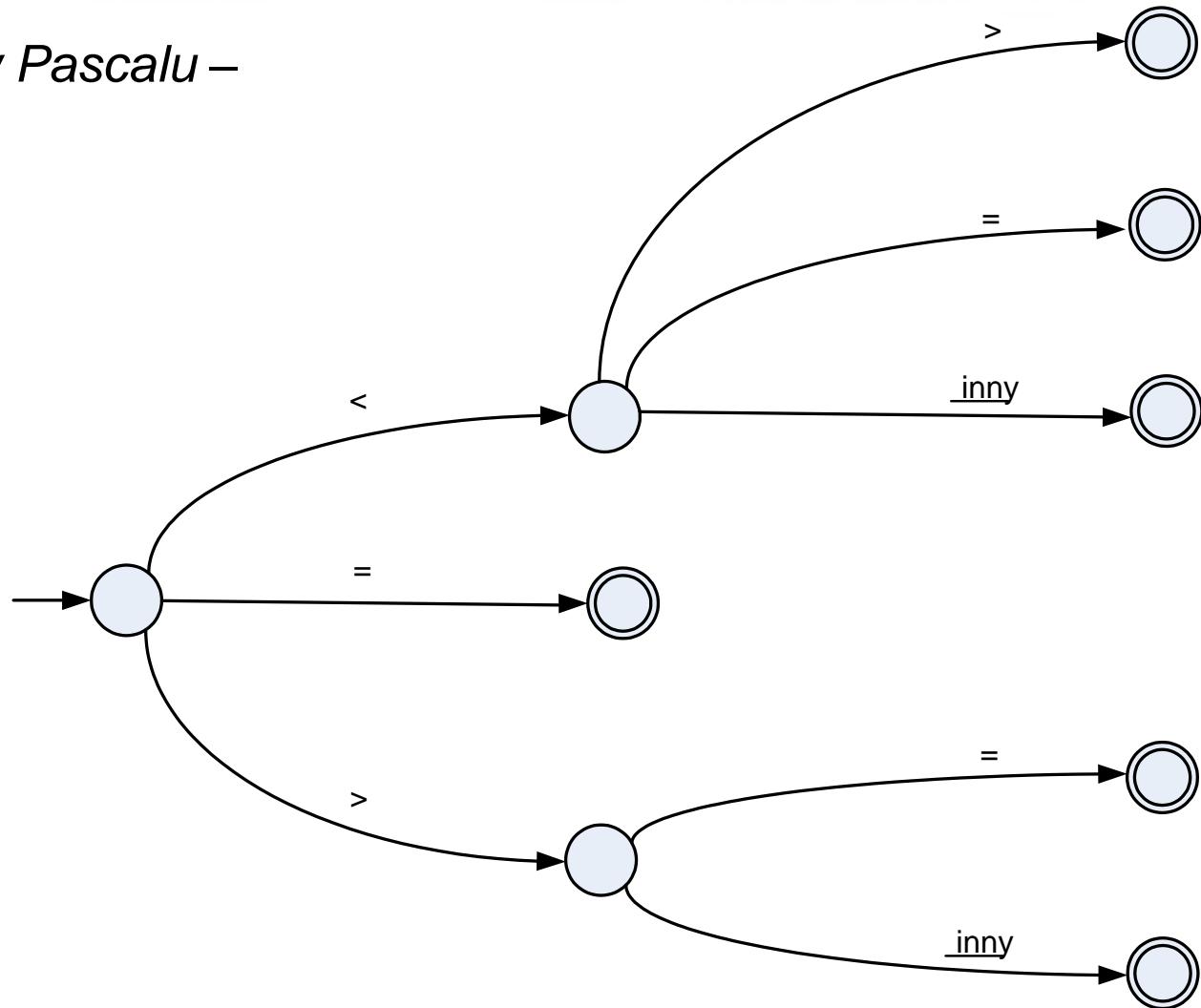
cyfra⁺ (. cyfra⁺)? ((e|E) (+|-)? cyfra⁺)?



Diagramy przejść (2)

Operatory relacyjne w Pascalu –
token relop

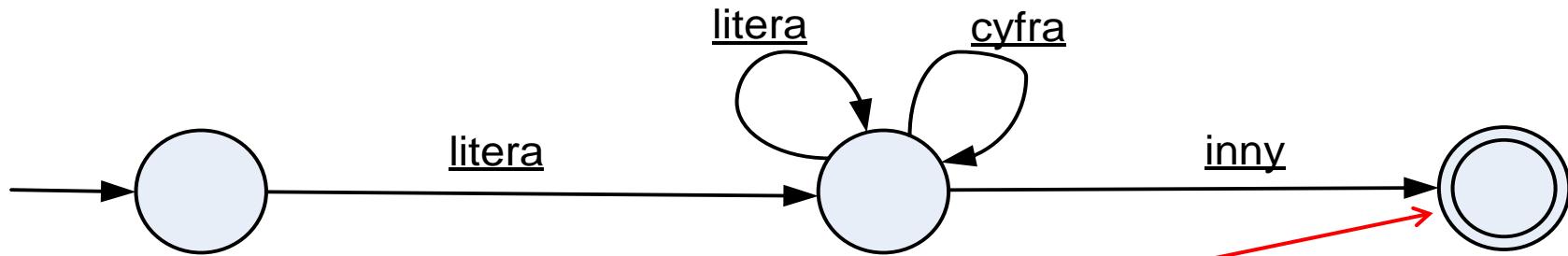
< | <= | <> | = | >= | >



Diagramy przejść (3)

Identyfikator – token *id*

litera (litera | cyfra)*

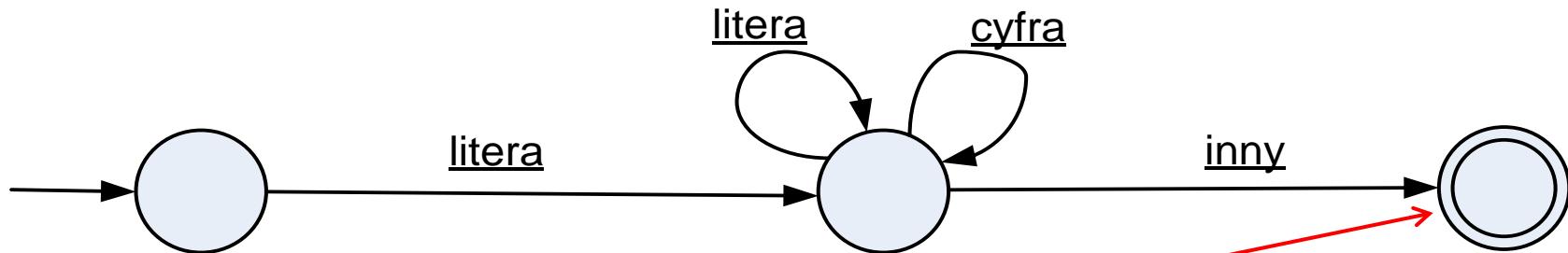


„oddaj” ostatni przeczytany symbol na wejście;
sprawdź, czy leksem to słowo kluczowe;
jeśli tak – zwróć odpowiednie słowo kluczowe;
jeśli nie – sprawdź, czy identyfikator jest
już w tablicy symboli;
jeśli jest – zwróć adres jego pozycji;
jeśli nie ma – utwórz nową pozycję, wpisz
identyfikator do tablicy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

Diagramy przejść (4)

Identyfikatory

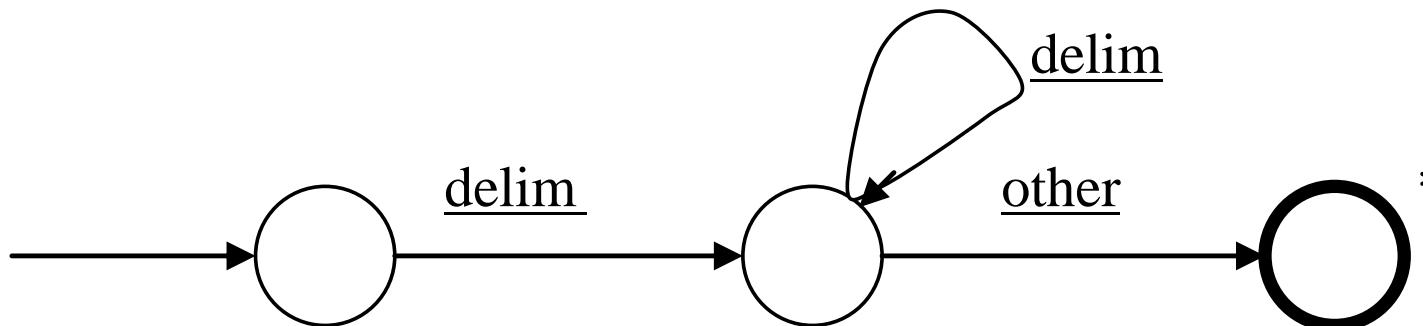
litera (litera | cyfra)*



„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

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 możliwe najdłuższemu leksemowi.
- W przypadku, gdy ten sam leksem odpowiada więcej niż jednemu tokenowi, uzgadniany jest token odpowiadający najwcześniej 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

gdzię:

- 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**

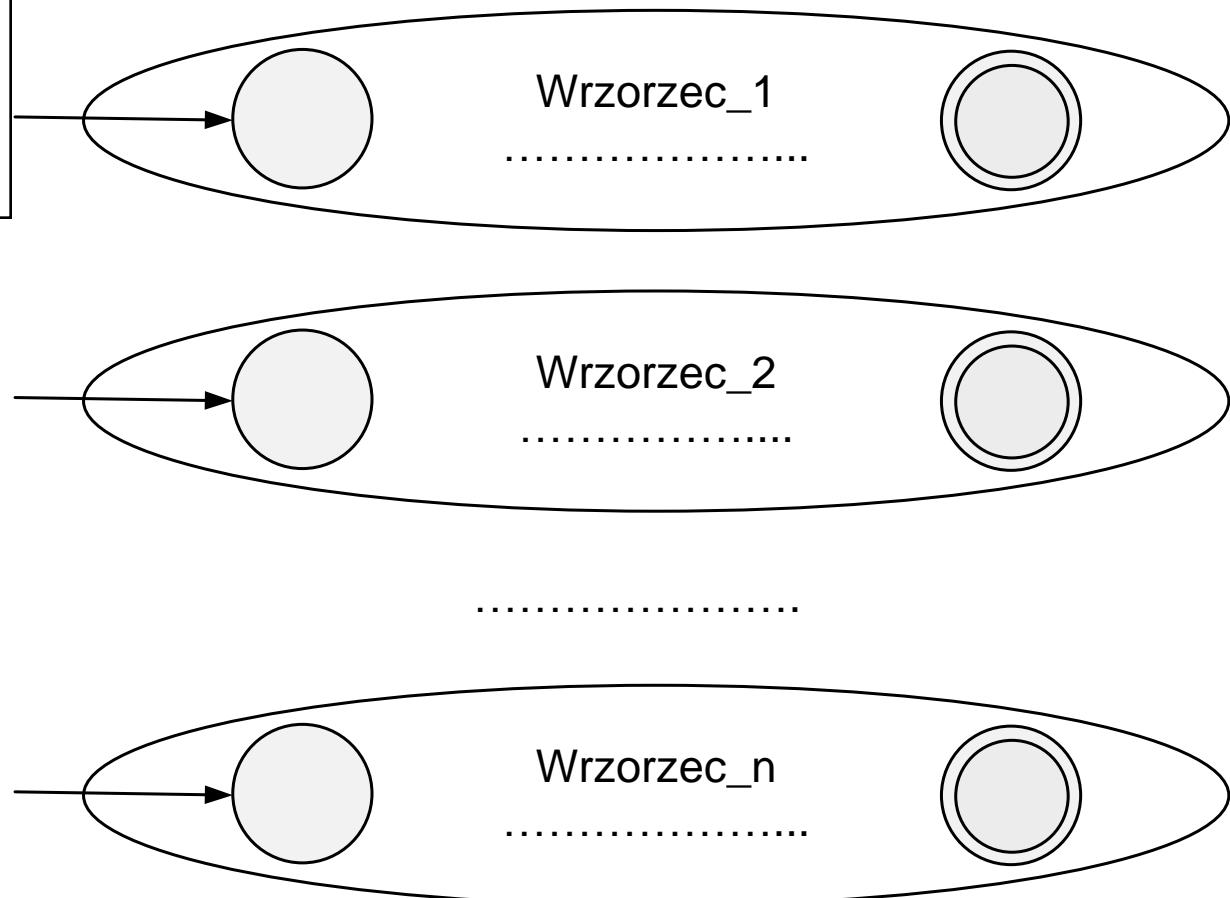
2 1

gdzięż:

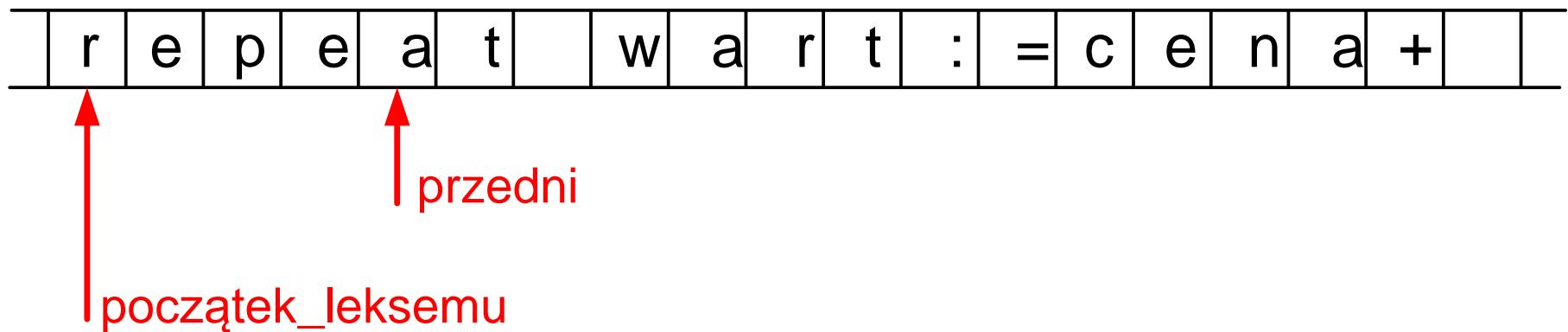
- W przypadku, gdy ten sam leksem odpowiada więcej niż jednemu tokenowi, uzgadniany jest token odpowiadający najwcześniej z „pasujących” pozycji specyfikacji.

Izolowane automaty deterministyczne

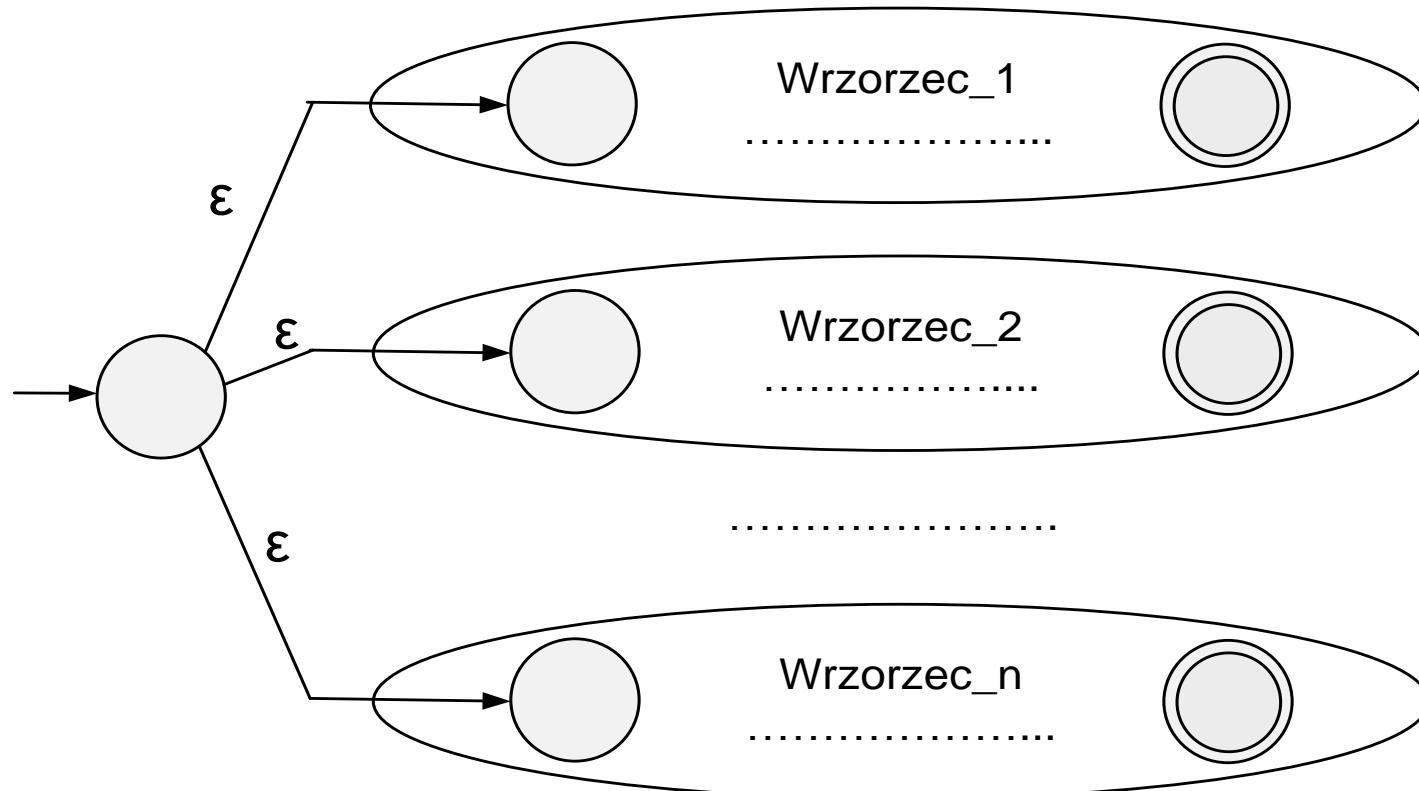
wzorzec_1	{akcja_1}
wzorzec_2	{akcja_2}
.....
wzorzec_n	{akcja_n}



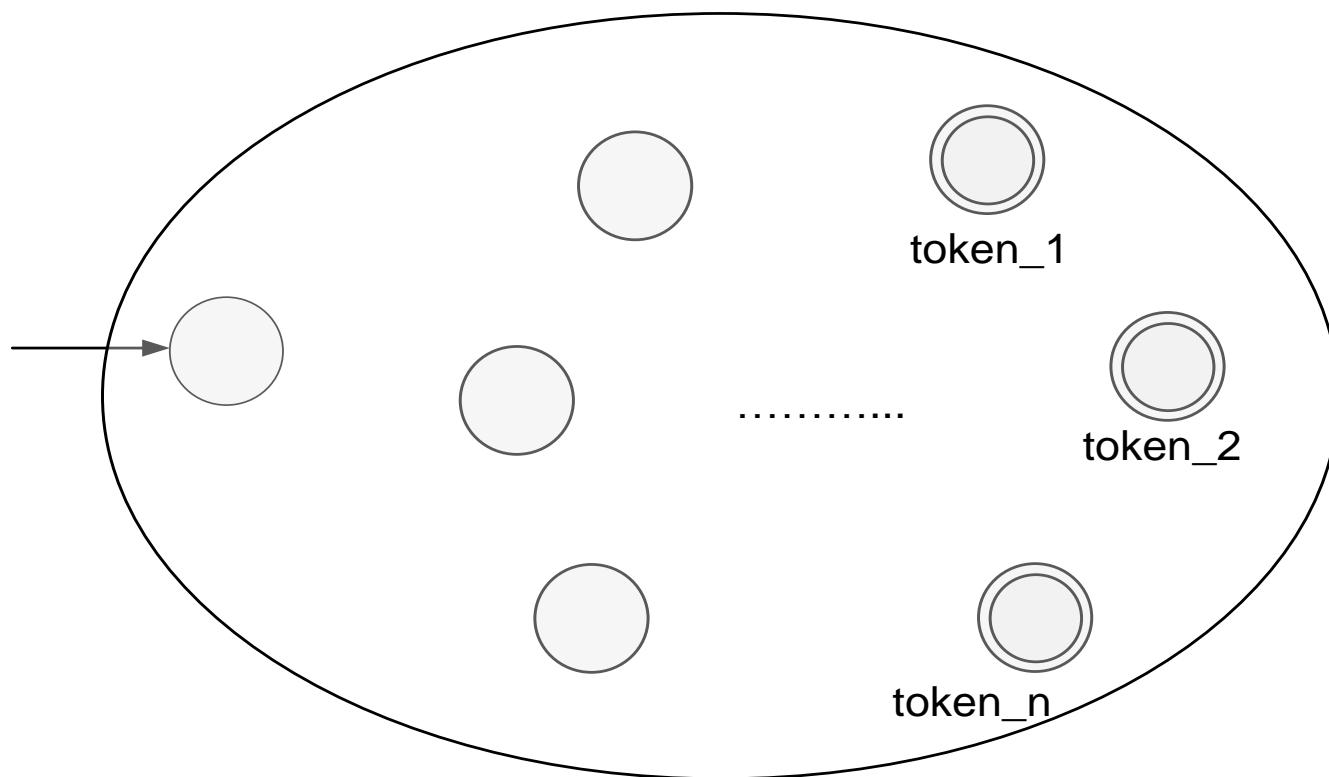
Najprostsza implementacja – algorytm z powrotami



Kombinowany automat niedeterministyczny



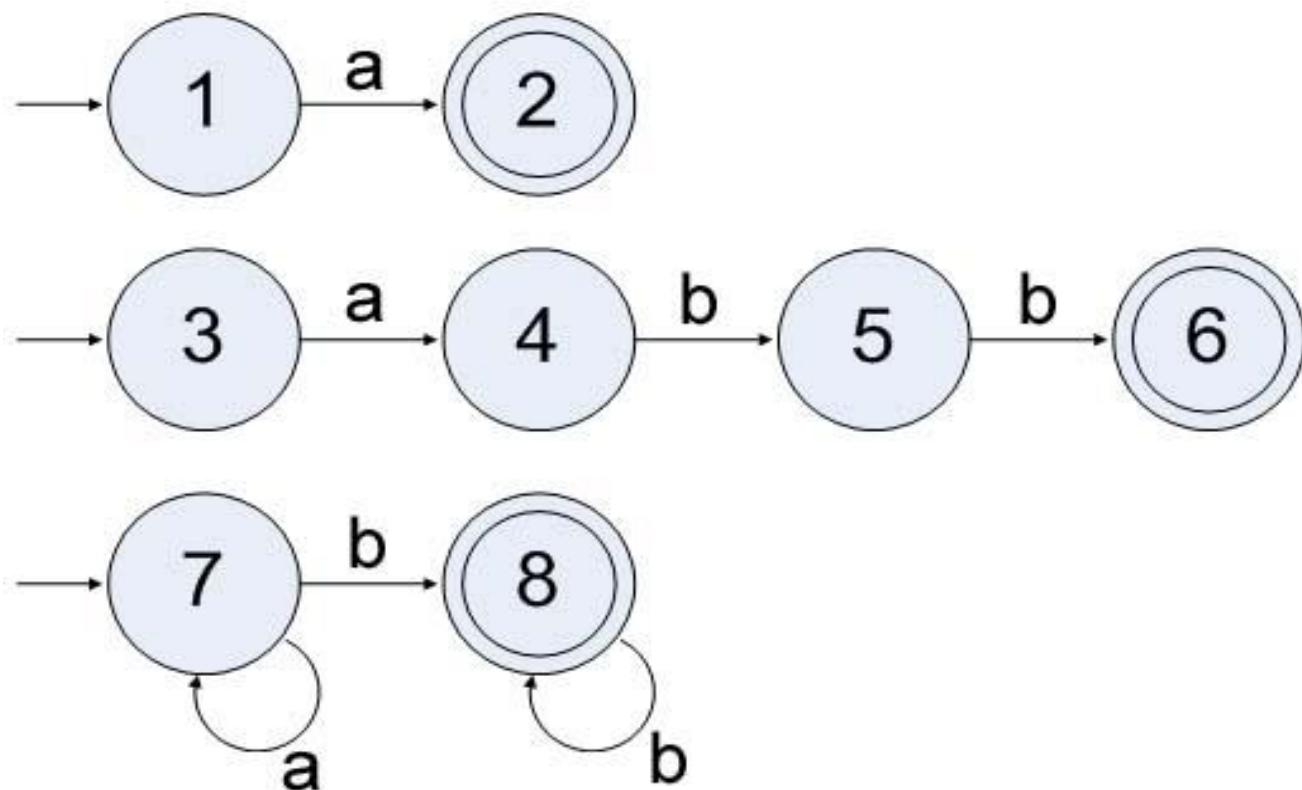
Automat deterministyczny



Budowa automatu skońzonego (1)

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

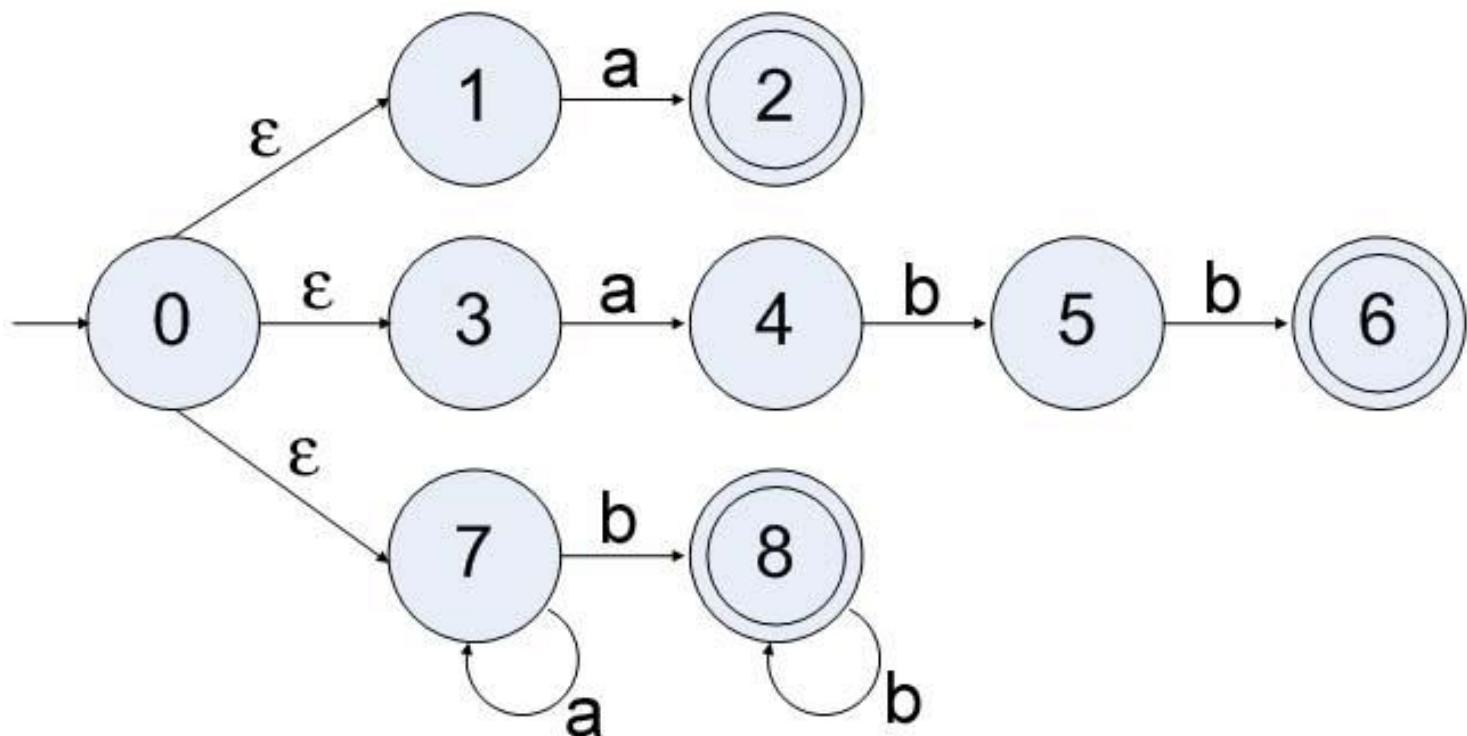
pojedyncze NFA



Budowa automatu skońzonego (2)

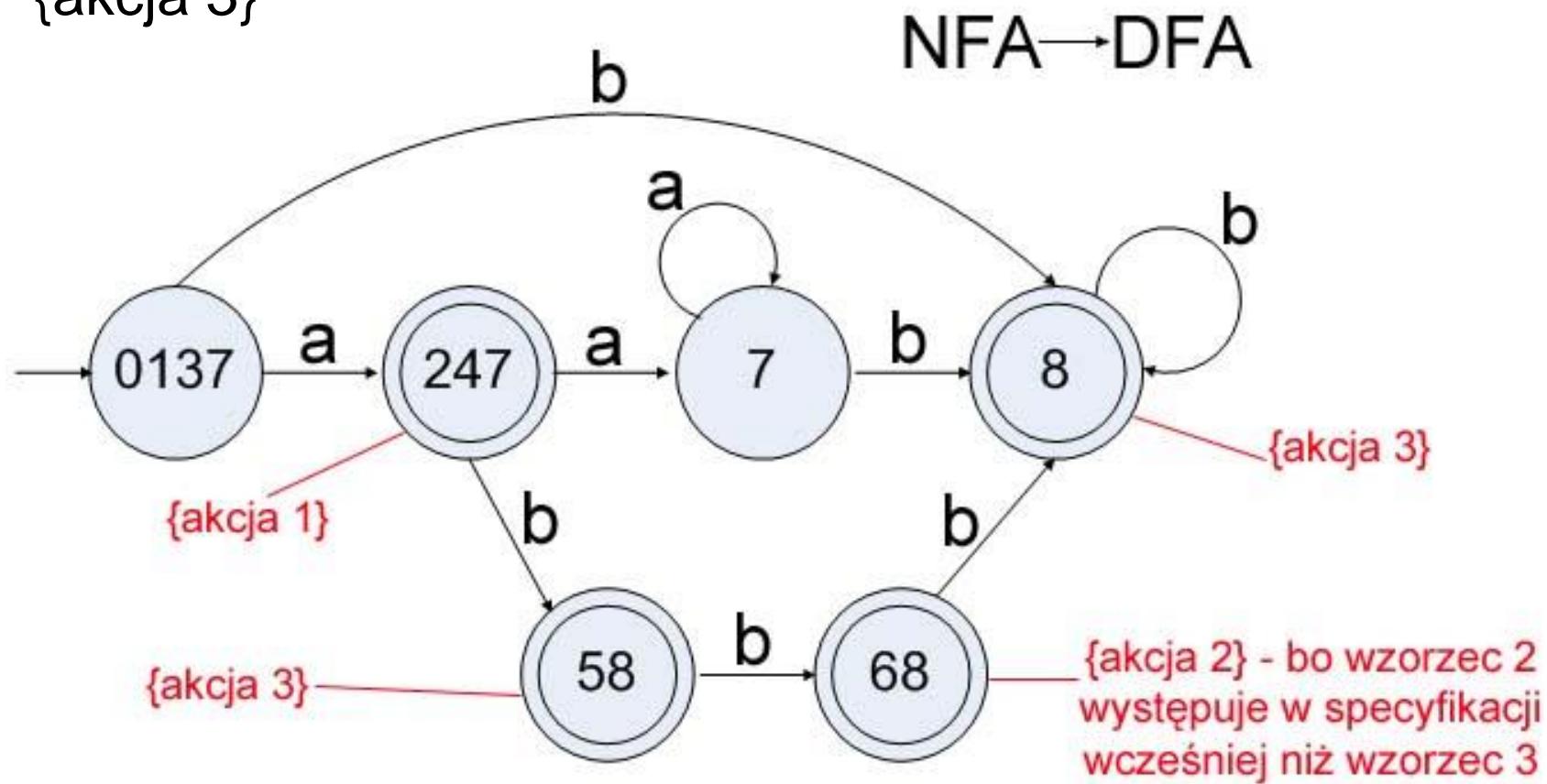
a {akcja 1}
abb {akcja 2}
 a^*b^+ {akcja 3}

kombinowany NFA



Budowa automatu skońzonego (3)

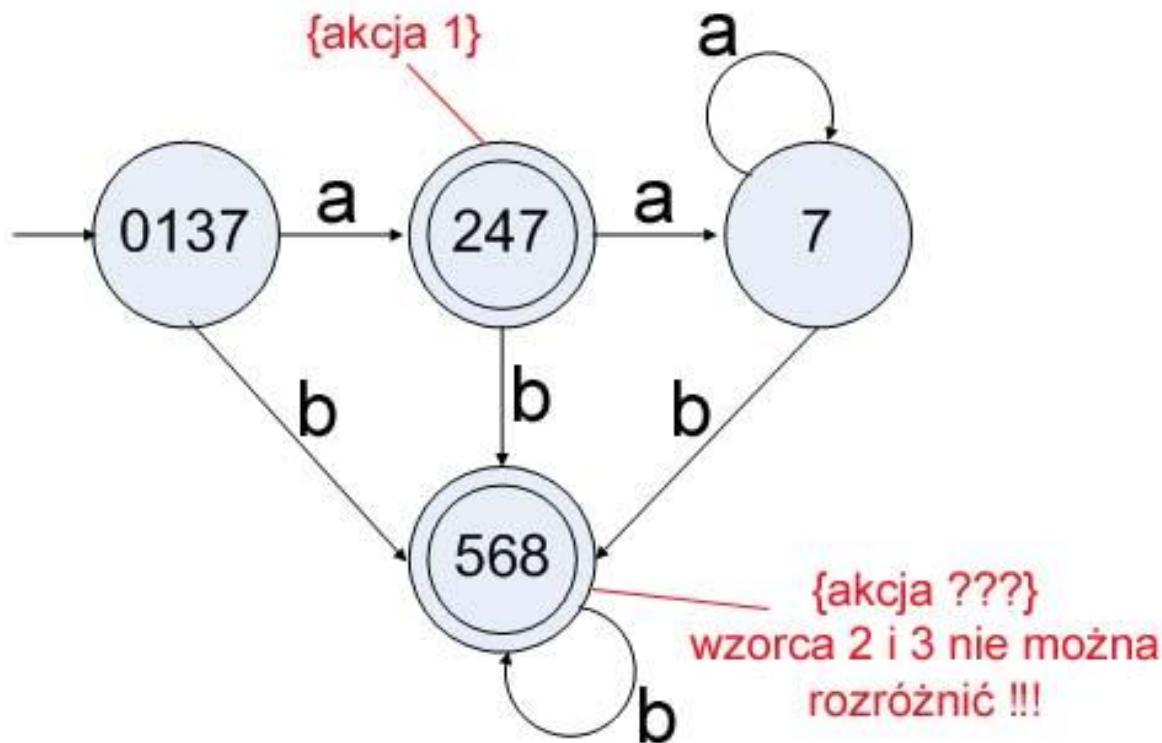
- a {akcja 1}
- abb {akcja 2}
- a^*b^+ {akcja 3}



Budowa automatu skońzonego (4)

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

optymalizacja DFA



Implementacja skanera (1)

Wymagania:

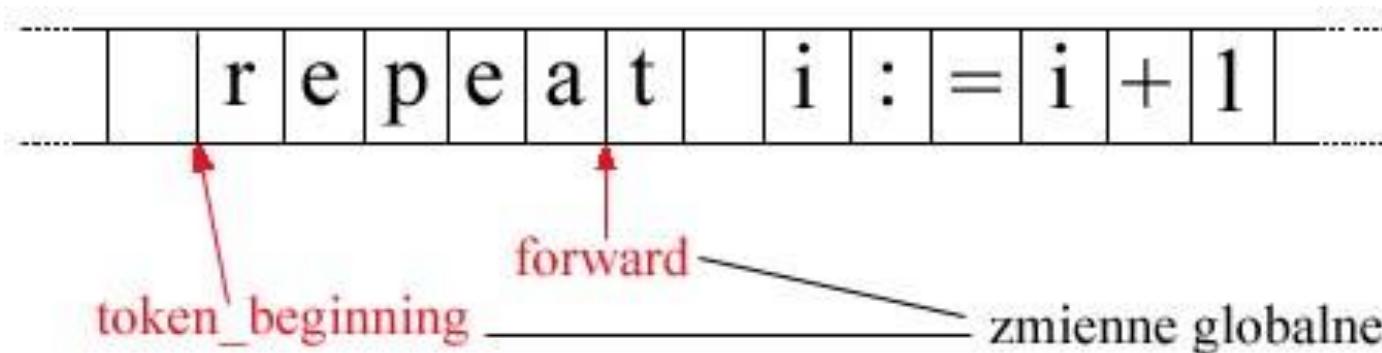
- - minimalna ilość operacji przypadających na jeden znak czytany z wejścia
- - rozsądna zajętość pamięci

Sposoby realizacji:

(a) schemat blokowy analizatora \equiv 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;
```

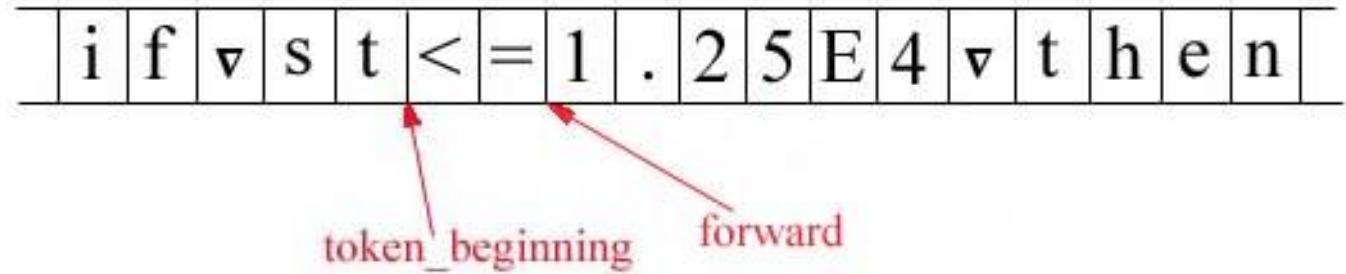
```
    case 12: err_recover();
```

```
        break;
```

```
}
```

```
    return (start);
```

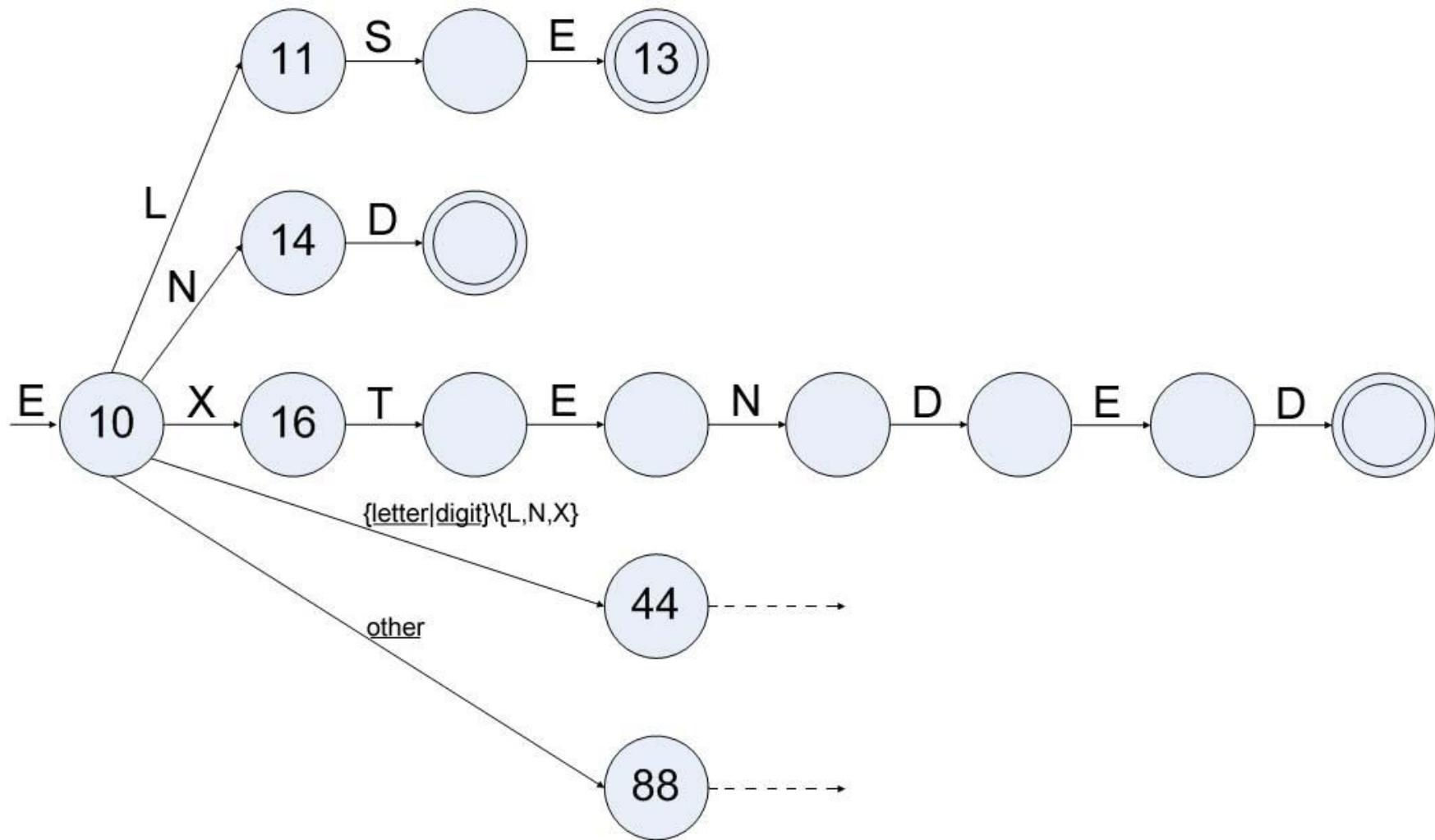
```
}
```



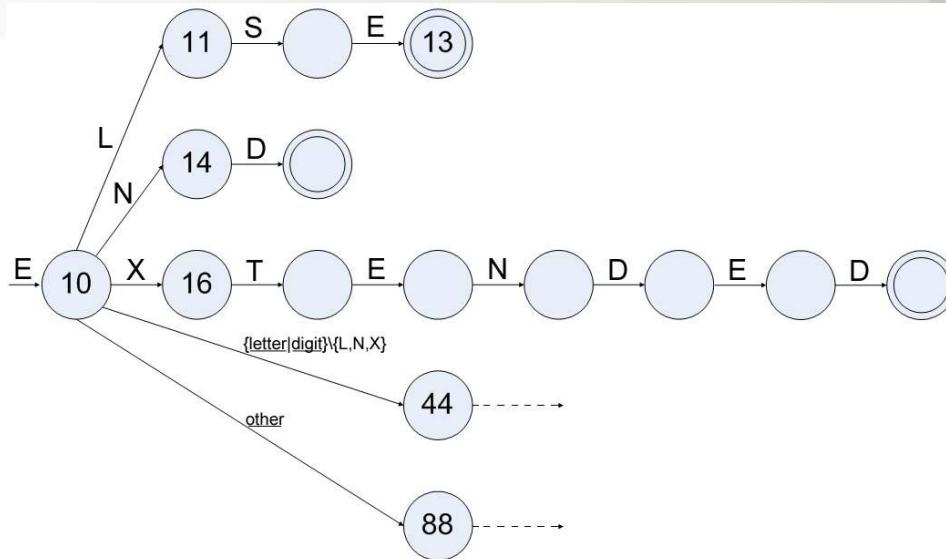
Implementacja skanera (7)

(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(<u>else</u>)	(start state)
...

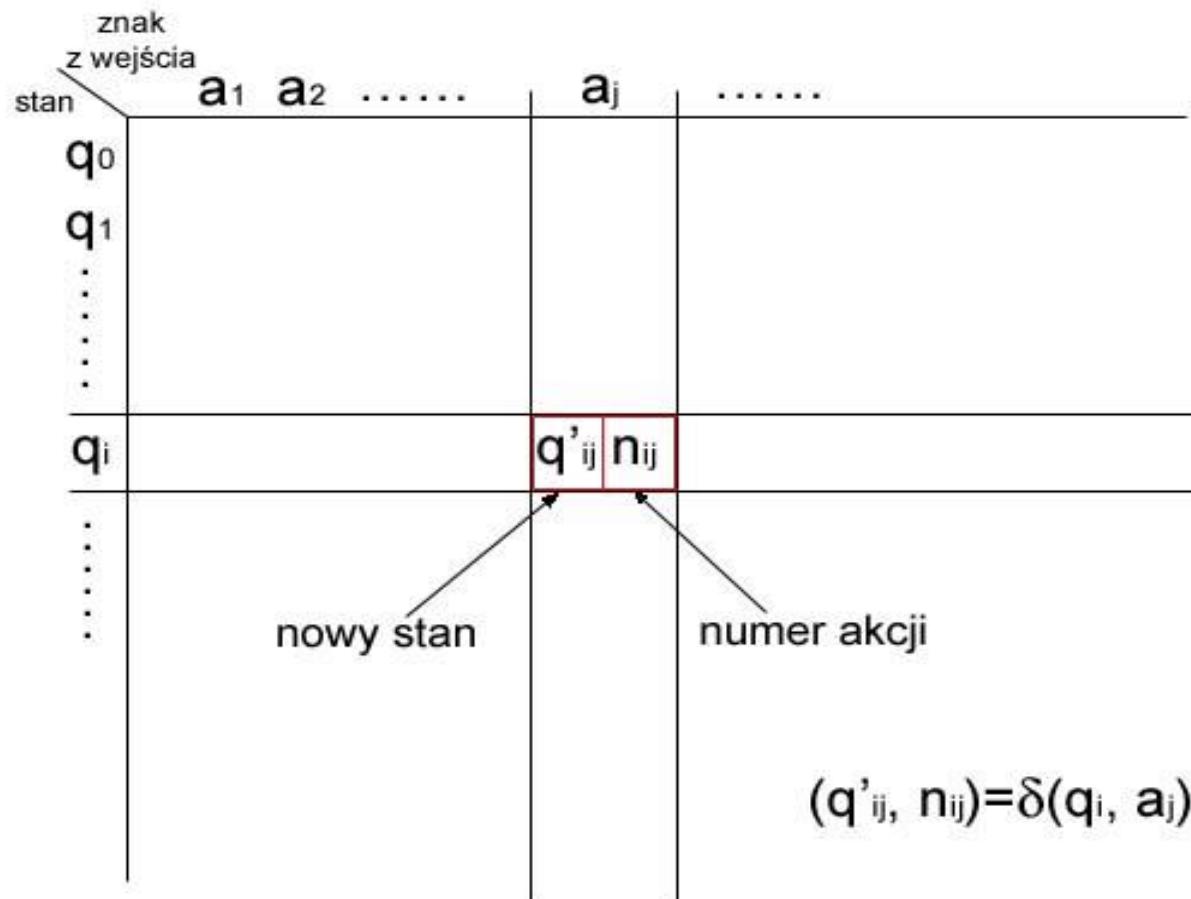
Check	input char.	yes=next state	no=check
1000	L	11	1001
1001	N	14	1002
1002	X	16	1003
1003	letter digit	44	1004
1004	other	88	

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

Implementacja skanera (10)

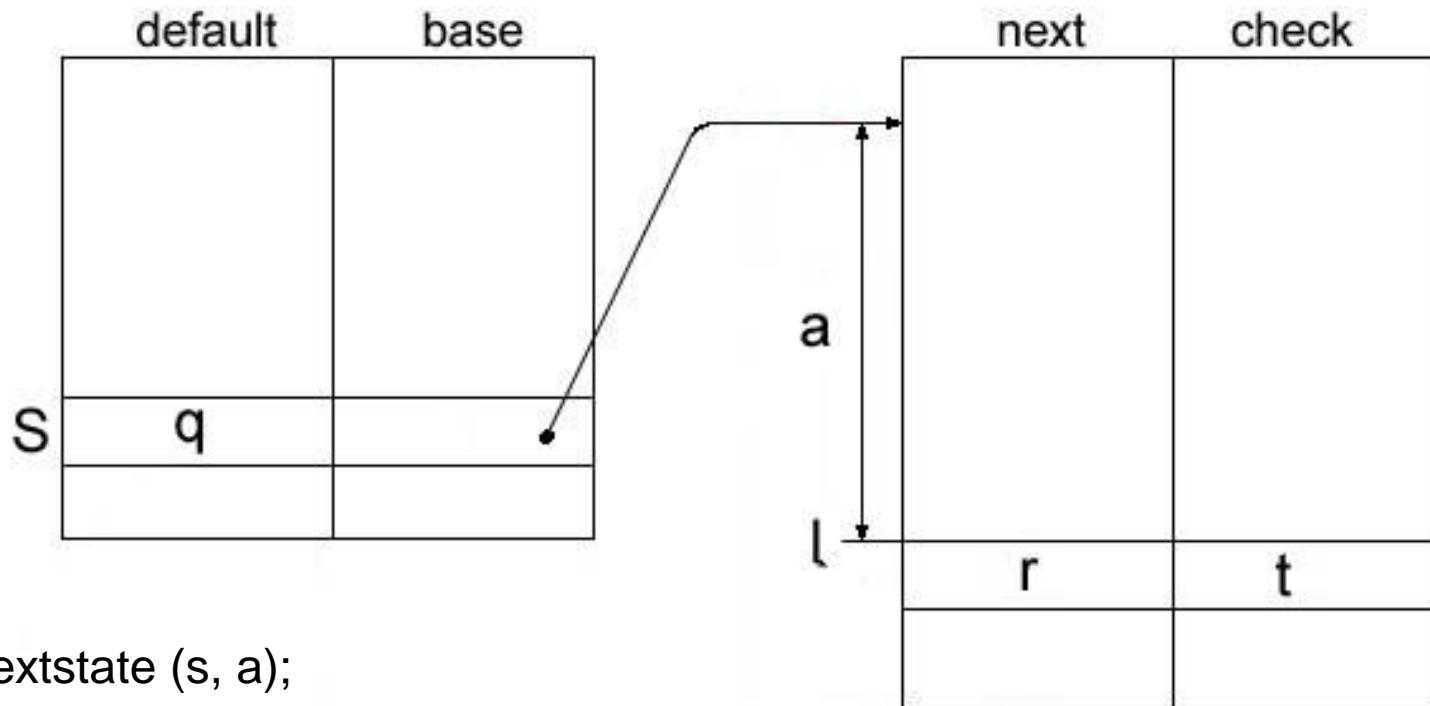
(c) dwuwymiarowa tablica do realizacji funkcji przejścia

(organizacja efektywna czasowo, ale bardzo pamięciochłonna; duża regularność i łatwość automatycznej generacji)



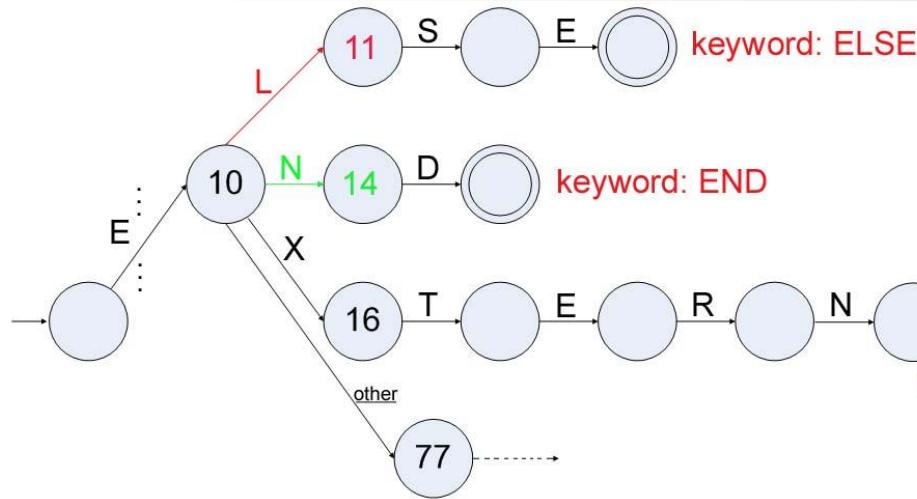
Implementacja skanera (11)

(d) organizacja mieszana

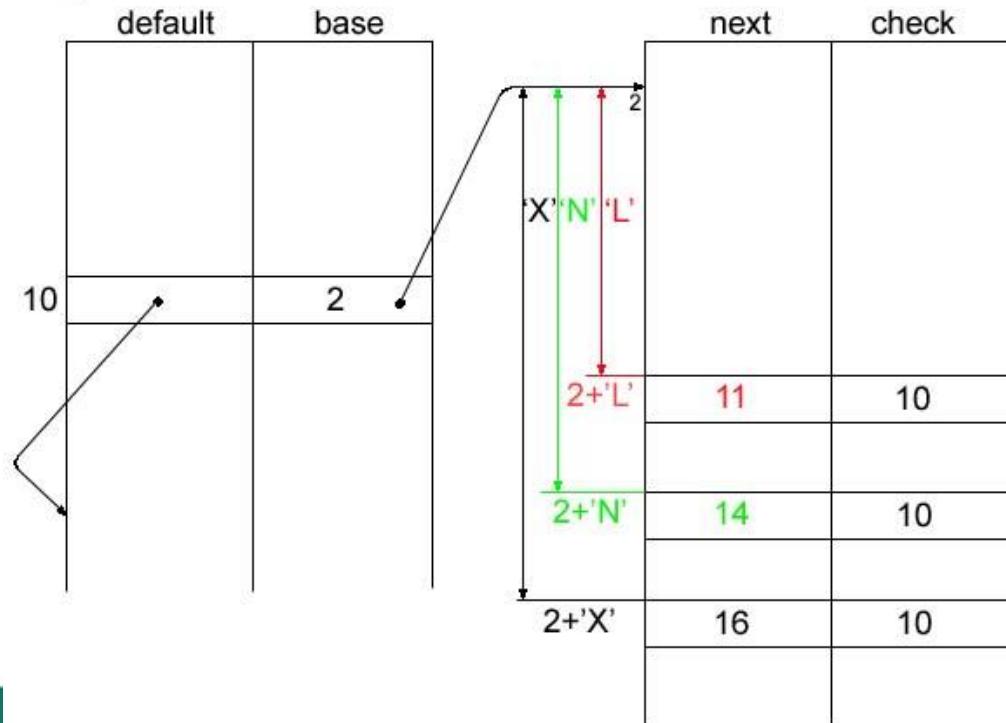


```
procedure nextstate (s, a);
    l:=base[s]+a;
    if check[l]=s then
        return next[l]
    else
        return nextstate (default[s], a);
```

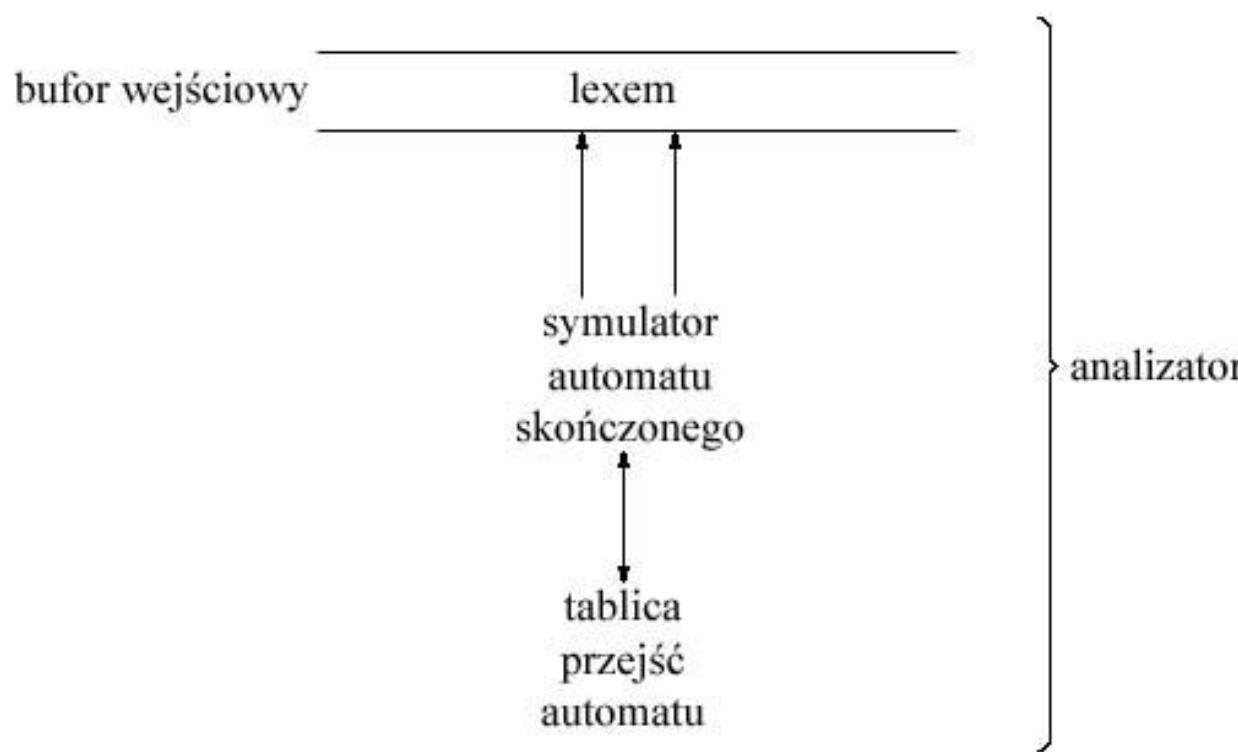
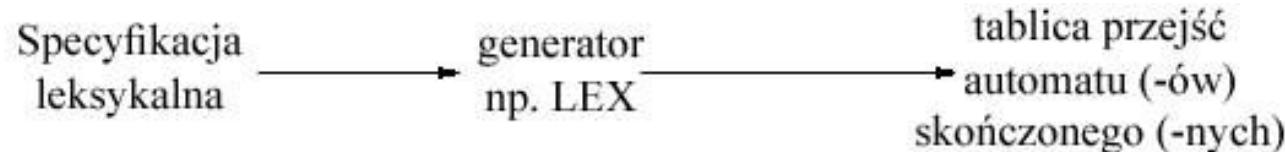
Implementacja skanera (12)



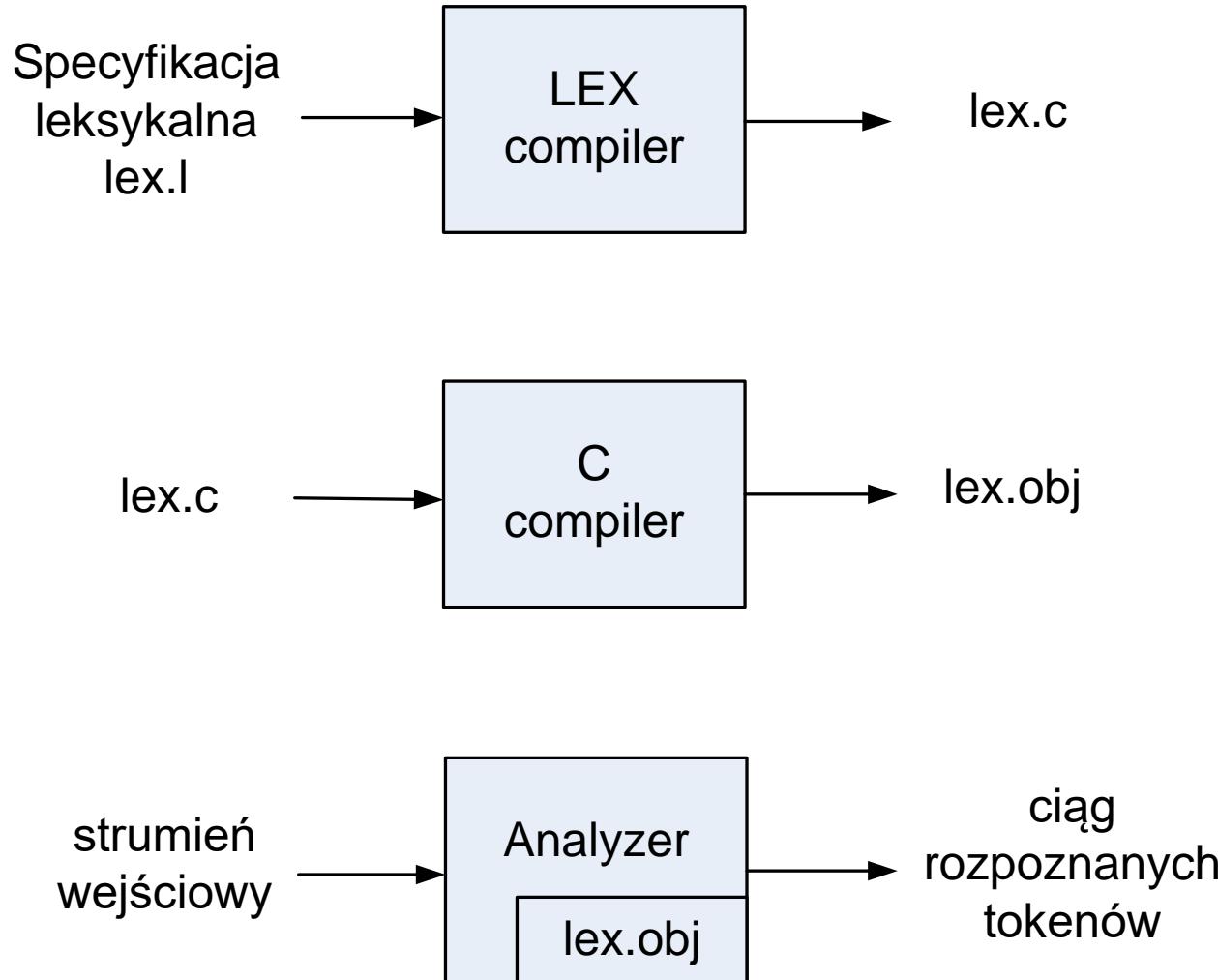
(organizacja stanowiąca kompromis pomiędzy dużą pamięciochłonnością organizacji czysto tablicowej (c), a czasochłonnością metod opartych na listach (b); wadą metody jest skomplikowany algorytm budowania tablic „base”, „default”, „next” i „check”)



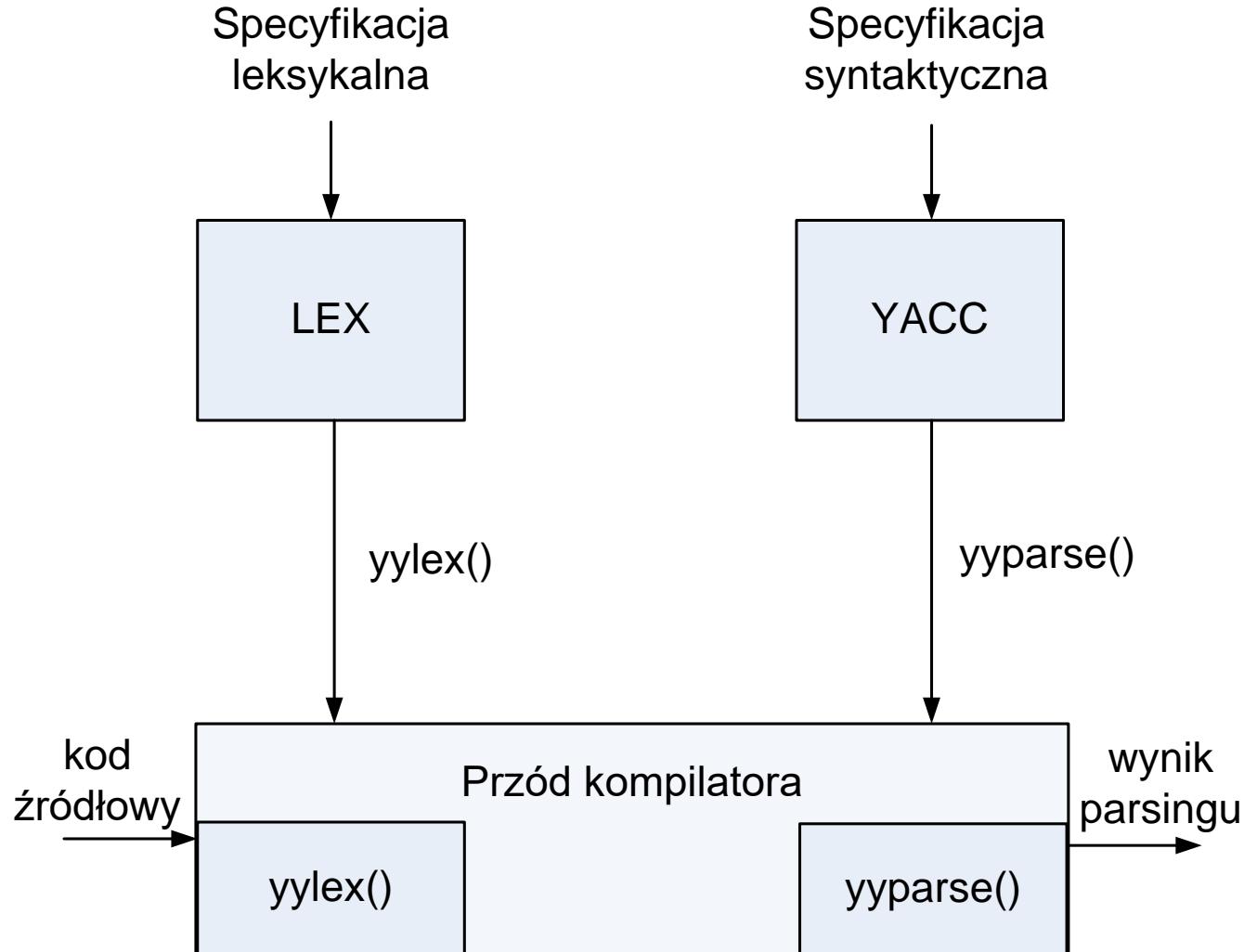
Generatory analizatorów leksykalnych



Generatory analizatorów leksykalnych



Generatory analizatorów leksykalnych i syntaktycznych



„Język” tworzenia specyfikacji leksykalnych

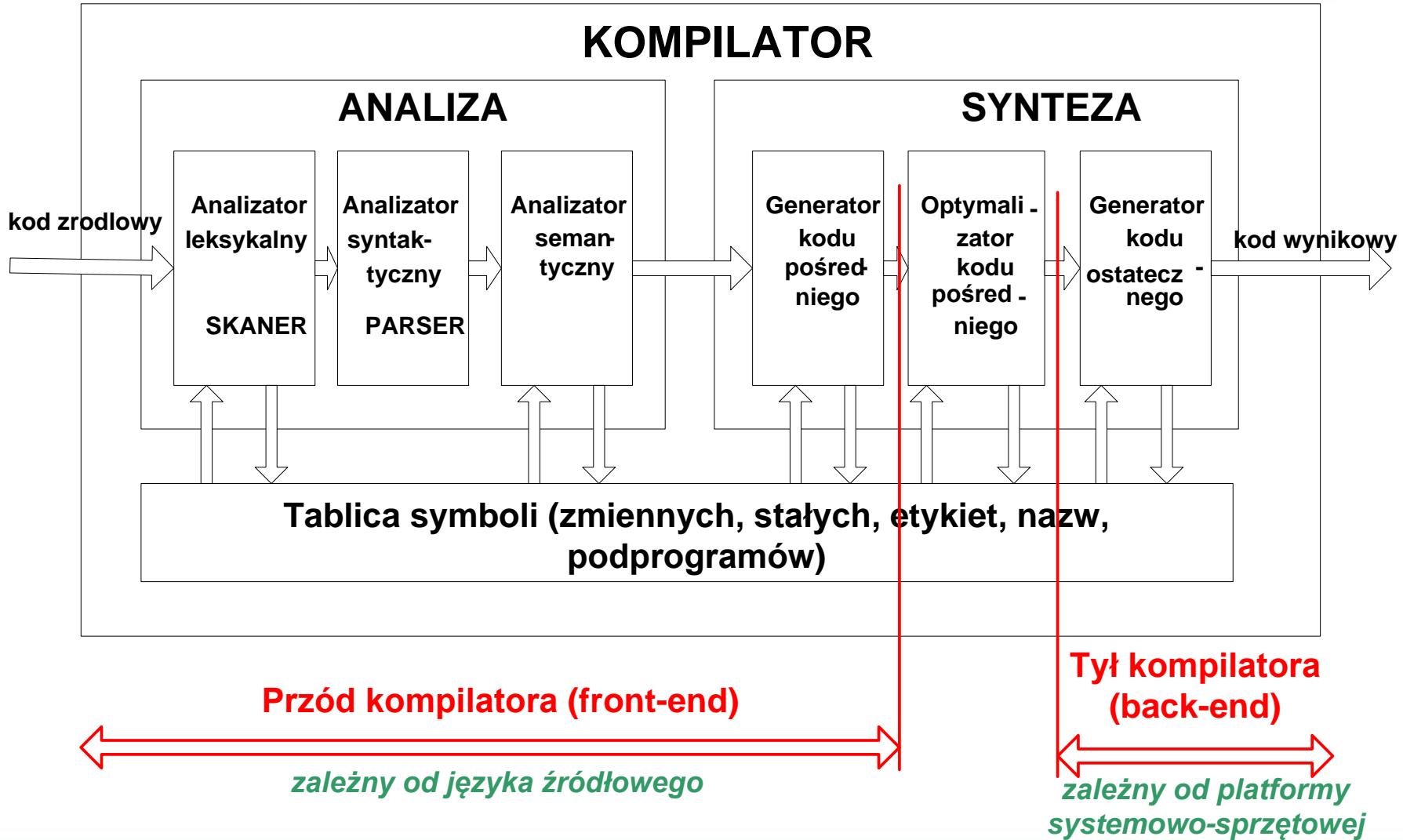
x	znak „x”
„x”	znak „x” nawet gdy x jest operatorem
\x	znak „x” nawet gdy x jest operatorem
[xy]	znak „x” lub „y”
[x-z]	znak „x” lub „y” lub „z”
[^x]	dowolny znak z wyjątkiem „x”
.	dowolny znak z wyjątkiem „\n”
^x	znak „x” na początku linii
<y>x	znak „x” gdy LEX jest w stanie początkowym „y”
x\$	znak „x” na końcu linii
x?	0 lub 1 wystąpienie „x”
x*	0, 1, 2,... wystąpień „x”
x+	1, 2,... wystąpienie „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”

Analiza syntaktyczna

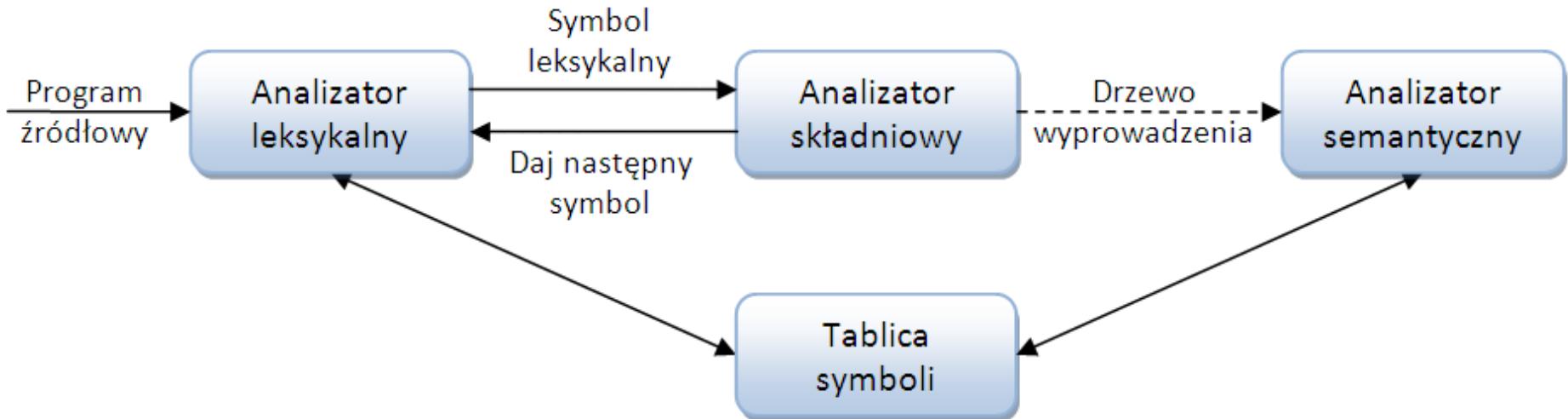
Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Przypomnienie: struktura kompilatora



Współpraca parsera z innymi analizatorami



Analizator składniowy (parser) jest modułem wykonującym analizę syntaktyczną oraz zarządzającym całym przodem kompilatora, a w szczególności całym blokiem analizy. W takt pracy parsera działa zarówno analizator leksykalny (dostarczając parserowi kolejne tokeny), jak i analizator semantyczny (analizując np. poprawność typów) oraz generator kodu pośredniego (tłumacząc zanalizowane fragmenty programu do kodu pośredniego).

Zadanie analizy syntaktycznej



Analizator syntaktyczny (parser) ma za zadanie dokonać rozboru gramatycznego tekstu źródłowego analizowanego programu. Analizator leksykalny (skaner) dokonuje zamiany tekstu wejściowego na ciąg tokenów. Parser pracuje na podstawie gramatyki syntaktycznej. Buduje on drzewo rozboru analizowanego ciągu tokenów, czyli przeprowadza wyprowadzenie łańcucha tokenów w gramatyce składniowej. Od parsera oczekujemy jakiegoś zanotowania tego wywodu. Często stosowanym sposobem notowania wyprowadzenia jest podanie ciągu numerów produkcji, na podstawie którego można odtworzyć drzewo rozboru.

Zadanie analizy syntaktycznej c.d.

Niekiedy równolegle z analizą syntaktyczną parser lub inicjuje wykonanie:

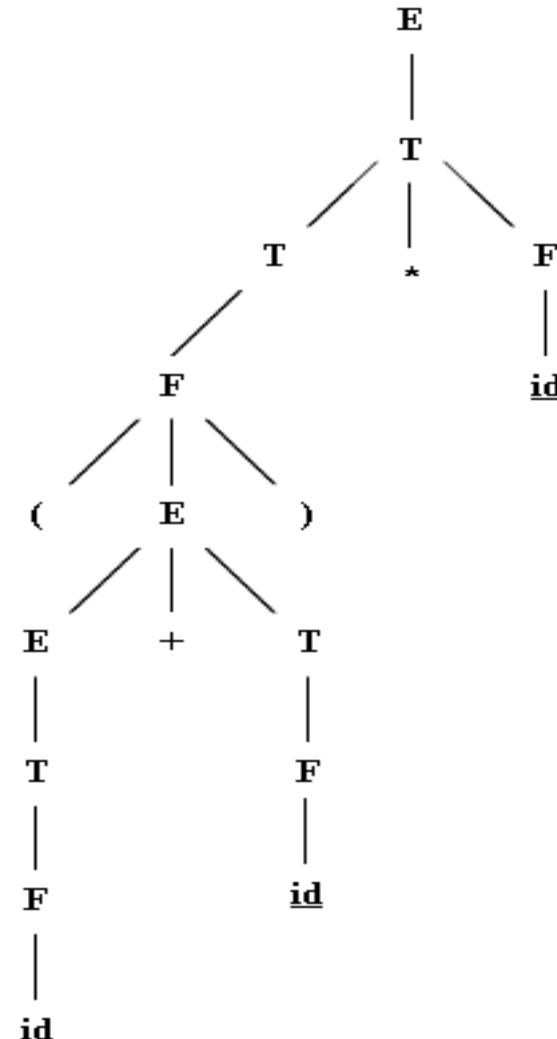
- pewnych akcji semantycznych, jak np. będącej czynnością o charakterze analitycznym kontroli typów, i/lub
- pewnych akcji o charakterze syntetycznym, jak np. generowania kodu pośredniego w postaci ONP, tetrad (n-tek), drzew składniowych itd.

Działanie parsera:

- We: $\omega \in \Sigma_S^*$ (ω jest słowem nad alfabetem Σ_S , Σ_S jest zbiorem tokenów)
- Wy: $\pi = p_1 \dots p_i \dots p_n$ (p_i – numer produkcji) lub err (błąd syntaktyczny)

Gramatyka składniowa:

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$
- 3) $T \rightarrow T^* F$
- 4) $T \rightarrow F$
- 5) $F \rightarrow (E)$
- 6) $F \rightarrow \underline{id}$



Analizowane słowo: $(\underline{id}+\underline{id})^*\underline{id}$

Do zbudowania drzewa rozbioru syntakt. (podstawa konstrukcji kodu wynikowego) niezbędna jest wiadomość, że ciąg numerów produkcji związany jest z wywodem (tutaj) lewostronnym tworzonym metodą top-down

	E
(2)	$\Rightarrow T$
(3)	$\Rightarrow T^*F$
(4)	$\Rightarrow F^*F$
(5)	$\Rightarrow (E)^*F$
(1)	$\Rightarrow (E+T)^*F$
(2)	$\Rightarrow (T+T)^*F$
(4)	$\Rightarrow (F+T)^*F$
(6)	$\Rightarrow (\underline{id}+T)^*F$
(4)	$\Rightarrow (\underline{id}+F)^*F$
(6)	$\Rightarrow (\underline{id}+\underline{id})^*F$
(6)	$\Rightarrow (\underline{id}+\underline{id})^*\underline{id} \Rightarrow acc$

ciąg numerów produkcji jest wyjściem z parsera

Tworzenie parsera

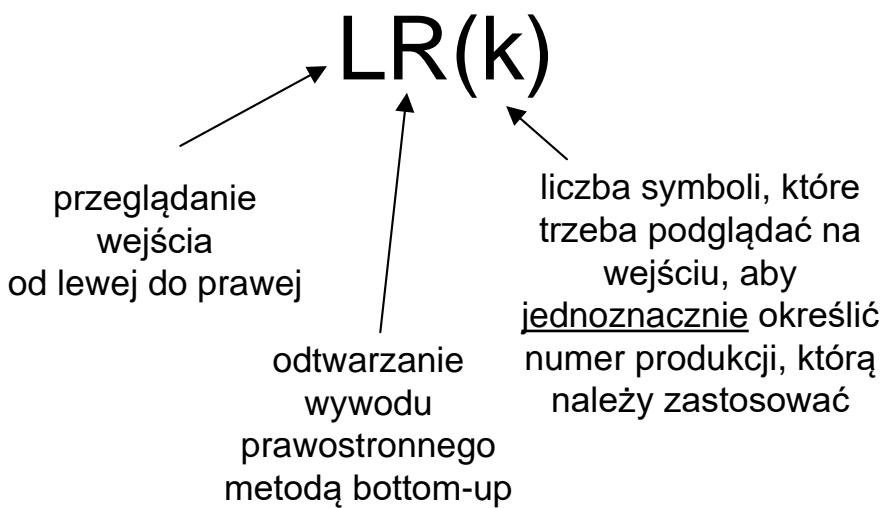
Generacja parsera:

- We: gramatyka syntaktyczna $G_S = \langle V_S, \Sigma_S, P_S, S_S \rangle$ (oznaczana dalej $G = \langle V, \Sigma, P, S \rangle$)
- Wy: algorytm parsera

Algorytm parsera musi być efektywny. Uniwersalny algorytm Earley'a ma dla gramatyki bezkontekstowej (bez ograniczeń na jednoznaczność) złożoność obliczeniową: czasową $O(n^3)$, pamięciową $O(n^2)$, zaś dla gramatyk jednoznacznych złożoność czasowa wynosi $O(n^2)$. Są to złożoności niedopuszczalne w praktyce. Algorytm Cocke'a-Youngera-Kasamiego (CYK) (także bez ograniczeń na jednoznaczność) ma złożoność obliczeniową: czasową $O(n^3)$, pamięciową $O(n^2)$. Musimy się posługiwać algorytmami prostszymi, ale wiążę się to z zawężeniem klasy gramatyk.

Dalej rozpatrywane będą algorytmy parsingu dla gramatyk LL(k) i LR(k). Dla gramatyk LL(k) i LR(k) złożoność obliczeniowa algorytmu parsera wynosi: czasowa $O(n)$ oraz pamięciowa $O(n)$.

Gramatyki LL i LR



Na ogół stosowane są gramatyki:

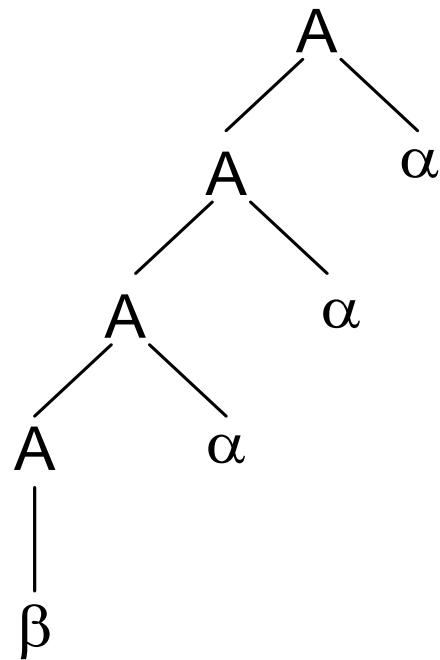
- LL(1)
- LR(1) – oraz jej podklasy: SLR(1) lub LALR(1), dające możliwość znacznego uproszczenia algorytmu parsingu przy niewielkim ograniczeniu „możliwości” gramatyki.

Usuwanie lewostronnej rekurencji

Mówimy, że gramatyka jest lewostronnie rekurencyjna, jeśli ma taki nieterminal A , że istnieje wyprowadzenie $A \Rightarrow^+ A\alpha$ dla pewnego niepustego łańcucha α .

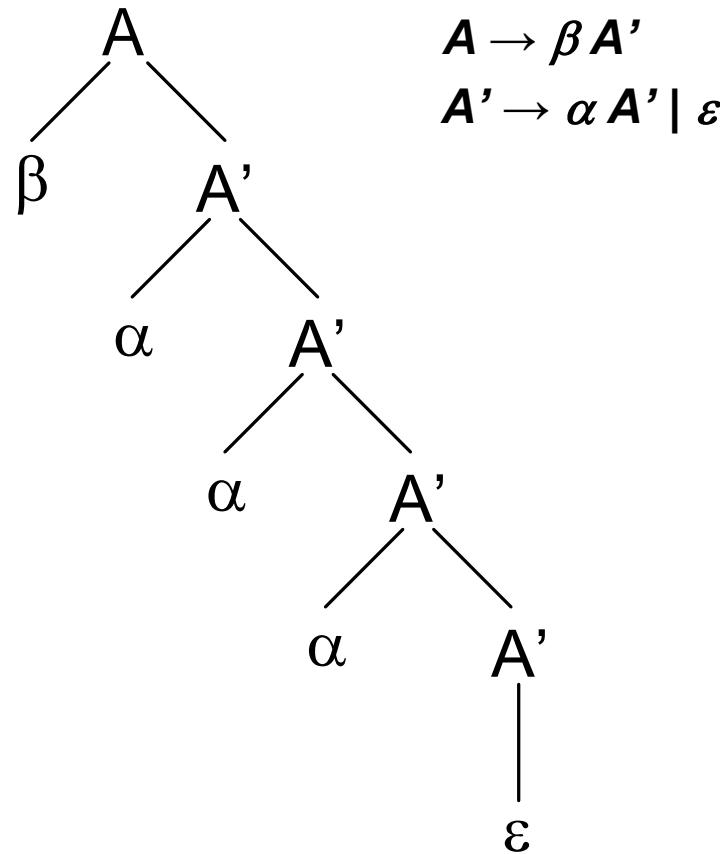
Jeśli mamy produkcje:

$$A \rightarrow A\alpha \mid \beta$$



To możemy je zastąpić przez:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$



Przykład

$$A \rightarrow A\alpha | \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \varepsilon$$

$$1) E \rightarrow E + T$$

$$1) E \rightarrow TE'$$

$$2) E \rightarrow T$$

$$2) E' \rightarrow +TE'$$

$$3) T \rightarrow T^*F$$

$$4) T \rightarrow FT'$$

$$4) T \rightarrow F$$

$$5) T' \rightarrow *FT'$$

$$5) F \rightarrow (E)$$

$$7) F \rightarrow (E)$$

$$6) F \rightarrow \underline{id}$$

$$8) F \rightarrow \underline{id}$$

Zbiory FIRST_k

Zbiór FIRST_k(α) jest zbiorem wszystkich terminalnych przedrostków długości k (lub mniejszej, jeżeli z α wyprowadza się łańcuch terminalny krótszy niż k) łańcuchów, które mogą być wyprowadzalne z α

Formalnie:

Niech $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

Definiujemy:

$$\text{FIRST}_k(\alpha) = \{x \in \Sigma^* \mid (\alpha \Rightarrow^* x\beta \wedge |x|=k) \vee (\alpha \Rightarrow^* x \wedge |x|< k); \quad \alpha, \beta \in (V \cup \Sigma)^*\}$$

Zbiory FOLLOW_k

Zbiór FOLLOW_k(β) zawiera terminalnełańcuchy o długości k (lub mniejszej—jak poprzednio), które mogą pojawić się w wyprowadzeniach jako następcy β.

Uwaga: jeśli FOLLOW_k(β) zawiera x , $|x| < k$, to wówczas zastępujemy x przez $x\$$, gdzie \$—prawy ogranicznik słowa (dla wygody).

Formalnie:

Niech $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

Definiujemy:

$$\begin{aligned} \text{FOLLOW}_k(\beta) = \\ \{x \in \Sigma^*: (S \Rightarrow^* \alpha\beta\gamma \wedge x \in \text{FIRST}_k(\gamma); \alpha, \beta, \gamma \in (V \cup \Sigma)^*)\} \end{aligned}$$

Wyznaczanie FIRST₁(x) dla x ∈ (V ∪ Σ)

```
1) if x ∈ Σ then FIRST1(x) := {x} ;  
2) if (x → ε) ∈ P then FIRST1(x) := FIRST1(x) ∪ {ε} ;  
3) if x ∈ V and (x → y1y2...yk) ∈ P then  
begin  
    FIRST1(x) := FIRST1(x) ∪ (FIRST1(y1) - {ε}) ;  
    for i := 2 to k do  
        if (ε ∈ FIRST1(y1)) and ... and (ε ∈ FIRST1(yi-1))  
        then FIRST1(x) := FIRST1(x) ∪ (FIRST1(yi) - {ε}) ;  
        if (ε ∈ FIRST1(y1)) and ... and (ε ∈ FIRST1(yk)) then  
            FIRST1(x) := FIRST1(x) ∪ {ε}  
end ;
```

Aby wyznaczyć FIRST₁(x) dla wszystkich x ∈ (V ∪ Σ) należy stosować reguły (1), (2) i (3) tak długo, aż nic nowego nie da się dołączyć do któregokolwiek zbioru FIRST₁(x).

Przykład (1)

- 1) $E \rightarrow TE'$
 - 2) $E' \rightarrow +TE'$
 - 3) $E' \rightarrow \epsilon$
 - 4) $T \rightarrow FT'$
 - 5) $T' \rightarrow *FT'$
 - 6) $T' \rightarrow \epsilon$
 - 7) $F \rightarrow (E)$
 - 8) $F \rightarrow \underline{id}$
- }
- $\text{FIRST}_1(F) = \{ (, \underline{id} \}$

Przykład (2)

- 1) $E \rightarrow TE'$
 - 2) $E' \rightarrow +TE'$
 - 3) $E' \rightarrow \epsilon$
 - 4) $T \rightarrow FT' \xrightarrow{\text{orange}} \text{FIRST}_1(T) = \text{FIRST}_1(F) = \{(), \underline{id}\}$
 - 5) $T' \rightarrow *FT'$
 - 6) $T' \rightarrow \epsilon$
 - 7) $F \rightarrow (E)$
 - 8) $F \rightarrow \underline{id}$
- }
- $\text{FIRST}_1(F) = \{(), \underline{id}\}$

Przykład (3)

- 1) $E \rightarrow TE' \xrightarrow{\text{orange}} \text{FIRST}_1(E) = \text{FIRST}_1(T) = \{(, \underline{id}\}$
 - 2) $E' \rightarrow +TE'$
 - 3) $E' \rightarrow \epsilon$
 - 4) $T \rightarrow FT' \xrightarrow{\text{light blue}} \text{FIRST}_1(T) = \text{FIRST}_1(F) = \{(, \underline{id}\}$
 - 5) $T' \rightarrow *FT'$
 - 6) $T' \rightarrow \epsilon$
 - 7) $F \rightarrow (E)$
 - 8) $F \rightarrow \underline{id}$
- }
- $\text{FIRST}_1(F) = \{(, \underline{id}\}$

Przykład (4)

- 1) $E \rightarrow TE' \xrightarrow{\text{ }} \text{FIRST}_1(E) = \text{FIRST}_1(T) = \{(, \underline{id}\}$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \varepsilon$
- 4) $T \rightarrow FT' \xrightarrow{\text{ }} \text{FIRST}_1(T) = \text{FIRST}_1(F) = \{(, \underline{id}\}$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \varepsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow \underline{id}$
- $\} \quad \text{FIRST}_1(E') = \{+, \varepsilon\}$
- $\} \quad \text{FIRST}_1(T') = \{*, \varepsilon\}$
- $\} \quad \text{FIRST}_1(F) = \{(, \underline{id}\}$

Wyznaczanie FIRST₁(x₁x₂...x_n)

FIRST₁(x₁x₂...x_n) := FIRST₁(x₁) - {ε}

for i:=2 to k do

if (ε ∈ FIRST₁(x₁)) and...and (ε ∈ FIRST₁(x_{i-1}))

then

 FIRST₁(x₁...x_n) :=

 FIRST₁(x₁...x_n) ∪ (FIRST₁(x_i) - {ε}) ;

if (ε ∈ FIRST₁(x₁)) and...and (ε ∈ FIRST₁(x_n))

then FIRST₁(x₁...x_n) := FIRST₁(x₁...x_n) ∪ {ε} ;

Wyznaczanie zbiorów FOLLOW₁

Wyznaczenie FOLLOW₁ dla wszystkich A ∈ V

Stosować poniższe reguły (1), (2), (3) dopóty nic nowego nie może już zostać dodane do żadnego zbioru FOLLOW₁.

- 1) FOLLOW₁(S) := FOLLOW₁(S) ∪ { \$ }
- 2) if (A → αBβ) ∈ P then
FOLLOW₁(B) := FOLLOW₁(B) ∪ (FIRST₁(β) - { ε })
- 3) if (A → αB) ∈ P or ((A → αBβ) ∈ P and
(ε ∈ FIRST₁(β))) then
FOLLOW₁(B) := FOLLOW₁(B) ∪ FOLLOW₁(A) ;

Przykład (5)

- 1) $E \rightarrow TE' \xrightarrow{\text{orange arrow}} FOLLOW_1(E) \supseteq \{\$\}$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow \underline{id}$

Przykład (6)

- 1) $E \rightarrow TE'$  $\text{FOLLOW}_1(E) \supseteq \{\$\}$
- 2) $E' \rightarrow +TE'$  $\text{FOLLOW}_1(T) \supseteq \{+\}$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$  $\text{FOLLOW}_1(F) \supseteq \{*\}$
- 5) $T' \rightarrow *FT'$ 
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$  $\text{FOLLOW}_1(E) \supseteq \{()\}$
- 8) $F \rightarrow \underline{id}$

Czyli w tej chwili:

$\text{FOLLOW}_1(E) \supseteq \{\$, ()\}$
 $\text{FOLLOW}_1(T) \supseteq \{+\}$
 $\text{FOLLOW}_1(F) \supseteq \{*\}$

Przykład (7)

- 1) $E \rightarrow TE'$  $\text{FOLLOW}_1(E') \leftarrow \text{FOLLOW}_1(E)$
- 2) $E' \rightarrow +TE'$  $\text{FOLLOW}_1(T) \leftarrow \text{FOLLOW}_1(E)$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$  $\text{FOLLOW}_1(T') \leftarrow \text{FOLLOW}_1(T)$
- 5) $T' \rightarrow *FT'$  $\text{FOLLOW}_1(F) \leftarrow \text{FOLLOW}_1(T)$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow \underline{id}$

Poprzednio:

$\text{FOLLOW}_1(E) \supseteq \{\$,)\}$
 $\text{FOLLOW}_1(T) \supseteq \{+\}$
 $\text{FOLLOW}_1(F) \supseteq \{* \}$

Ostatecznie:

$\text{FOLLOW}_1(E) = \{\$,)\}$
 $\text{FOLLOW}_1(E') = \{\$,)\}$
 $\text{FOLLOW}_1(T) = \{+, \$,)\}$
 $\text{FOLLOW}_1(T') = \{+, \$,)\}$
 $\text{FOLLOW}_1(F) = \{*, +, \$,)\}$

Lewostronna faktoryzacja

Jeżeli produkcje gramatyki mające ten sam symbol po lewej stronie posiadają prawe strony rozpoczynające się od tego samego przedrostka, to można dla nich wykonać przekształcenie zwane lewostronną faktoryzacją. Polega ona na zamianie produkcji postaci:

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_k$$

na produkcje:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_k$$



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Gramatyki LL(1)

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

Nazwa gramatyki: LL(k)

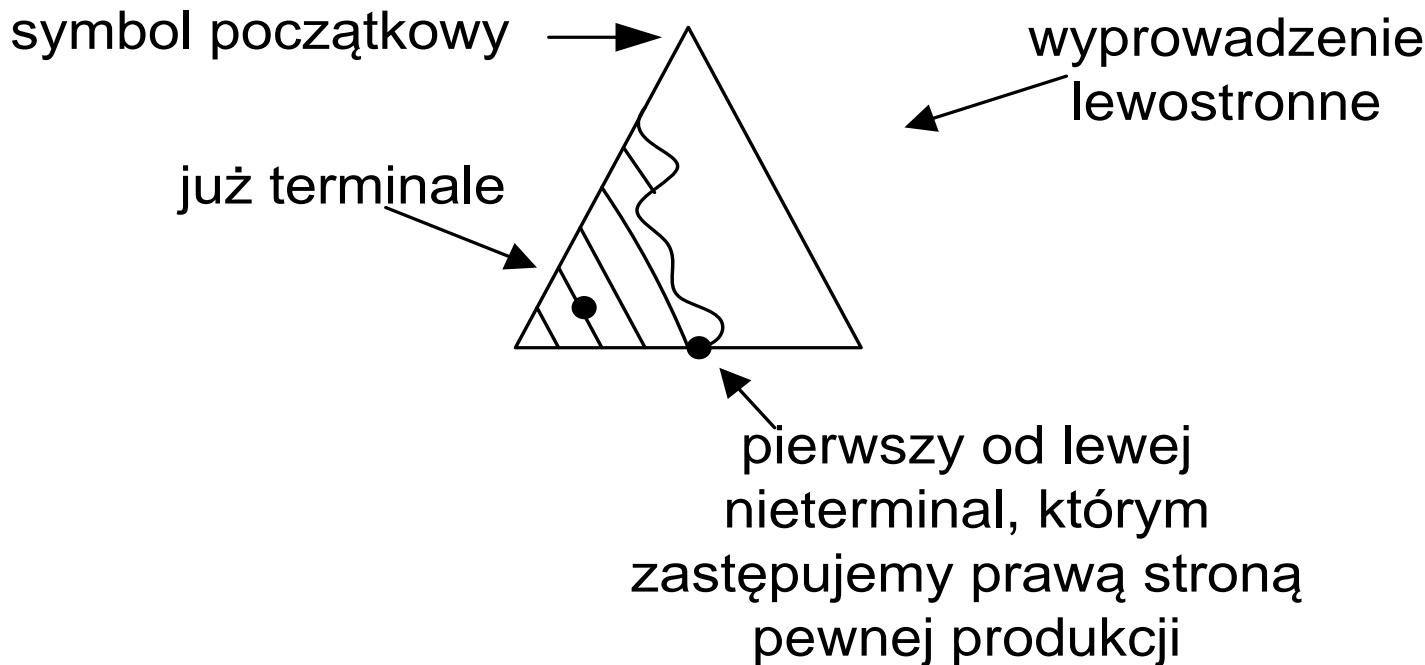
L L (k)

Przeglądanie
wejścia od
lewej strony
do prawej

Odtwarzanie
wywodu
lewostronnego

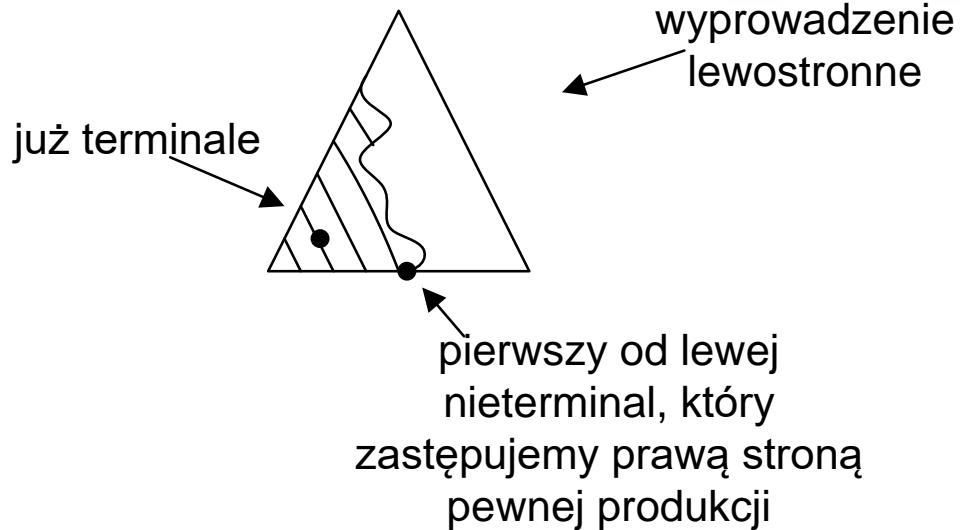
Wystarcza znajomość "k"
następnych symboli
łańcucha wejściowego
oraz skutków poprzednich
kroków, aby wyznaczyć
jednoznacznie produkcję,
która należy zastosować
przy budowie drzewa
wyprowadzenia

Zadanie analizy generacyjnej (zstępującej, top-down)



Odtworzenie wywodu lewostronnego metodą top-down

Istota wywodu top-down



Niech $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

$v = a_1 \dots a_n$ – analizowane słowo

Założymy, że $v \in L(G)$.

Wówczas:

$$S = u_0 \xrightarrow{p_1} u_1 \xrightarrow{p_2} u_2 \xrightarrow{p_m} \dots \xrightarrow{p_m} u_m = v$$

p_i – numery produkcji

$p_1 \dots p_m$ – rozkład lewostronny słowa v

Poszukujemy tego rozkładu!

Istota definicji gramatyki LL(1)

Przypuśćmy, że: $u_i = a_1 \dots a_j A \alpha$

(wyprowadzono już poprawnie pierwsze symbole)

Szukamy: u_{i+1} i p_{i+1} takich, że:

$$u_i \Rightarrow_L^{p_{i+1}} u_{i+1}$$

Znamy:

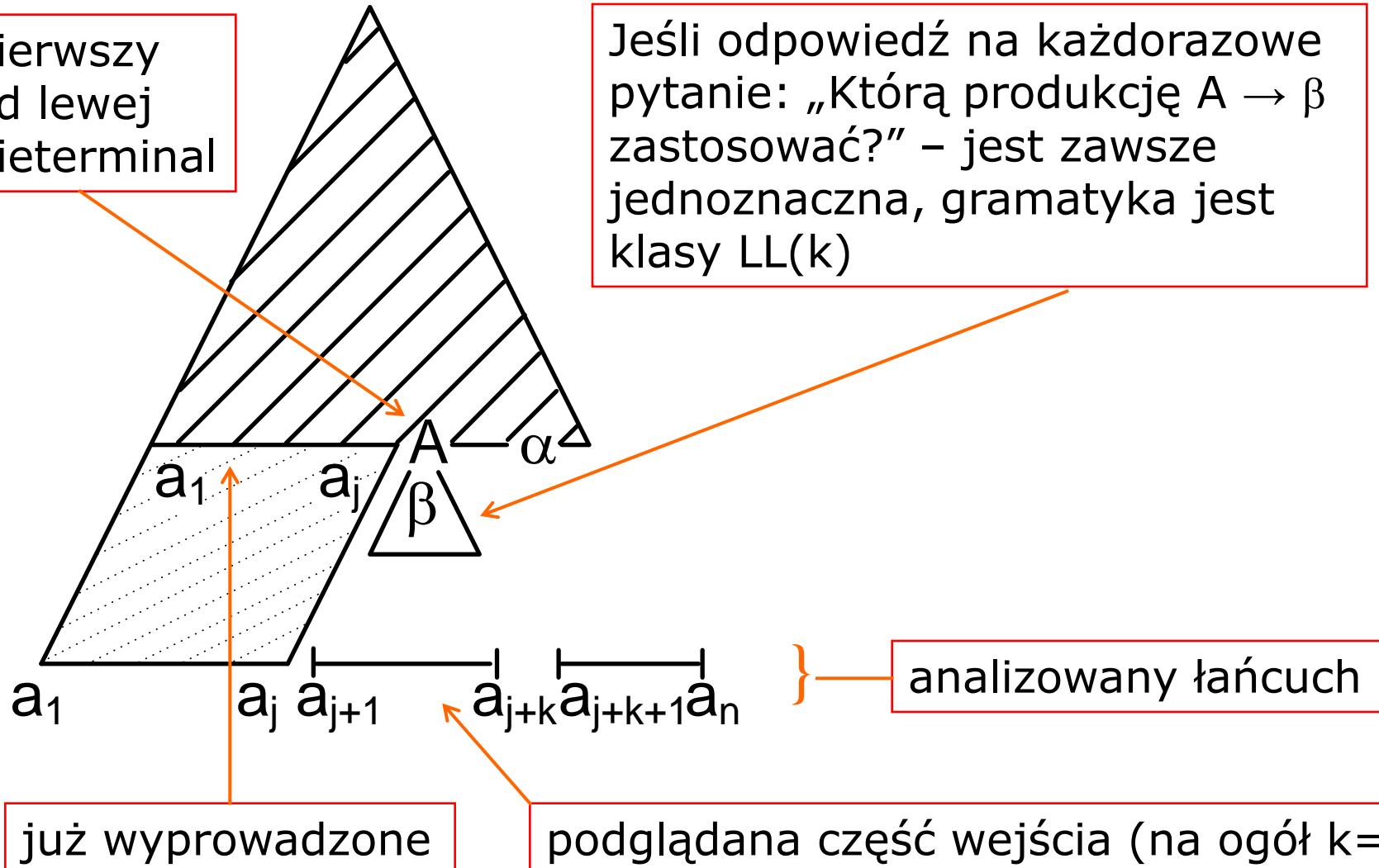
- (1) $a_1 \dots a_j$ – wyprowadzona, początkowa część słowa
- (2) $a_{j+1} \dots a_{j+k}$ – następnych k symboli słowa wejściowego
- (3) nieterminal A

Pytamy:

która produkcję $A \rightarrow \beta$ należy zastosować w wyprowadzeniu. Jeśli odpowiedź na to pytanie jest jednoznaczna ($\forall i$) to gramatyka jest klasy LL(k)

Istota gramatyki i parsera LL(k)

pierwszy
od lewej
nieterminal



Jeśli odpowiedź na każdorazowe pytanie: „Która produkcję $A \rightarrow \beta$ zastosować?” – jest zawsze jednoznaczna, gramatyka jest klasy LL(k)

już wyprowadzone

podglądana część wejścia (na ogólny $k=1$)

Definicja gramatyki LL(1)

Niech:

$$\omega, x, y \in \Sigma^*$$

$$\beta, \gamma \in (V \cup \Sigma)^*$$

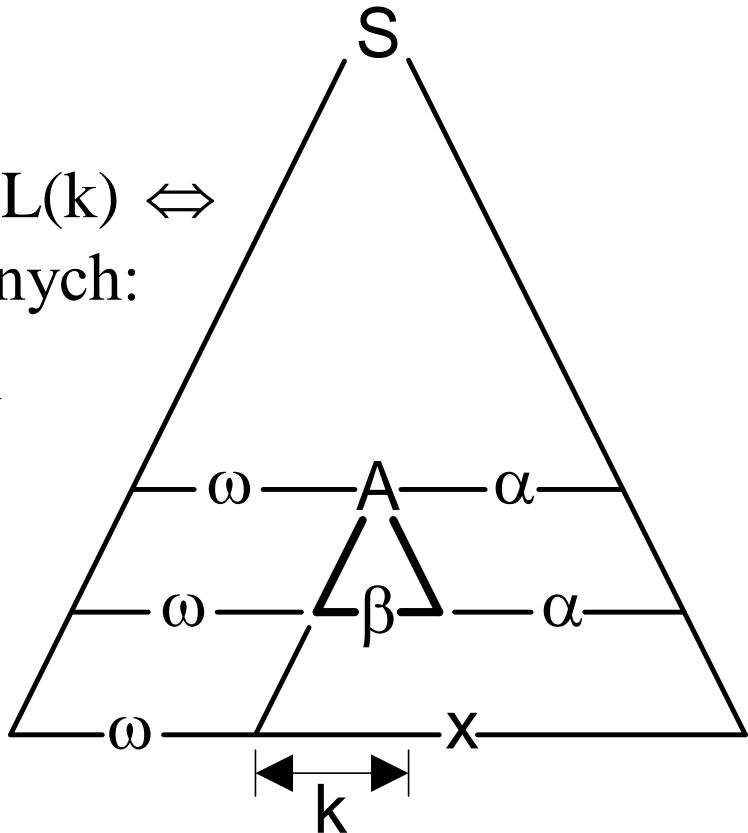
$$A \in V$$

$G = < V, \Sigma, P, S >$ jest klasy LL(k) \Leftrightarrow
gdy dla każdej pary wywodów lewostronnych:

$$(1) \quad S \xrightarrow{L}^* \omega A \alpha \xrightarrow{L} \omega \beta \alpha \xrightarrow{L}^* \omega x$$

$$(2) \quad S \xrightarrow{L}^* \omega A \alpha \xrightarrow{L} \omega \gamma \alpha \xrightarrow{L}^* \omega y$$

jeżeli $\text{FIRST}_k(x) = \text{FIRST}_k(y)$ to $\beta = \gamma$



Przykład

$$G = \langle \{ S, D \}, \{ a, b \}, P, S \rangle$$

- (1) $S \rightarrow a D S$
- (2) $S \rightarrow b$
- (3) $D \rightarrow a$
- (4) $D \rightarrow b S D$

Pytanie: czy G jest klasy LL(1) ?

$$\begin{array}{ll}
 S \Rightarrow_L^* a D S & \left\{ \begin{array}{ll}
 \Rightarrow_L^{(3)} a a S & \Rightarrow_L^* a \boxed{a} b \\
 \omega \beta \alpha & \omega \underline{x} \\
 \Rightarrow_L^{(3)} a a S & \Rightarrow_L^* a \boxed{a} a a b \\
 \omega \gamma \alpha & \omega \underline{y} \\
 \Rightarrow_L^{(4)} a \underline{b S D} S & \Rightarrow_L^* a \boxed{b} b a b \\
 \omega \beta \alpha & \omega \underline{x} \\
 \Rightarrow_L^{(4)} a \underline{b S D} S & \Rightarrow_L^* a \boxed{b} a a b a b \\
 \omega \gamma \alpha & \omega \underline{y}
 \end{array} \right. \\
 s \quad \omega A \alpha &
 \end{array}$$

Jeżeli FIRST(x) = FIRST(y)
= a to produkcja (3)

Jeżeli FIRST(x) = FIRST(y)
= b to produkcja (4)

Takie rozumowanie można uogólnić na wszystkie możliwe wywody

$$S \Rightarrow_L^* \omega S \alpha \Rightarrow \dots \quad \text{oraz} \quad S \Rightarrow_L^* \omega D \alpha \Rightarrow \dots$$

Więc gramatyka jest LL(1) !

Przykład

$G = \langle \{ S, D, E \}, \{ 0, 1, a, b \}, P, S \rangle$

- (1) $S \rightarrow D$
- (2) $S \rightarrow E$
- (3) $D \rightarrow a D b$
- (4) $D \rightarrow 0$
- (5) $E \rightarrow a E b b$
- (6) $E \rightarrow 1$

$$S \xrightarrow[L]{\omega A \alpha} S \left\{ \begin{array}{l} \xrightarrow[L]{(1)} D \xrightarrow[L]{\omega \beta \alpha} \boxed{a^k} 0 b^k \\ \xrightarrow[L]{(2)} E \xrightarrow[L]{\omega \gamma \alpha} \boxed{a^k} 1 b^{2k} \end{array} \right.$$

$$L(G) = \{ a^n 0 b^n : n \geq 0 \} \cup \{ a^n 1 b^{2n} : n \geq 0 \}$$

$$(\omega = \varepsilon ; \alpha = \varepsilon ;)$$

Dla dowolnego k $\text{FIRST}_k(x) = \text{FIRST}_k(y) = a^k$, ale $\beta \neq \gamma$ ($D \neq E$).

Ponieważ k było dowolne – gramatyka G nie jest LL(k) dla żadnego k !

Twierdzenie 1

Twierdzenie pozwalające stwierdzić, czy gramatyka G jest klasy LL(k) dla danego k

$G \in \mathcal{G}_{BK}$ jest klasy LL(k) \Leftrightarrow

$$\forall (A \rightarrow \beta) \in P, (A \rightarrow \gamma) \in P : \beta \neq \gamma$$

$$\forall \alpha : S \xrightarrow{L^*} \omega A \alpha, \quad \omega \in \Sigma^*$$

zachodzi: $\text{FIRST}_k(\beta\alpha) \cap \text{FIRST}_k(\gamma\alpha) = \emptyset$

Przykład

- (1) $S \rightarrow S a$
- (2) $S \rightarrow b$

Rozważamy: wyprowadzenie produkcje

$$\begin{array}{ccc} S \Rightarrow_L^i S a^i & S \rightarrow S a & S \rightarrow b \\ S \Rightarrow_L^* \omega A \alpha & A \rightarrow \beta & A \rightarrow \gamma \end{array}$$

Dla $k \leq i$ mamy

$$\text{FIRST}_k(\beta \alpha) = \text{FIRST}_k(S a a^i) = \{ba^{k-1}\}$$

$$\text{FIRST}_k(\gamma \alpha) = \text{FIRST}_k(b a^i) = \{ba^{k-1}\}$$

Ponieważ i – dowolne, $k \leq i$, więc G nie może być klasy $LL(k)$ dla żadnego k .

Uwaga: żadna gramatyka z lewostronną rekursją nie jest klasy $LL(k)$ dla żadnego k .

Twierdzenie 2

$G \in \mathcal{G}_{BK}$ jest klasy LL(1) \Leftrightarrow

$\forall (A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n) \in P$ zachodzi:

- 1) $\text{FIRST}_1(\alpha_1), \text{FIRST}_1(\alpha_2), \dots, \text{FIRST}_1(\alpha_n)$ są parami rozłączne, oraz
- 2) jeżeli $\alpha_i \Rightarrow^* \varepsilon$ to
 $\text{FIRST}_1(\alpha_j) \cap \text{FOLLOW}_1(A) = \emptyset$ (pusty)
dla $1 \leq j \leq n \wedge i \neq j$

Przykład

Poprzednia gramatyka

$$S \rightarrow S a \mid b$$

po przekształceniu ma postać:

$$\begin{array}{ll} S \rightarrow b S' & S' \rightarrow a S' \mid \varepsilon \\ A & \alpha_1 \quad \alpha_2 \end{array}$$

i jest klasy LL(1) bo:

$$(1) \quad \text{FIRST}_1(\alpha_1) = \text{FIRST}_1(aS') = \{a\}$$

$$\text{FIRST}_1(\alpha_2) = \text{FIRST}_1(\varepsilon) = \{\varepsilon\}$$

$$\text{FIRST}_1(\alpha_1) \cap \text{FIRST}_1(\alpha_2) = \emptyset$$

$$(2) \quad \varepsilon \xrightarrow{*} \varepsilon$$

$$\text{FIRST}_1(\alpha_1) = \text{FIRST}_1(aS') = \{a\}$$

$$\text{FOLLOW}_1(A) = \text{FOLLOW}_1(S') = \{\$\}$$

$$\text{FIRST}_1(\alpha_1) \cap \text{FOLLOW}_1(A) = \text{FIRST}_1(aS') \cap \text{FOLLOW}_1(S') = \emptyset$$

Twierdzenie 3

- (1) $G \in G_{BK}$ jest klasy LL(k) $\Rightarrow \exists G' \in G_{BK} :$
- (a) G' - jest klasy LL($k+1$)
 - (b) G' - nie zawiera ε -produkacji
 - (c) $L(G') = L(G)$
- (2) $G \in G_{BK}$ jest klasy LL(k), gdzie $k \geq 2$,
oraz G nie zawiera ε -produkacji $\Rightarrow \exists G' \in G_{BK} :$
- (a) G' - jest klasy LL($k-1$)
 - (b) $L(G') = L(G)$

Lewostronna faktoryzacja

Przekształceniem obniżającym stopień gramatyki LL kosztem wprowadzenia ε -produkci jest lewostronna faktoryzacja. Polega ona na zamianie produkcji postaci:

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_k$$

na produkcje:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_k$$

Przykład

Gramatyka:

$$S \rightarrow a S \mid a$$

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$$

jest gramatyką klasy LL(2) bez ε -produkcji.

Po lewostronnej faktoryzacji

$$S \rightarrow a S'$$

$$S' \rightarrow S \mid \varepsilon$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

jest klasy LL(1), ale zawiera ε -produkce



AGH

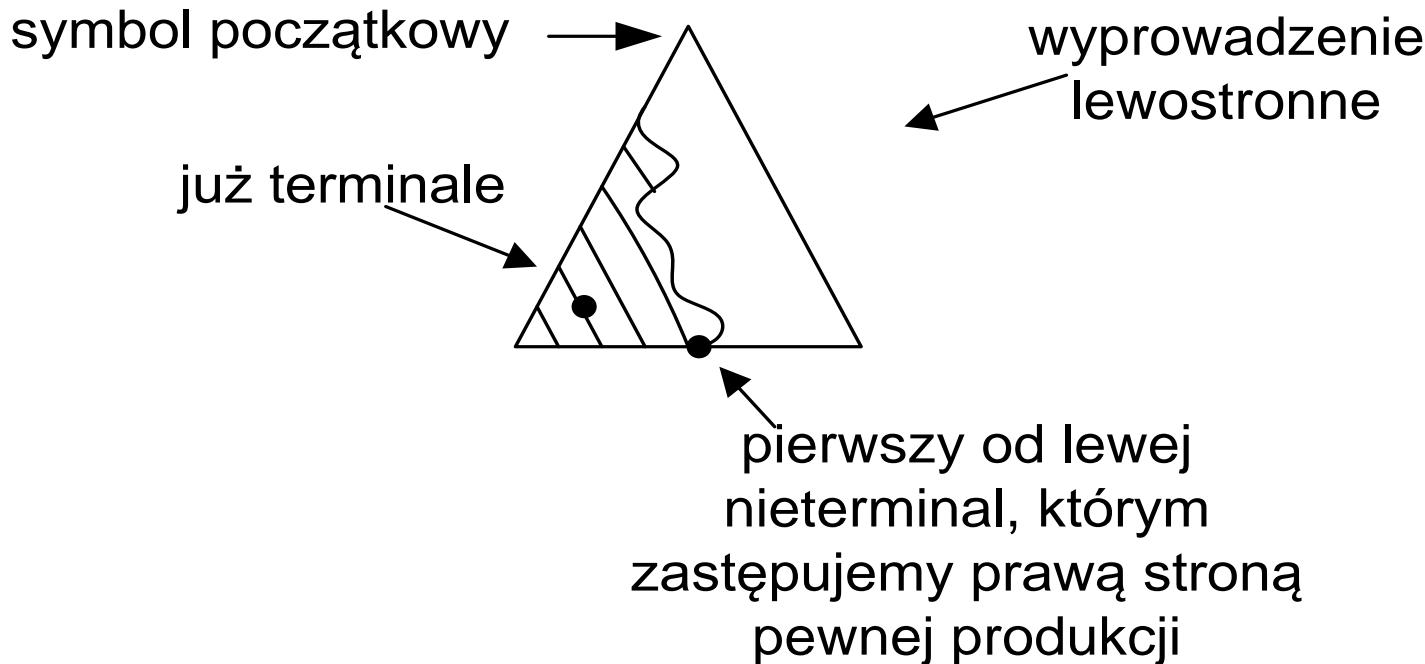
AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Parsery LL(1)

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Zadanie analizy generacyjnej (zstępującej, top-down)

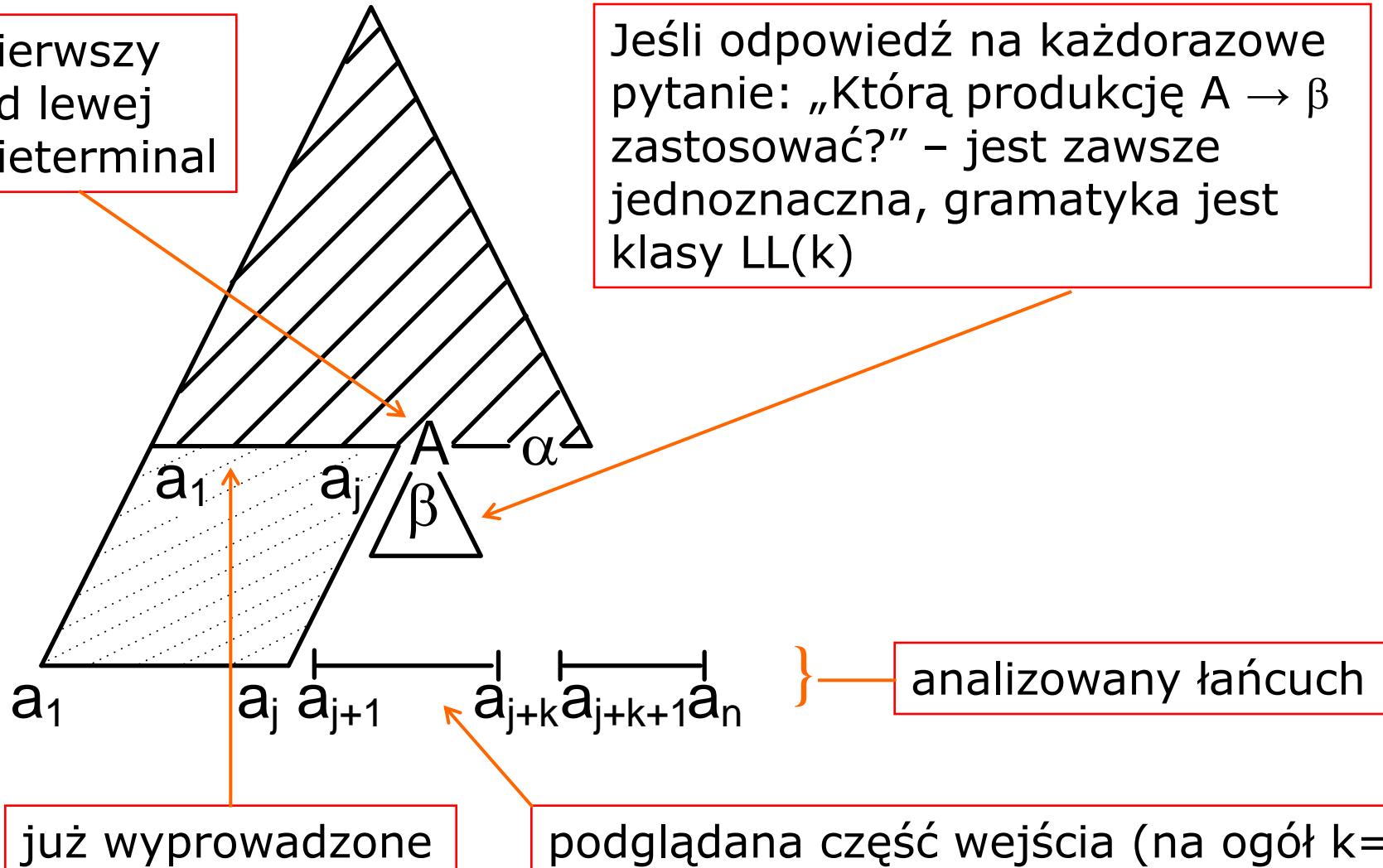


Odtworzenie wywodu lewostronnego metodą top-down

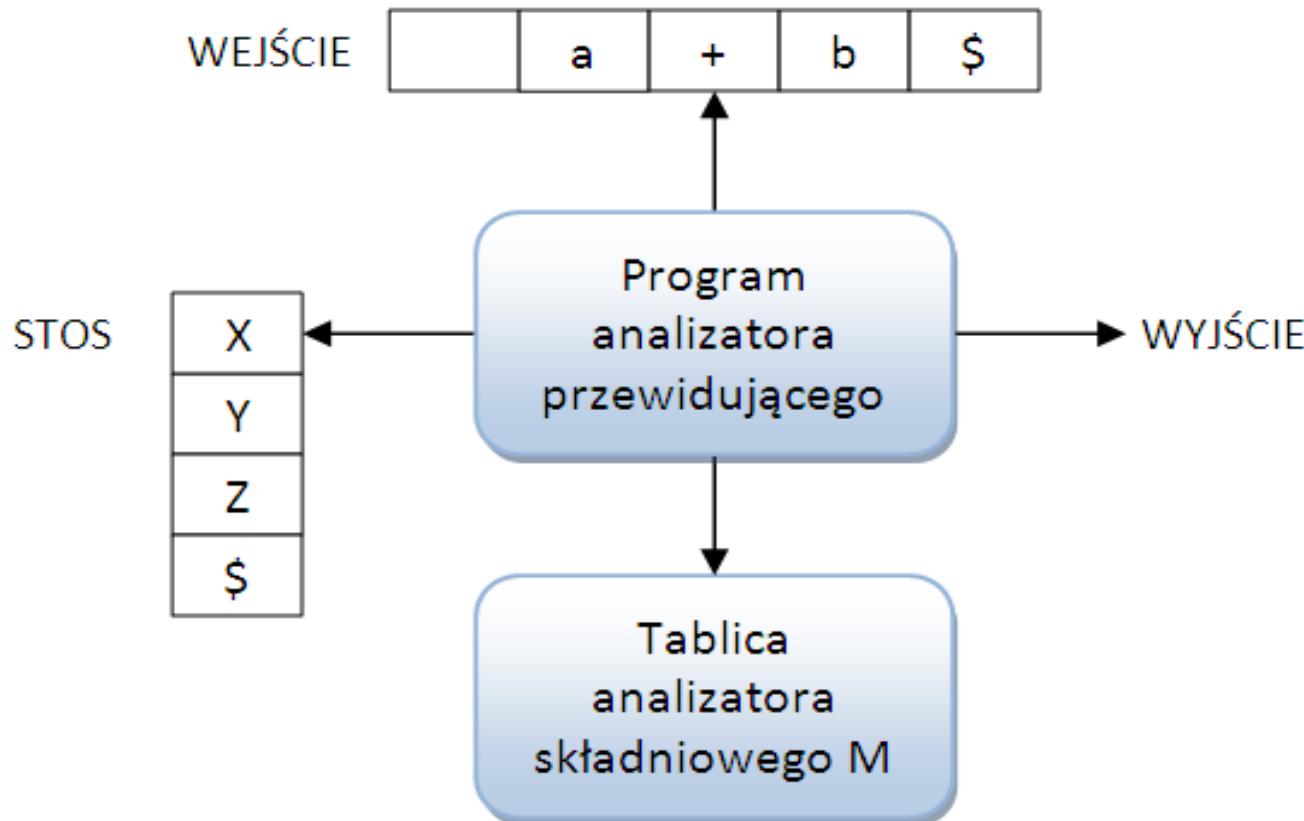
Istota gramatyki i parsera LL(k)

pierwszy
od lewej
nieterminal

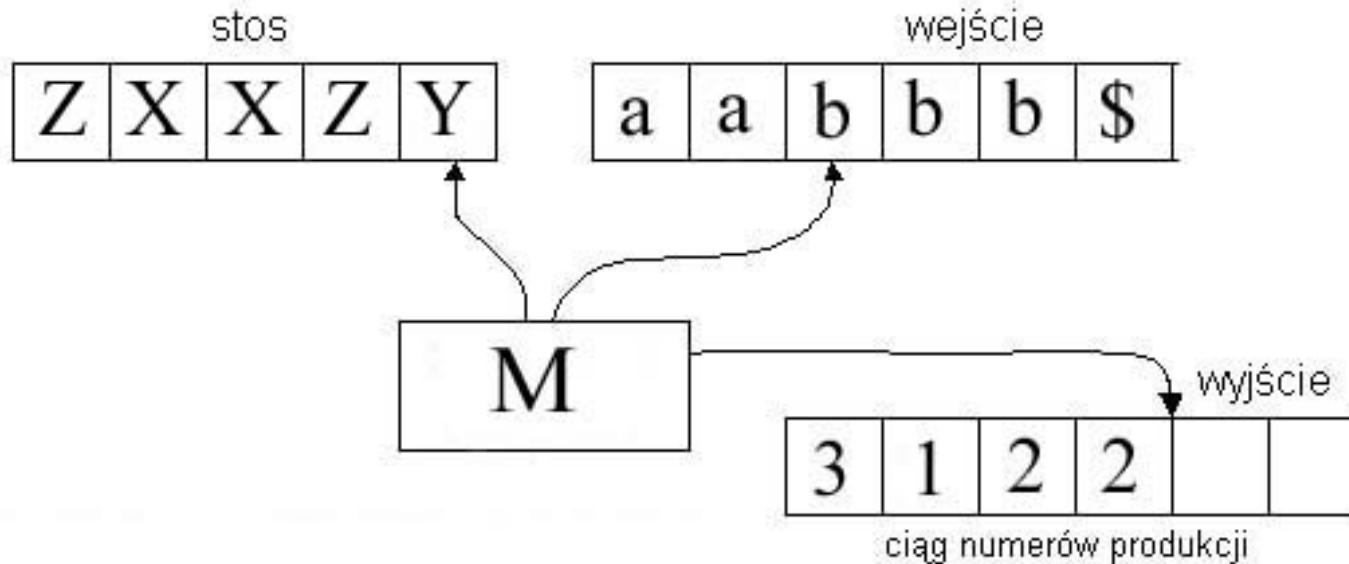
Jeśli odpowiedź na każdorazowe pytanie: „Która produkcję $A \rightarrow \beta$ zastosować?” – jest zawsze jednoznaczna, gramatyka jest klasy LL(k)



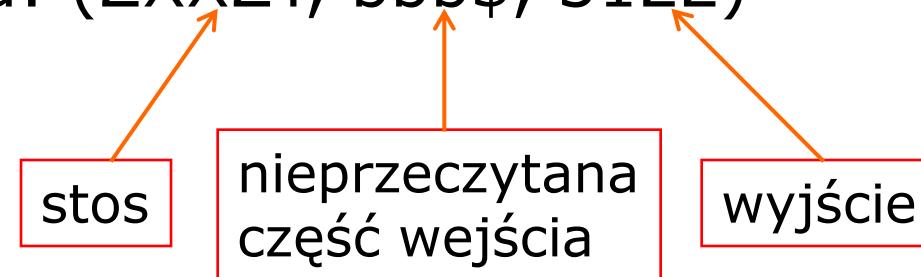
Nierekurencyjny analizator top-down



Konfiguracja parsera LL



Konfiguracja: (ZXXZY, bbb\$, 3122)



Parser LL(k)

$G = \langle V, \Sigma, P, S \rangle$ – gramatyka opisująca składnię

$A = \langle \Sigma, \Gamma, Z_0, \Delta, M, \$ \rangle$

Σ – zbiór symboli terminalnych

Γ – zbiór symboli stosowych

$$\Gamma = V \cup \Sigma$$

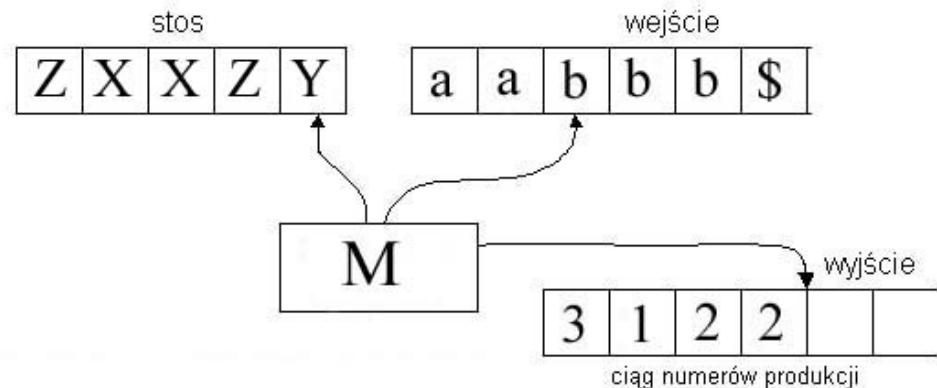
$Z_0 = S$ – symbol początkowy stosu oraz gramatyki

Δ – alfabet wyjściowy: zbiór numerów produkcji

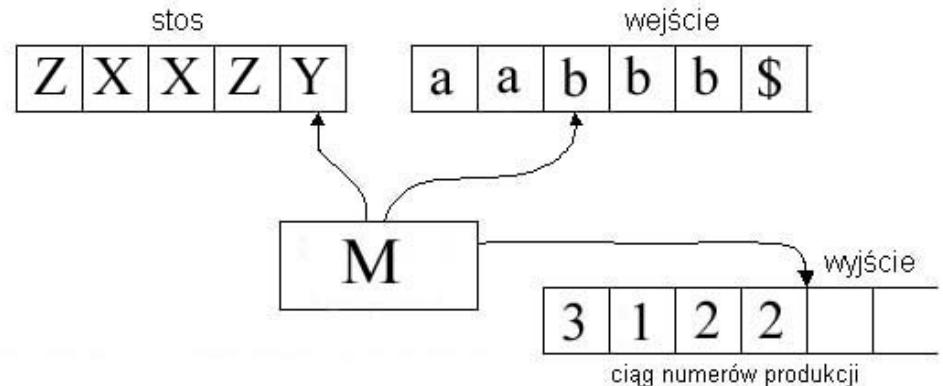
M – tablica sterująca parsera

$$M : V \times (\Sigma \cup \{\$\})^* \rightarrow \begin{cases} (V \cup \Sigma)^* \times \Delta \\ \underline{ERROR} \end{cases}$$

$\$$ - ogranicznik końca słowa wejściowego



Parser LL(k)



Konfiguracja parsera: $(\alpha Y, x, \pi)$

αY - zawartość stosu

Y - wierzchołek stosu

x - nieprzeczytana część wejścia

π - łańcuch numerów produkcji na wyjściu

Parser podgląda na wejściu $u = FIRST_K(x)$

Działanie parsera LL(k)

- Automat widzi wierzchołek stosu i k symboli wejściowych.
- Tablicę M wykorzystuje się, gdy na wierzchołku stosu znajduje się nieterminal.
- Konfiguracja początkowa: $(S, \omega\$, \varepsilon)$, przy czym:
 - ω – analizowane słowo;
 - S – symbol początkowy gramatyki

Działanie parsera LL(k)

Konfiguracja parsera: $(\alpha Y, x, \pi)$ zaś $u = FIRST_k(x)$

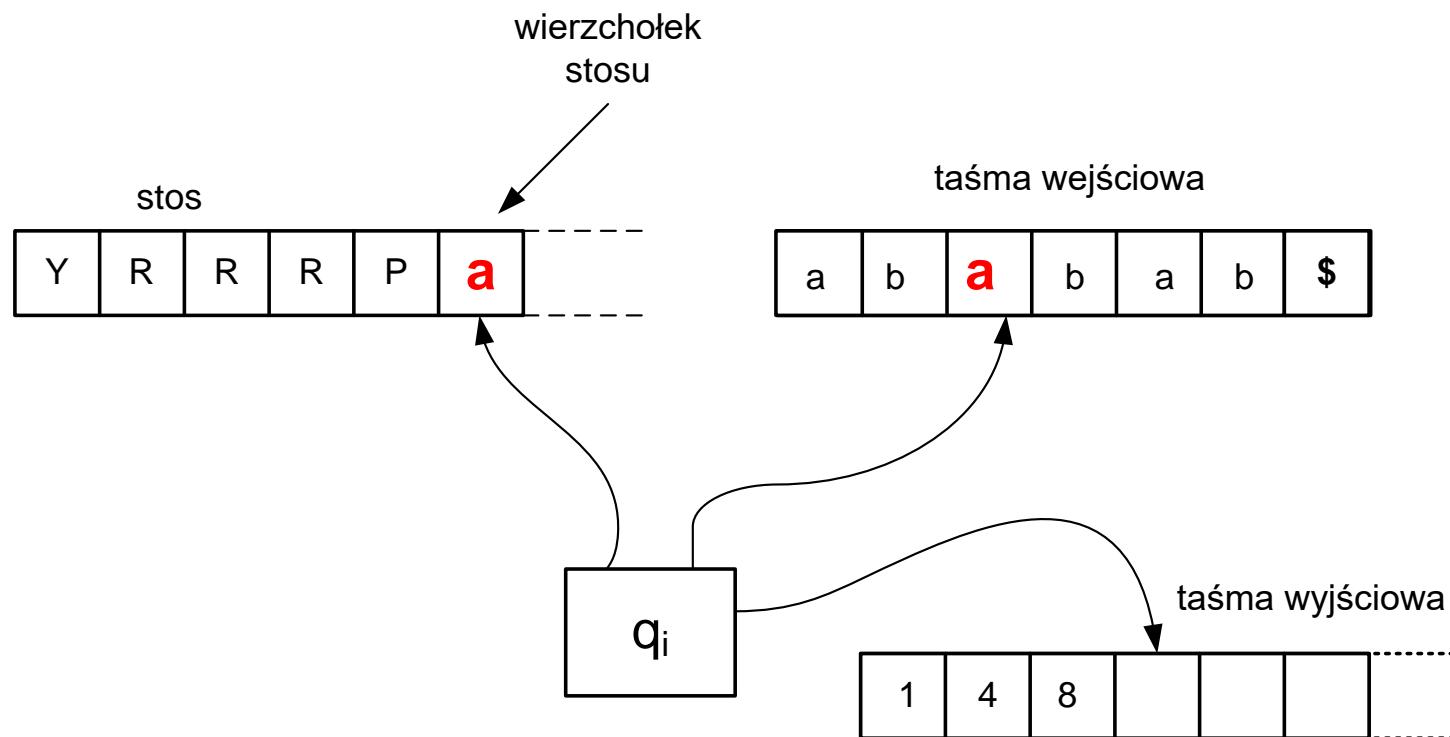
- (1) if $Y \in \Sigma$ then
 if $Y = a$ then /*pop*/
 $(\alpha a, a\gamma, \pi) \xrightarrow{} (\alpha, \gamma, \pi)$
 else ERROR;
- (2) if $Y \in V$ then
 if $M(Y, u) = (\beta, i)$ then /* wyprowadzenie */
 $(\alpha Y, x, \pi) \xrightarrow{} (\alpha\beta^R, x, \pi i)$
 /* $Y \rightarrow \beta$ jest produkcją o numerze i */
 else ERROR;
- (3) if $(Y = \epsilon)$ then
 if $x = \$$ then ACCEPT
 else ERROR;

Działanie parsera LL(k)

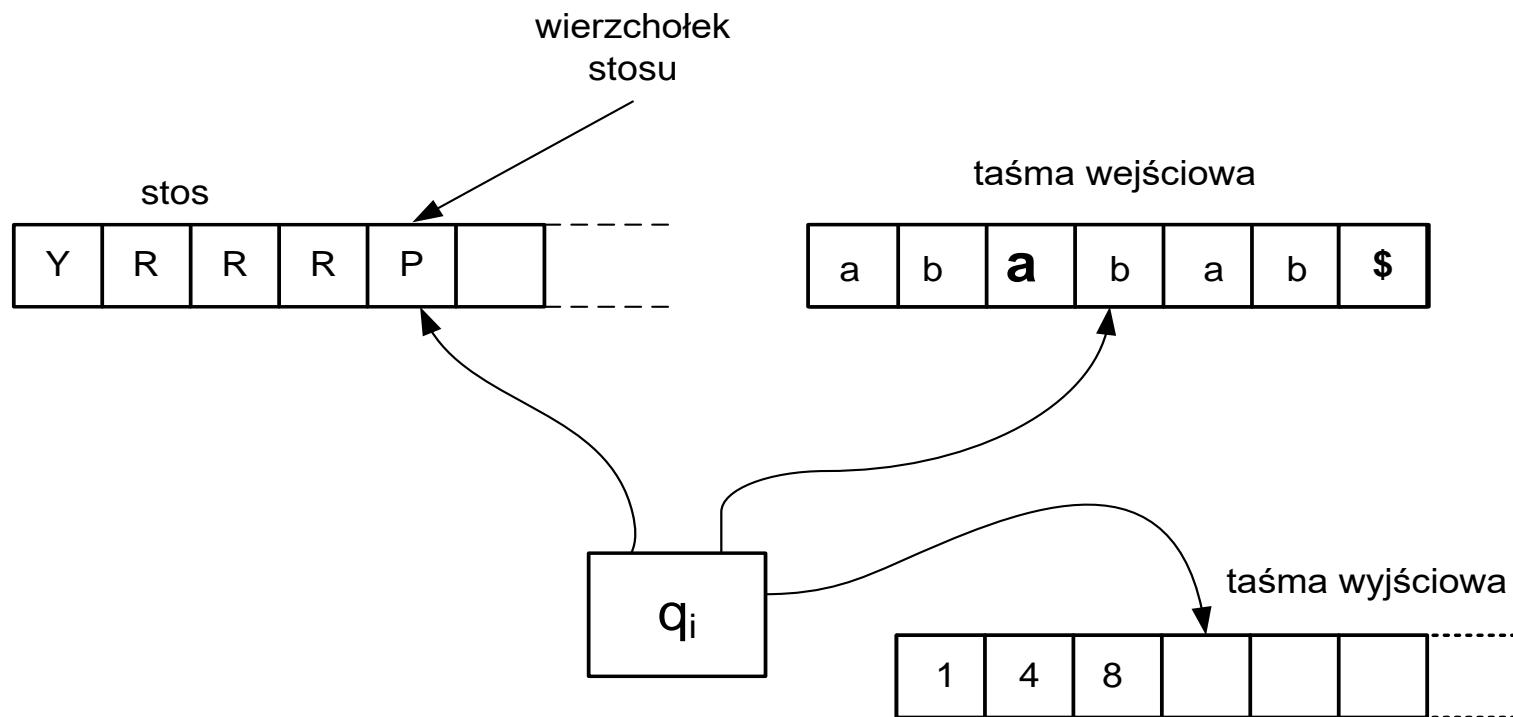
Konfiguracja końcowa akceptująca: $(\varepsilon, \$, \pi)$

π - ciąg numerów produkcji opisujący rozkład lewostronny
słowa $\omega \in T^*$ w gramatyce G. Wówczas $\omega \in L(G)$

Krok „pop”

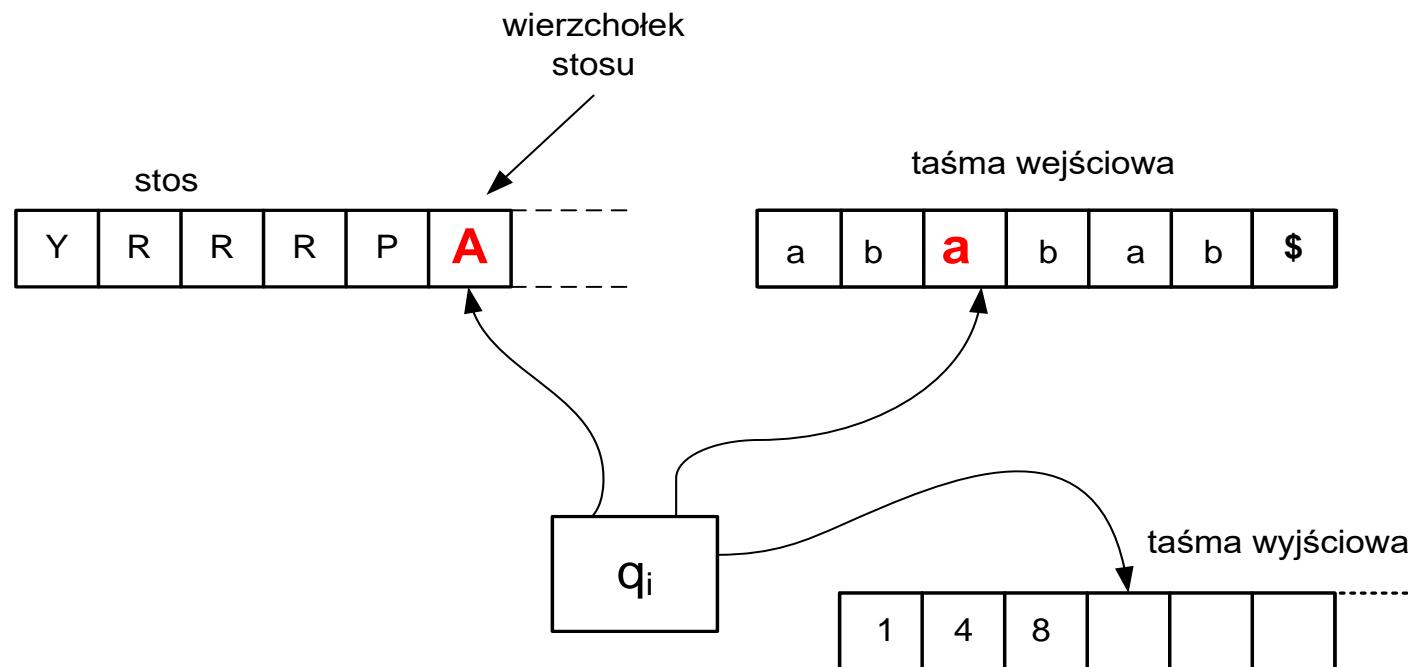


Krok „pop”



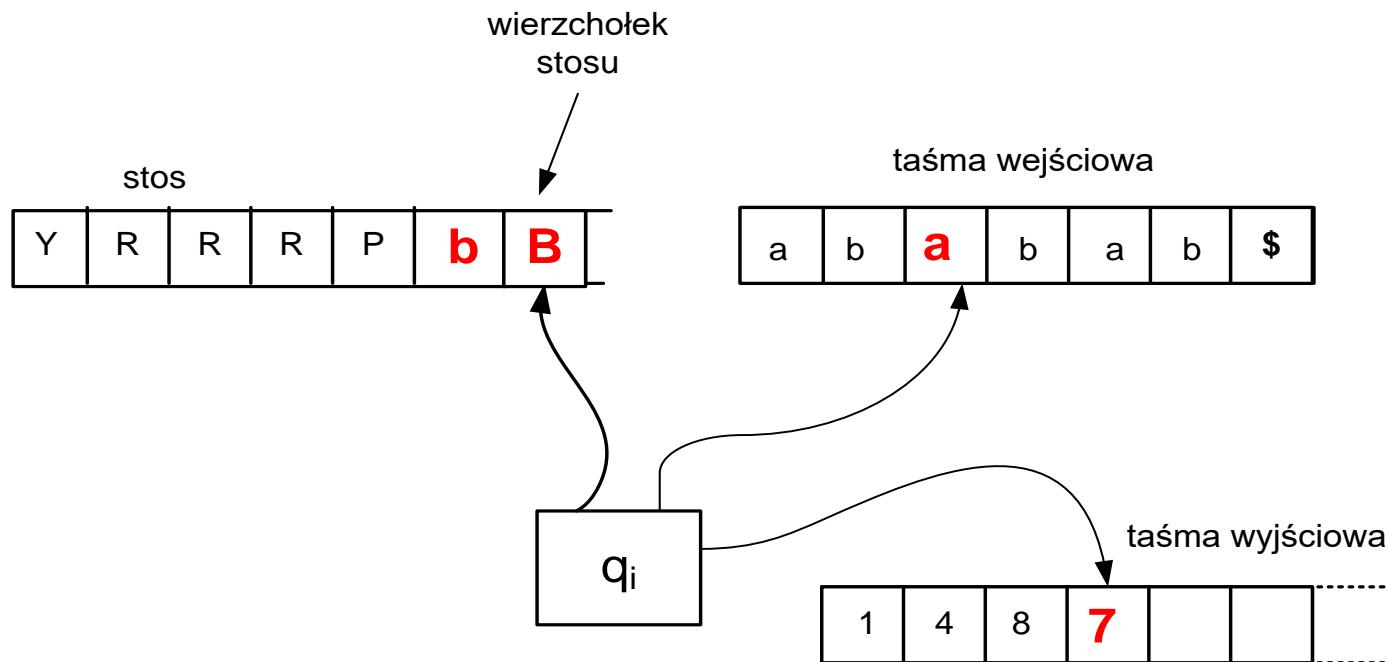
Krok „wyprowadzenia”

A→Bb



Krok „wyprowadzenia”

A→Bb



Konstrukcja tablicy sterującej M

Konstrukcja tablicy M dla parsera LL(1)

We: $G = \langle N, T, P, Z \rangle \in G_{BK}$

G – jest klasy LL(1)

Wy: tablica $M : V \times (\Sigma \cup \{\$\}) \rightarrow \begin{cases} (V \cup \Sigma)^* \times \Delta \\ \underline{ERROR} \end{cases}$

gdzie: Δ - zbiór numerów produkcji w gramatyce G

Konstrukcja tablicy sterującej M

for $(A \rightarrow \beta) \in P$ and $A \rightarrow \beta$ - produkcja numer i do

begin

(1) for każde $a \in FIRST_1(\beta) : a \neq \varepsilon$ do

$M(A, a) := (\beta, i);$

(2) if $\varepsilon \in FIRST_1(\beta)$ then

for każde $b \in FOLLOW_1(A)$ do

$M(A, b) := (\beta, i);$

end;

for każde niezdefiniowane $M(A, a)$ do

$M(A, a) := \underline{ERROR};$

Projektowanie tablicy parsera LL

Mając przekształconą gramatykę języka oraz wyznaczone (dla symboli nieterminalnych tej gramatyki) zbiory FIRST_1 i FOLLOW_1 , możemy przystąpić do projektowania tablicy parsera LL.

- 1) $E \rightarrow TE'$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \varepsilon$
- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \varepsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow \underline{id}$

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)						
E'	+ , ε	\$,)						
T	(, <u>id</u>	+ , \$,)						1) $E \rightarrow TE'$ 2) $E' \rightarrow +TE'$ 3) $E' \rightarrow \epsilon$ 4) $T \rightarrow FT'$ 5) $T' \rightarrow *FT'$ 6) $T' \rightarrow \epsilon$ 7) $F \rightarrow (E)$ 8) $F \rightarrow \underline{id}$
T'	* , ε	+ , \$,)						
F	(, <u>id</u>	* , + , \$,)						

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+ , ε	\$,)						
T	(, <u>id</u>	+ , \$,)						
T'	* , ε	+ , \$,)						
F	(, <u>id</u>	* , + , \$,)						

- 1) $E \rightarrow TE'$
 2) $E' \rightarrow +TE'$
 3) $E' \rightarrow \varepsilon$
 4) $T \rightarrow FT'$
 5) $T' \rightarrow *FT'$
 6) $T' \rightarrow \varepsilon$
 7) $F \rightarrow (E)$
 8) $F \rightarrow \underline{id}$

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+ , ϵ	\$,)						
T	(, <u>id</u>	+ , \$,)	$T \rightarrow FT'$ (4)			$T \rightarrow FT'$ (4)		
T'	* , ϵ	+ , \$,)						
F	(, <u>id</u>	* , + , \$,)						

1) $E \rightarrow TE'$
 2) $E' \rightarrow + TE'$
 3) $E' \rightarrow \epsilon$
 4) $T \rightarrow FT'$
 5) $T' \rightarrow *FT'$
 6) $T' \rightarrow \epsilon$
 7) $F \rightarrow (E)$
 8) $F \rightarrow id$

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+ , ϵ	\$,)						
T	(, <u>id</u>	+ , \$,)	$T \rightarrow FT'$ (4)			$T \rightarrow FT'$ (4)	1) $E \rightarrow TE'$ 2) $E' \rightarrow +TE'$ 3) $E' \rightarrow \epsilon$ 4) $T \rightarrow FT'$ 5) $T' \rightarrow *FT'$ 6) $T' \rightarrow \epsilon$ 7) $F \rightarrow (E)$ 8) $F \rightarrow \underline{id}$	
T'	* , ϵ	+ , \$,)						
F	(, <u>id</u>	* , + , \$,)	$F \rightarrow \underline{id}$ (8)			$F \rightarrow (E)$ (7)		

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+, ϵ	\$,)						
T	(, <u>id</u>	+, \$,)	$T \rightarrow FT'$ (4)			$T \rightarrow FT'$ (4)		
T'	*, ϵ	+, \$,)			$T' \rightarrow *FT'$ (5)			
F	(, <u>id</u>	*, +, \$,)	$F \rightarrow id$ (8)			$F \rightarrow (E)$ (7)		

- 1) $E \rightarrow TE'$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow id$

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+ , ϵ	\$,)						
T	(, <u>id</u>	+ , \$,)	$T \rightarrow FT'$ (4)			$T \rightarrow FT'$ (4)		
T'	* , ϵ	+ , \$,)		$T' \rightarrow \epsilon$ (6)	$T' \rightarrow *FT'$ (5)		$T' \rightarrow \epsilon$ (6)	$T' \rightarrow \epsilon$ (6)
F	(, <u>id</u>	* , + , \$,)	$F \rightarrow id$ (8)			$F \rightarrow (E)$ (7)		

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)		1) $E \rightarrow TE'$ 2) $E' \rightarrow +TE'$ 3) $E' \rightarrow \epsilon$ 4) $T \rightarrow FT'$ 5) $T' \rightarrow *FT'$ 6) $T' \rightarrow \epsilon$ 7) $F \rightarrow (E)$ 8) $F \rightarrow id$			
E'	+, ϵ	\$,)		$E' \rightarrow +TE'$ (2)		$E' \rightarrow \epsilon$ (3)	$E' \rightarrow \epsilon$ (3)	
T	(, <u>id</u>	+, \$,)	$T \rightarrow FT'$ (4)					
T'	\ast , ϵ	+, \$,)		$T' \rightarrow \epsilon$ (6)	$T' \rightarrow *FT'$ (5)	$T' \rightarrow \epsilon$ (6)	$T' \rightarrow \epsilon$ (6)	
F	(, <u>id</u>	\ast , +, \$,)	$F \rightarrow id$ (8)			$F \rightarrow (E)$ (7)		

Tablica parsera

	FIRST	FOLLOW	<u>id</u>	+	*	()	\$
E	(, <u>id</u>	\$,)	$E \rightarrow TE'$ (1)			$E \rightarrow TE'$ (1)		
E'	+ , ϵ	\$,)		$E' \rightarrow +TE'$ (2)			$E' \rightarrow \epsilon$ (3)	$E' \rightarrow \epsilon$ (3)
T	(, <u>id</u>	+ , \$,)	$T \rightarrow FT'$ (4)			$T \rightarrow FT'$ (4)		
T'	* , ϵ	+ , \$,)		$T' \rightarrow \epsilon$ (6)	$T' \rightarrow *FT'$ (5)		$T' \rightarrow \epsilon$ (6)	$T' \rightarrow \epsilon$ (6)
F	(, <u>id</u>	* , + , \$,)	$F \rightarrow id$ (8)			$F \rightarrow (E)$ (7)		

Symulacja działania parsera LL

Stos

E
E'T
E'T'F
E'T'id
E'T'
E'
E'T+
E'T
E'T'F
E'T'id
E'T'
E'
 ϵ

Wejście

|id+id\$
|id+id\$
|id+id\$
|id+id\$
+id\$
+id\$
+id\$
id\$
id\$
id\$
\$
\$
\$

Wyjście

ϵ
1
14
148
148
1486
14862
14862
148624
1486248
14862486
148624863

akceptacja

Metoda zejść rekurencyjnych

- (1) Dla każdego symbolu nieterminalnego gramatyki budujemy procedurę (funkcję) rekurencyjną. Z programu nadziednego (głównego) wywoływana jest procedura (funkcja) dla symbolu początkowego gramatyki.
- (2) Wewnątrz procedury (funkcji) dokonuje się wyboru produkcji na podstawie tokenu podglądanego na wejściu (lookahead). Po dokonaniu wyboru zapewnia się zanotowanie numeru wybranej produkcji (raport). Następnie każdy symbol nieterminalny prawej strony produkcji przekształca się w wywołanie odpowiadającej mu procedury (funkcji), każdy symbol terminalny prowadzi do wywołania operacji „match”, która sprawdza obecność tego symbolu na wejściu. W przypadku zgodności czyta następny token, w przeciwnym razie sygnalizuje błąd. Symbole z prawej strony produkcji znajdują swoje odpowiedniki w procedurze w tej kolejności, w jakiej są one umieszczone w prawej stronie produkcji. Niemożność dokonania wyboru produkcji wewnątrz procedury sygnalizowana jest błędem.

Metoda zejść rekurencyjnych

	<u>id</u>	+	*	()	\$
E	(1) $E \rightarrow TE_1$			(1) $E \rightarrow TE_1$		
E_1		(2) $E_1 \rightarrow +TE_1$			(3) $E_1 \rightarrow \varepsilon$	(3) $E_1 \rightarrow \varepsilon$
T	(4) $T \rightarrow FT_1$			(4) $T \rightarrow FT_1$		
T_1		(6) $T_1 \rightarrow \varepsilon$	(5) $T_1 \rightarrow *FT_1$		(6) $T_1 \rightarrow \varepsilon$	(6) $T_1 \rightarrow \varepsilon$
F	(8) $F \rightarrow \underline{id}$			(7) $F \rightarrow (E)$		

```

E()
{
    if (lookahead == ID || lookahead == '(')
    {
        raport(1); T(); E1();
    }
    else error();
}

```

Metoda zejść rekurencyjnych

	<u>id</u>	+	*	()	\$
E	$(1) E \rightarrow TE_1$			$(1) E \rightarrow TE_1$		
E_1		$(2) E_1 \rightarrow +TE_1$			$(3) E_1 \rightarrow \varepsilon$	$(3) E_1 \rightarrow \varepsilon$
T	$(4) T \rightarrow FT_1$			$(4) T \rightarrow FT_1$		
T_1		$(6) T_1 \rightarrow \varepsilon$	$(5) T_1 \rightarrow *FT_1$		$(6) T_1 \rightarrow \varepsilon$	$(6) T_1 \rightarrow \varepsilon$
F	$(8) F \rightarrow id$			$(7) F \rightarrow (E)$		

```

E1()
{
    if(lookahead == '+')
    {
        rapport(2); match('+'); T(); E1();
    }
    else if (lookahead == ')' || lookahead == '$')
        rapport(3);
    else error();
}

```

Metoda zejść rekurencyjnych

	<u>id</u>	+	*	()	\$
E	(1) $E \rightarrow TE_1$			(1) $E \rightarrow TE_1$		
E_1		(2) $E_1 \rightarrow +TE_1$			(3) $E_1 \rightarrow \varepsilon$	(3) $E_1 \rightarrow \varepsilon$
T	(4) $T \rightarrow FT_1$			(4) $T \rightarrow FT_1$		
T_1		(6) $T_1 \rightarrow \varepsilon$	(5) $T_1 \rightarrow *FT_1$		(6) $T_1 \rightarrow \varepsilon$	(6) $T_1 \rightarrow \varepsilon$
F	(8) $F \rightarrow \underline{id}$			(7) $F \rightarrow (E)$		

```

T()
{
    if(lookahead == ID || lookahead == '(')
    {
        report(4); F(); T1();
    }
    else error();
}

```

Metoda zejść rekurencyjnych

	<u>id</u>	+	*	()	\$
E	(1) $E \rightarrow TE_1$			(1) $E \rightarrow TE_1$		
E_1		(2) $E_1 \rightarrow +TE_1$			(3) $E_1 \rightarrow \varepsilon$	(3) $E_1 \rightarrow \varepsilon$
T	(4) $T \rightarrow FT_1$			(4) $T \rightarrow FT_1$		
T_1		(6) $T_1 \rightarrow \varepsilon$	(5) $T_1 \rightarrow *FT_1$		(6) $T_1 \rightarrow \varepsilon$	(6) $T_1 \rightarrow \varepsilon$
F	(8) $F \rightarrow id$			(7) $F \rightarrow (E)$		

```

T1 ()
{
    if (lookahead == '*' )
    {
        raport(5); match('*'); F(); T1();
    }
    else if(lookahead == '+' || lookahead == ')' ||
        lookahead == '$') raport(6);
    else error();
}

```

Metoda zejść rekurencyjnych

	<u>id</u>	+	*	()	\$
E	(1) $E \rightarrow TE_1$			(1) $E \rightarrow TE_1$		
E_1		(2) $E_1 \rightarrow +TE_1$			(3) $E_1 \rightarrow \varepsilon$	(3) $E_1 \rightarrow \varepsilon$
T	(4) $T \rightarrow FT_1$			(4) $T \rightarrow FT_1$		
T_1		(6) $T_1 \rightarrow \varepsilon$	(5) $T_1 \rightarrow *FT_1$		(6) $T_1 \rightarrow \varepsilon$	(6) $T_1 \rightarrow \varepsilon$
F	(8) $F \rightarrow id$			(7) $F \rightarrow (E)$		

```

F()
{
    if(lookahead == ID)
    {
        rapport(8); match(ID);
    }
    else if(lookahead == '(')
    {
        rapport(7); match('('); E(); match(')');
    }
    else error();
}

```



AGH

Metoda zejść rekurencyjnych

```
error()
{ ..... }

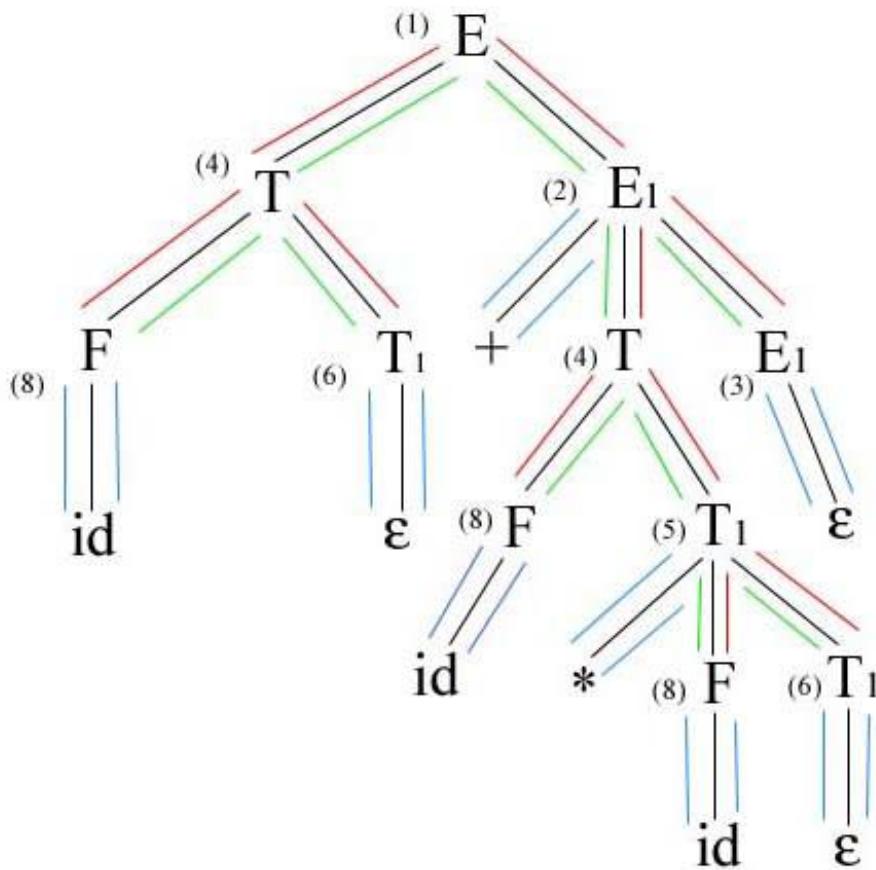
match(t)
int t;
{
    if(lookahead == t) lookahead=gettoken();
    else error();
}

raport()
{ ..... }

gettoken()
{ ..... }

int lookahead;
main()
{
    lookahead=gettoken();
    E();           /*wywołanie procedury dla symbolu
                     początkowego gramatyki*/
}
```

Metoda zejść rekurencyjnych



Wejście:
 $\underline{\text{id}} + \underline{\text{id}} * \underline{\text{id}}$

- wywołania (zejścia) rekurencyjne
- powroty
- działanie procedur nie powodujących zejść rekurencyjnych
- (i) raportowanie numerów stosowanych produkcji



AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Parsery LR(1) – część 1

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Nazwa gramatyki: LR(k)

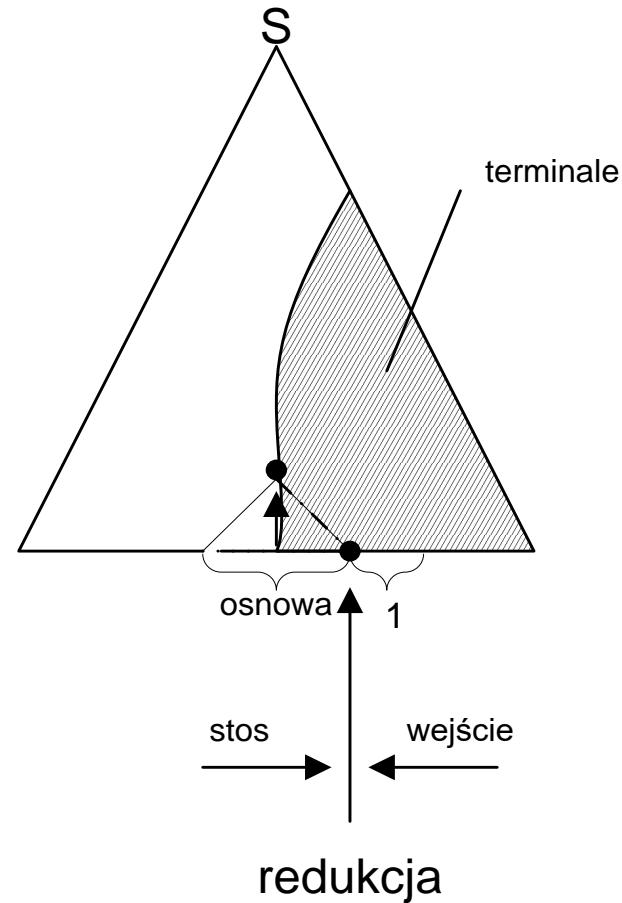
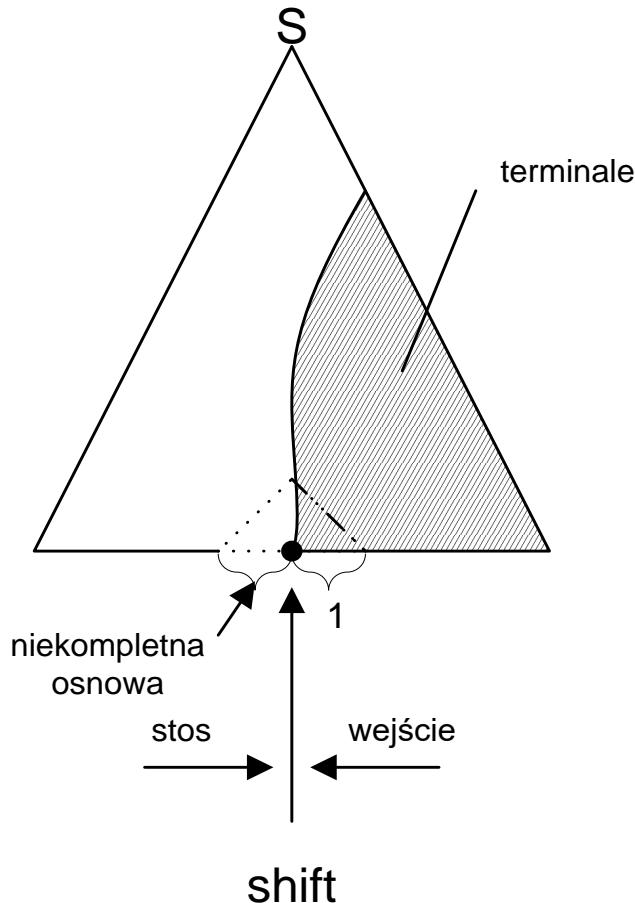
L R (k)

Przeglądanie
wejścia od
lewej strony
do prawej

Odtwarzanie
wywodu
prawostronnego

Wystarcza znajomość
"k" następnych symboli
łańcucha wejściowego i
historii
dotychczasowych
redukacji, aby wyznaczyć
jednoznacznie osnowę i
dokonać jej redukcji

Odtwarzanie wywodu prawostronnego metodą bottom-up



Gramatyka uzupełniona

Dla gramatyki $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

definiuje się gramatykę uzupełnioną G'

$$G' = \langle V \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S' \rangle$$

nowy symbol początkowy i dodatkowa produkcja $S' \rightarrow S$ mają wyłącznie znaczenie techniczne

Definicja gramatyki LR(k)

Gramatyka G jest gramatyką LR(k) dla $k \geq 0 \Leftrightarrow$ jeżeli z:

$$(i) \ S \xrightarrow[G'R]{*} \alpha Aw \xrightarrow[G'R]{*} \alpha\beta w$$

$$(ii) \ S \xrightarrow[G'R]{*} \gamma Bx \xrightarrow[G'R]{*} \alpha\beta y$$

$$(iii) \ FIRST_k(w) = FIRST_k(y)$$

wynika, że

$$\alpha Ay = \gamma Bx$$

tzn. $\alpha = \gamma$; $A = B$; $y = x$

$$A, B \in V$$

$$w, x, y \in \Sigma^*$$

$$\alpha, \beta, \gamma \in (V \cup \Sigma)^*$$

\Rightarrow wyprowadzenie prawostronne
 $G'R$

w gramatyce uzupełnionej G'

Definicja gramatyki LR(k)

Gramatyka G jest gramatyką LR(k) dla $k \geq 0 \Leftrightarrow$ jeżeli z:

$$(i) \ S' \xrightarrow[G'R]{*} \alpha Aw \xrightarrow[G'R]{*} \alpha\beta w$$

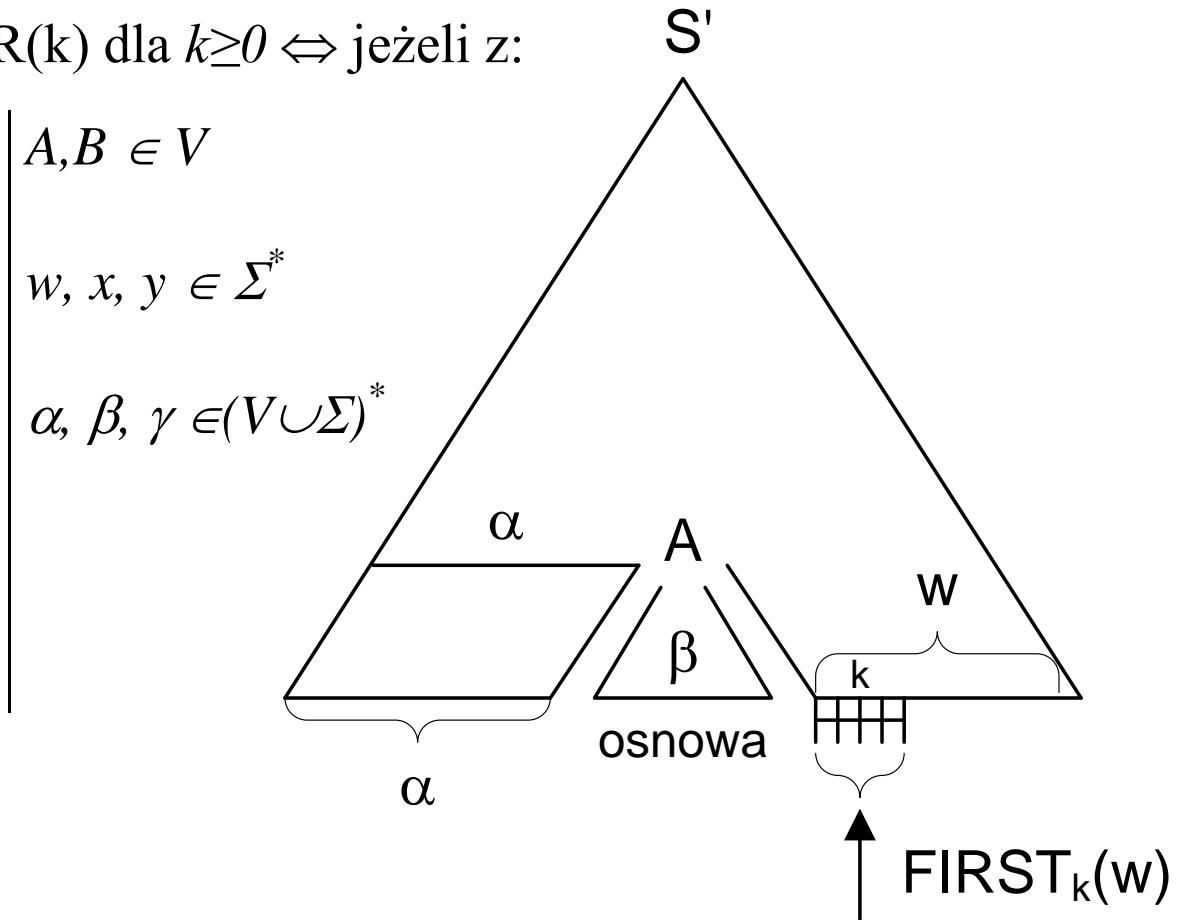
$$(ii) \ S' \xrightarrow[G'R]{*} \gamma Bx \xrightarrow[G'R]{*} \alpha\beta y$$

$$(iii) \ FIRST_k(w) = FIRST_k(y)$$

wynika, że

$$\alpha A y = \gamma B x$$

$$\text{tzn. } \alpha = \gamma; \quad A = B; \quad y = x$$



Przykład: $S \rightarrow Sa \mid a$

Gramatyka uzupełniona:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow Sa \\ S &\rightarrow a \end{aligned}$$

Gramatyka ta nie jest gramatyką LR(0), bo z tego, że

$$\begin{array}{ll} \text{(i)} \quad S' \xrightarrow{*} S' \Rightarrow S & \text{(ii)} \quad S' \xrightarrow{*} S \Rightarrow Sa \\ \color{blue} S' \qquad \alpha Aw \quad \alpha \beta w & \color{blue} S' \qquad \gamma Bx \quad \gamma \beta y \end{array}$$

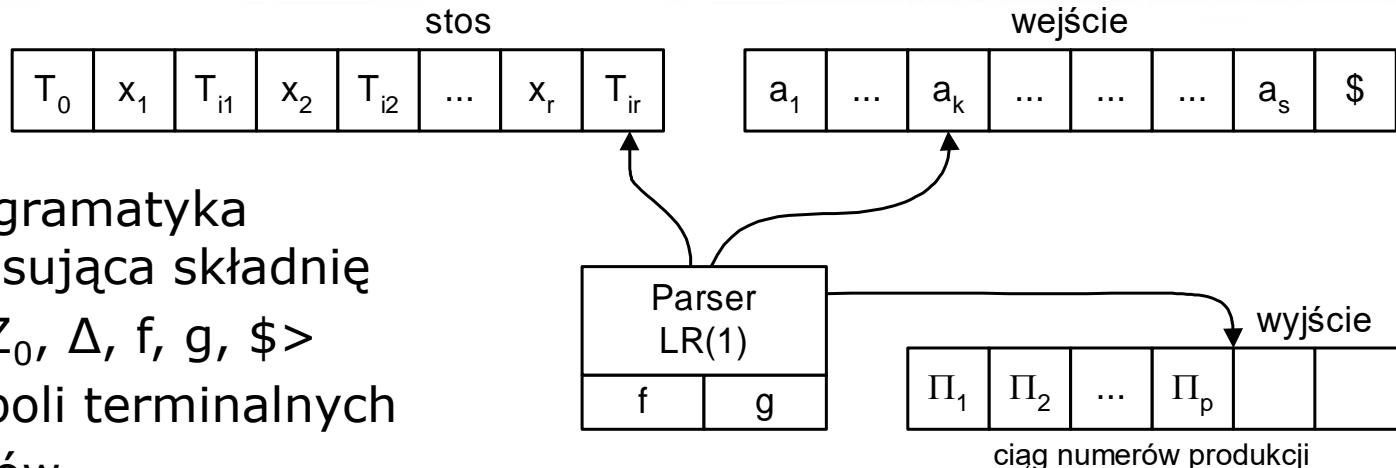
$$\begin{array}{ll} \text{(iii)} \quad FIRST_0(\varepsilon) = FIRST_0(a) = \{\varepsilon\} & \\ \color{blue} FIRST_0(w) \qquad \color{blue} FIRST_0(y) & \end{array}$$

nie wynika, że $S'a = S$

$$\color{blue} \alpha A y \quad \color{blue} \gamma B x$$

Nie można stwierdzić, czy S jest osnową nie znając następnego symbolu (którym może być $\$$ – wtedy S będzie osnową, lub a – wtedy osnową nie będzie S lecz Sa)

Parser LR(1)



$G = \langle V, \Sigma, P, S \rangle$ – gramatyka uzupełniona opisującą składnię

$A = \langle \Sigma, \mathfrak{S}, T_0, \Gamma, Z_0, \Delta, f, g, \$ \rangle$

Σ – zbiór symboli terminalnych

\mathfrak{S} - zbiór stanów

$T_0 \in \mathfrak{S}$ – stan początkowy

Γ – zbiór symboli stosowych

$$\Gamma = V \cup \Sigma \cup \mathfrak{S}$$

$Z_0 = T_0$ – symbol początkowy stosu oraz stan początkowy

Δ – alfabet wyjściowy: zbiór numerów produkcji

f – tablica działania parsera

g – tablica przejść parsera

$\$$ - ogranicznik końca słowa wejściowego

Automat parsera LR(1)

Analizator (parser) LR(1) jest zdeterminowanym automatem ze stosem, którego sterowanie określają dwie funkcje f i g :

f – funkcja działania

$g: \mathfrak{T} \times (\Sigma \cup \{\$\}) \alpha \{ \underline{\text{shift}} \ j, \underline{\text{red}} \ i, \underline{\text{err}}, \underline{\text{acc}} \}$

↑ ↑
numer stanu numer produkcji

g – funkcja przejścia

$g: \mathfrak{T} \times V \alpha \mathfrak{T} \cup \{\underline{\text{err}}\}$

gdzie:

\mathfrak{T} - zbiór stanów

$\mathfrak{T} = \{T_j : j = 0, 1, \dots, n\}$

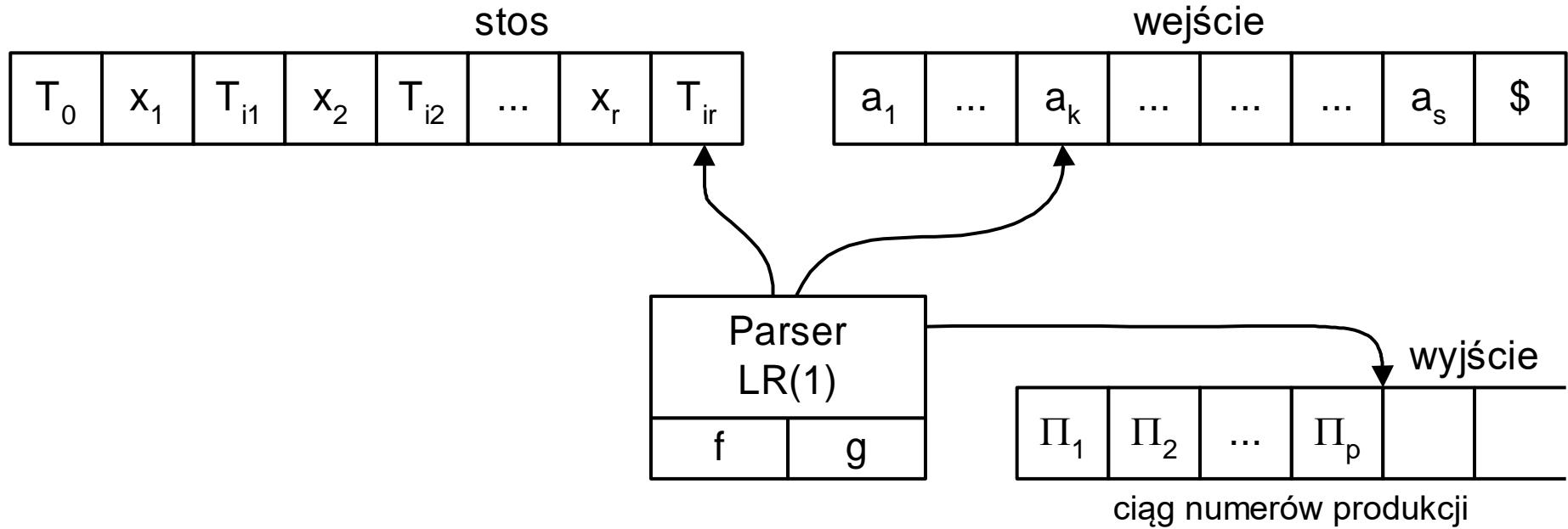
$i = 1, 2, \dots, m$

$m = \#P$

m – liczba produkcji gramatyki G

T_0 – stan początkowy, a zarazem początkowy symbol stosu

Automat parsera LR(1)



Konfiguracja parsera:

$$(T_0 x_1 T_{i_1} \dots x_r T_{i_r}, a_k \dots a_s \$, \pi_1 \dots \pi_p)$$

przy czym : x_i są symbolami gramatyki (nieterminalami lub terminalami)

Konfiguracja początkowa : $(T_0, a_1 a_2 \dots a_s \$, \varepsilon)$

Działanie parsera LR(1)

Niech parser znajduje się w konfiguracji:

$$(T_0x_1T_{i_1} \dots x_r T_{i_r}, a_k a_{k+1} \dots a_s \$, \pi)$$

(i) Jeśli $f(T_{i_r}, a_k) = \underline{\text{red}}$ i

oraz $A \rightarrow \gamma$ jest produkcją o numerze i

$$|\gamma| = d; d \leq r$$

to:

$$(T_0x_1T_{i_1} \dots x_{r-d} T_{i_{r-d}} \underbrace{\dots x_r T_{i_r}}, a_k a_{k+1} \dots a_s \$, \pi) \Rightarrow$$

zdejmie się $2d$ symboli

$$\Rightarrow (T_0x_1T_{i_1} \dots x_{r-d} T_{i_{r-d}} A T_j, a_k a_{k+1} \dots a_s \$, \pi)$$

gdzie $T_j = g(T_{i_{r-d}}, A)$

Działanie parsera LR(1)

Niech parser znajduje się w konfiguracji:

$$(T_0 x_1 T_{i_1} \dots x_r T_{i_r}, a_k a_{k+1} \dots a_s \$, \pi)$$

(ii) Jeśli $f(T_{i_r}, a_k) = \underline{shift} j$

to:

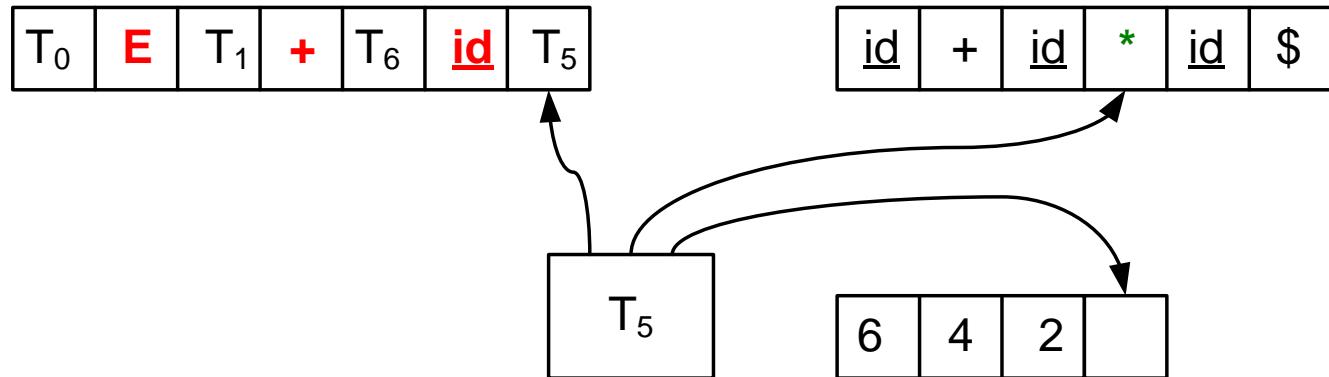
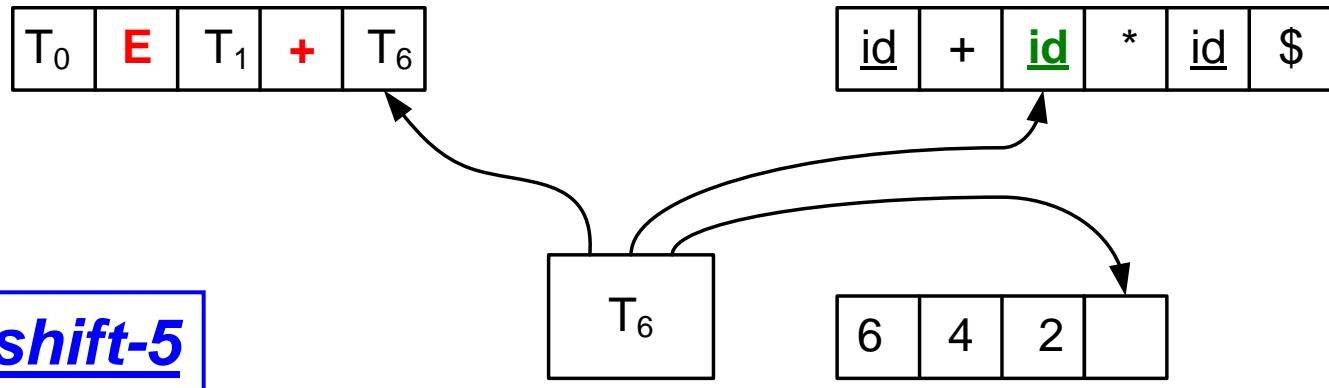
$$(T_0 x_1 T_{i_1} \dots x_r T_{i_r}, a_k a_{k+1} \dots a_s \$, \pi) \Rightarrow$$

$$\Rightarrow (T_0 x_1 T_{i_1} \dots x_r T_{i_{r-1}} a_k T_j, a_{k+1} \dots a_s \$, \pi)$$

(iii) Jeśli $f(T_{i_r}, a_k) = \underline{acc}$ — akceptacja,
parsing zakończony poprawnie

(iv) Jeśli $f(T_{i_r}, a_k) = \underline{err}$ — błąd syntaktyczny

Krok „shift”

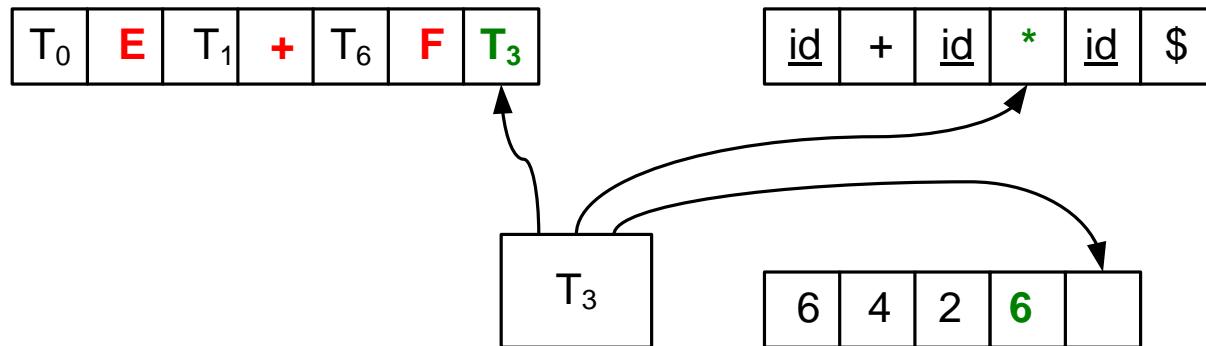
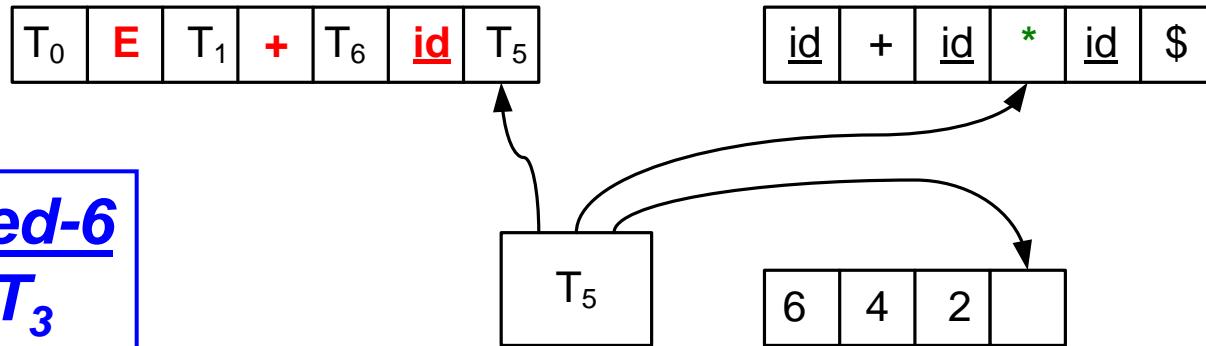


Przykład – gramatyka jednoznaczna

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

stan	f						g		
	\$	+	*	()	id	E	T	F
T_0				<u>shift4</u>		<u>shift5</u>		T_1	T_2
T_1	<u>acc</u>	<u>shift 6</u>						T_3	
T_2	<u>red-2</u>	<u>red-2</u>	<u>shift</u>		<u>red-2</u>				
T_3	<u>red-4</u>	<u>red-4</u>	<u>red-4</u>		<u>red-4</u>				
T_4				<u>shift4</u>		<u>shift5</u>		T_8	T_2
T_5	<u>red-6</u>	<u>red-6</u>	<u>red-6</u>		<u>red-6</u>				
T_6				<u>shift4</u>		<u>shift5</u>			T_9
T_7				<u>shift4</u>		<u>shift5</u>			T_{10}
T_8		<u>shift6</u>			<u>shift11</u>				
T_9	<u>red-1</u>	<u>red-1</u>	<u>shift7</u>		<u>red-1</u>				
T_{10}	<u>red-3</u>	<u>red-3</u>	<u>red-3</u>		<u>red-3</u>				
T_{11}	<u>red-5</u>	<u>red-5</u>	<u>red-5</u>		<u>red-5</u>				

Krok „redukacji”



Przykład – gramatyka jednoznaczna

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

stan	f						g		
	\$	+	*	()	id	E	T	F
T_0				<u>shift4</u>		<u>shift5</u>	T_1	T_2	T_3
T_1	<u>acc</u>	<u>shift 6</u>							
T_2	<u>red-2</u>	<u>red-2</u>	<u>shift</u>		<u>red-2</u>				
T_3	<u>red-4</u>	<u>red-4</u>	<u>red-4</u>		<u>red-4</u>				
T_4				<u>shift4</u>		<u>shift5</u>	T_8	T_2	T_3
T_5	<u>red-6</u>	<u>red-6</u>	<u>red-6</u>		<u>red-6</u>				
T_6				<u>shift4</u>		<u>shift5</u>		T_9	T_3
T_7				<u>shift4</u>		<u>shift5</u>			T_{10}
T_8		<u>shift6</u>			<u>shift11</u>				
T_9	<u>red-1</u>	<u>red-1</u>	<u>shift7</u>		<u>red-1</u>				
T_{10}	<u>red-3</u>	<u>red-3</u>	<u>red-3</u>		<u>red-3</u>				
T_{11}	<u>red-5</u>	<u>red-5</u>	<u>red-5</u>		<u>red-5</u>				

Przykład

Gramatyka uzupełniona:

- (0) $S' \rightarrow S$
- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

W tablicy parsera
nie umieszczamy
pozycji err

Stany	<i>f</i>			<i>g</i> <i>S</i>
	<i>a</i>	<i>b</i>	\$	
T_0	<u>red</u> 2	<u>err</u>	<u>red</u> 2	T_1
T_1	<u>shift</u> 2	<u>err</u>	<u>acc</u>	<u>err</u>
T_2	<u>red</u> 2	<u>red</u> 2	<u>err</u>	T_3
T_3	<u>shift</u> 4	<u>shift</u> 5	<u>err</u>	<u>err</u>
T_4	<u>red</u> 2	<u>red</u> 2	<u>err</u>	T_6
T_5	<u>red</u> 1	<u>err</u>	<u>red</u> 1	<u>err</u>
T_6	<u>shift</u> 4	<u>shift</u> 7	<u>err</u>	<u>err</u>
T_7	<u>red</u> 1	<u>red</u> 1	<u>err</u>	<u>err</u>

Przykład

Gramatyka uzupełniona:

- (0) $S' \rightarrow S$
- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

Stany	f			g
	a	b	$\$$	
T_0	<u>red</u> 2		<u>red</u> 2	T_1
T_1	<u>shift</u> 2		<u>acc</u>	
T_2	<u>red</u> 2	<u>red</u> 2		T_3
T_3	<u>shift</u> 4	<u>shift</u> 5		
T_4	<u>red</u> 2	<u>red</u> 2		T_6
T_5	<u>red</u> 1		<u>red</u> 1	
T_6	<u>shift</u> 4	<u>shift</u> 7		
T_7	<u>red</u> 1	<u>red</u> 1		

Przykład parsingu

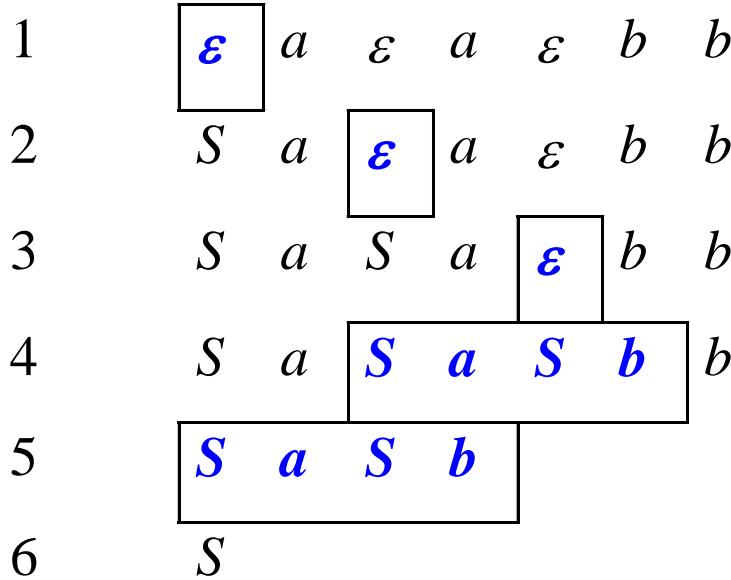
(T ₀ , aabb\$, ε)	φ	(T ₀ ST ₁ , aabb\$, 2)	(red-2)
	φ	(T ₀ ST ₁ aT ₂ , abb\$, 2)	(shift)
	φ	(T ₀ ST ₁ aT ₂ ST ₃ , abb\$, 22)	(red-2)
	φ	(T ₀ ST ₁ aT ₂ ST ₃ aT ₄ , bb\$, 22)	(shift)
	φ	(T ₀ ST ₁ aT ₂ ST ₃ aT ₄ ST ₆ , bb\$, 222)	(red-2)
	φ	(T ₀ ST ₁ aT ₂ ST₃aT₄ST₆bT₇ , b\$, 222)	(shift)
	φ	(T ₀ ST ₁ aT ₂ ST ₃ , b\$, 2221)	(red-1)
	φ	(T ₀ ST₁aT₂ST₃bT₅ , \$, 2221)	(shift)
	φ	(T ₀ ST ₁ , \$, 22211)	(red-1)
		(acc)	

Przykład c.d.

Sprawdzenie (zgodnie z wywodem prawostronnym dokonywanym przez parser):

$$aabb \stackrel{2}{\underset{R}{\Leftarrow}} Saabb \stackrel{2}{\underset{R}{\Leftarrow}} SaSabb \stackrel{2}{\underset{R}{\Leftarrow}} SaSaSbb \stackrel{1}{\underset{R}{\Leftarrow}} SaSb \stackrel{1}{\underset{R}{\Leftarrow}} S \quad \text{o.k.}$$

Formy zdaniowe z zaznaczonymi osnowami





AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Parsery LR(1) – część 2

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Przedrostki żywotne Sytuacje dopuszczalne

Żywotny przedrostek (viable prefix)

γ - żywotny (aktywny) prefiks gramatyki G

$\Leftrightarrow \gamma$ - prefiks łańcucha $\alpha\beta$

*

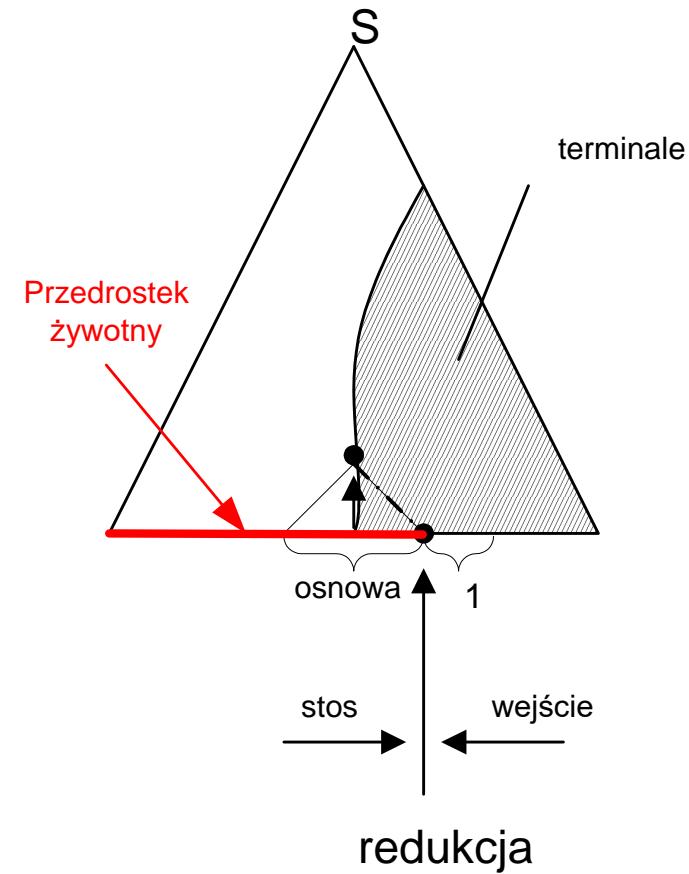
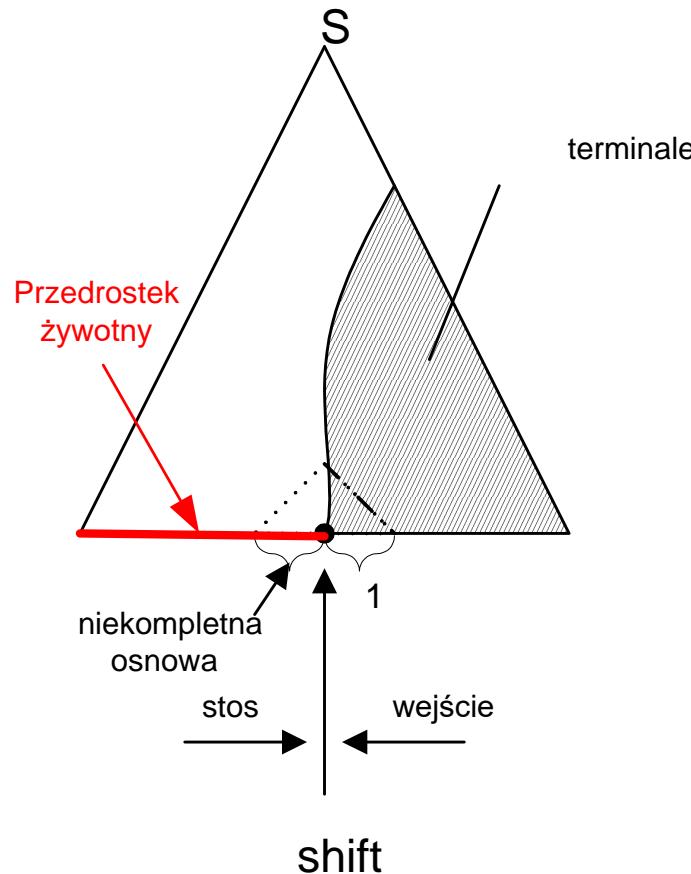
$$S \xrightarrow{R} \alpha A w \xrightarrow{R} \alpha \beta w$$

gdzie: $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$ $w \in \Sigma^*$ $A \in V$

Żywotny przedrostek jest to łańcuch będący przedrostkiem pewnej prawostroñnie wyprowadzalnej formy zdaniowej, nie wychodzący poza prawy koniec jej osnowy.

Przedrostki żywotne

Sytuacje dopuszczalne



Przedrostki żywotne Sytuacje dopuszczalne

LR(1)–sytuacja (LR(1)–item)

$[A \rightarrow \beta_1 \bullet \beta_2, v]$ - jest LR(1)–sytuacją, gdy $(A \rightarrow \beta_1 \beta_2) \in P$

LR(1)–sytuacja dopuszczalna (LR(1)–item valid for viable prefix)

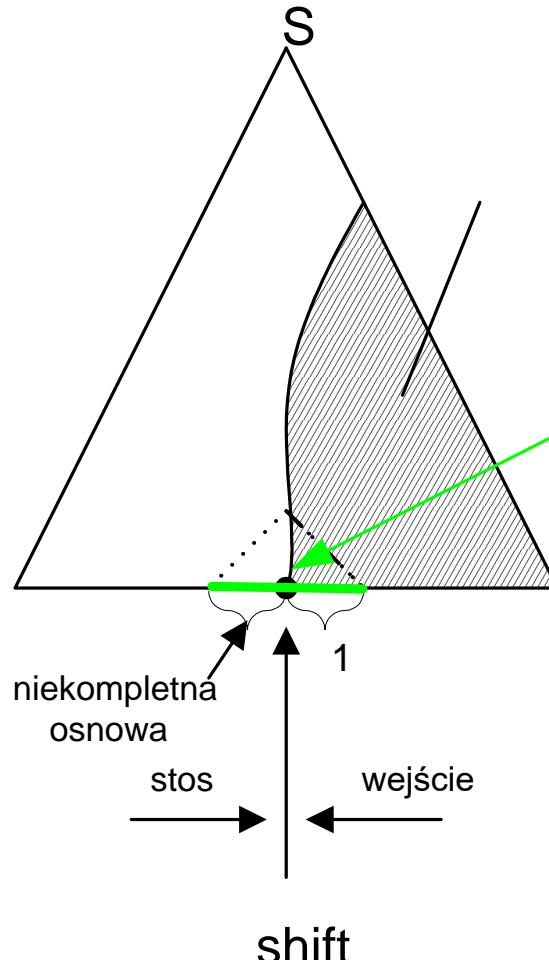
$[A \rightarrow \beta_1 \bullet \beta_2, v]$ – LR(1)–sytuacja jest sytuacją dopuszczalną dla żywotnego prefiksu $\alpha\beta_1$, wtedy i tylko wtedy, gdy

\exists wywód:

$$\left(S \xrightarrow[R]{*} \alpha A w \xrightarrow[R]{} \alpha \beta_1 \beta_2 w \quad \wedge \quad v \in FIRST_1(w) \right)$$

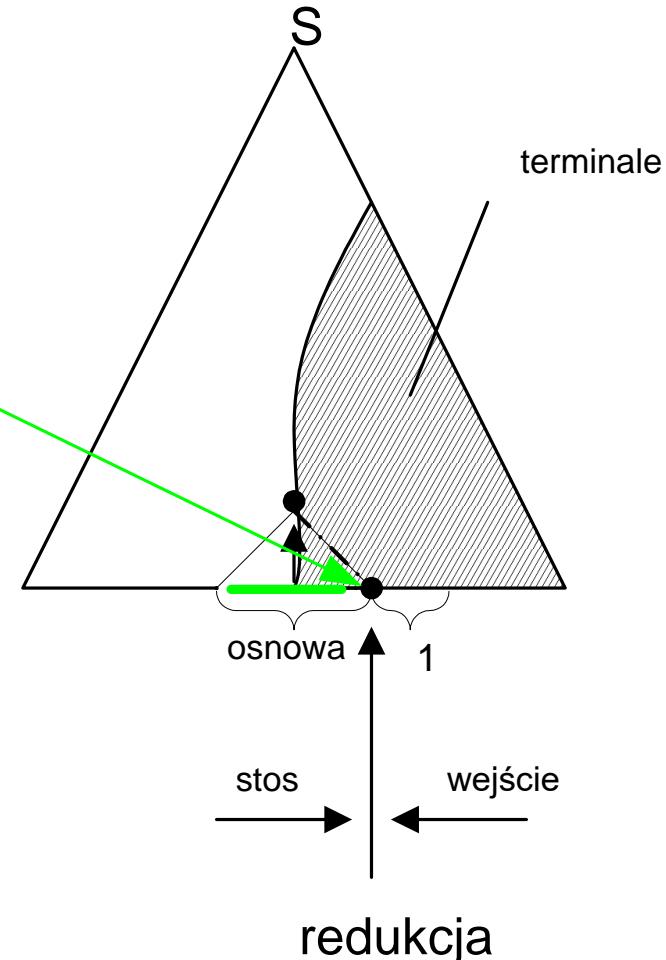
Przedrostki żywotne

Sytuacje dopuszczalne



terminale

sytuacja dopuszczalna
ilustruje okolice
wierzchołka stosu
a właściwie podaje interpretację
tego obszaru przez parser

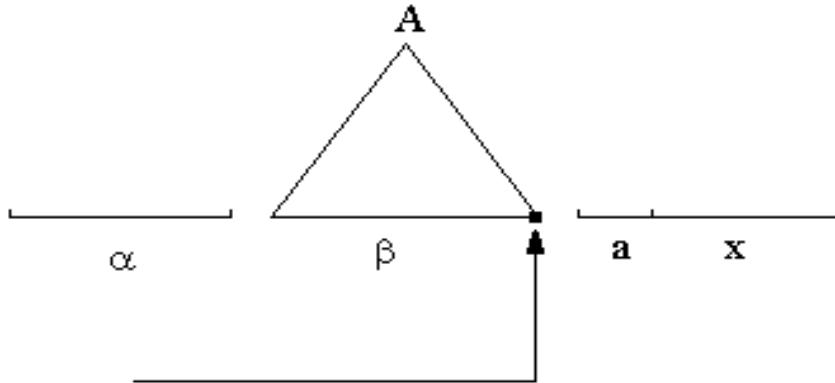


terminale

osnowa
1
stos
wejście
redukcja

Przedrostki żywotne

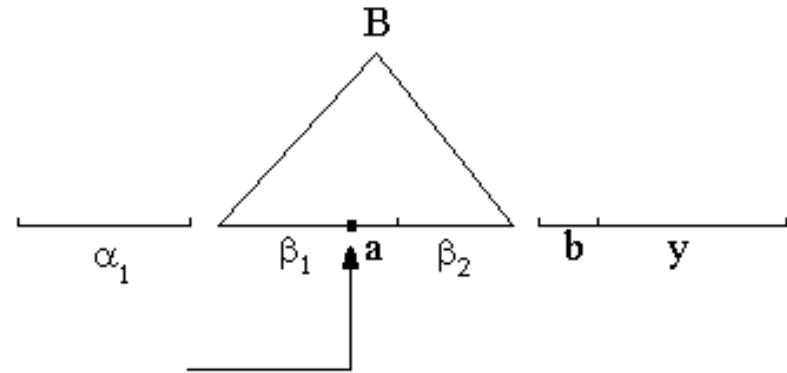
Sytuacje dopuszczalne



Sytuacja **$[A \rightarrow \beta \bullet, a]$**
 dopuszczalna dla żywotnego
 przedrostka $\alpha\beta$

Decyzja: redukcja wg produkcji
 $A \rightarrow \beta$ przy podglądzaniu a na
 wejściu

Efekt: nowa konfiguracja z
 żywotnym przedrostkiem αA



Sytuacja **$[B \rightarrow \beta_1 \bullet a \beta_2, b]$**
 dopuszczalna dla żywotnego przedrostka
 $\alpha_1\beta_1$, przy czym $a \in \Sigma$.

Decyzja: przesunięcie (shift) terminala a
 z wejścia na stos

Efekt: nowa konfiguracja opisana
 sytuacją: **$[B \rightarrow \beta_1 a \bullet \beta_2, b]$**
 dopuszczalną dla żywotnego przedrostka
 $\alpha_1\beta_1 a$

Przykład

Gramatyka: $S \rightarrow SaSb/\varepsilon$

analizowane słowo: $aabb$

$$S \xrightarrow{*} S a \textcolor{green}{S} b \Rightarrow S a \quad \textcolor{green}{S} \ a \ S \ \textcolor{green}{b} \ b$$

$$\textcolor{blue}{S} \quad \overline{\alpha} \ A \ w \quad \overline{\alpha} \quad \overline{\beta_1} \ \beta_2 \ w$$

<u>Forma zdaniowa:</u>	<u>Stos (bez stanów)</u>	<u>Wejście</u>	<u>Sytuacja</u>
$SaSaSbb$	$(*) SaSaS$	$\downarrow bb$	$[S \rightarrow SaS \bullet b, b]$ dopuszczalna dla żywotnego prefiku $SaSaS$
$SaSaSbb$	$(**) SaSaSb$	$\downarrow b$	$[S \rightarrow SaSb \bullet, b]$ dopuszczalna dla żywotnego prefiku $SaSaSb$

(*) Żywotnemu prefiksowi $SaSaS$ odpowiada stan T_6

(**) Żywotnemu prefiksowi $SaSaSb$ odpowiada stan T_7

Domykanie zbioru LR(1)-sytuacji dopuszczalnych

Załóżmy, że sytuacja $[A \rightarrow \alpha \bullet B\beta, a]$ jest dopuszczalna dla pewnego żywotnego prefiksu γ , co oznacza, że istnieje wyprowadzenie prawostronne:

$$S \xrightarrow[\gamma]{} \delta Aax \xrightarrow[\gamma]{} \delta \alpha B\beta ax$$

Przypuśćmy, że $\beta ax \xrightarrow[\gamma]{} by$ (by – łańcuch terminalny rozpoczynający się symbolem b). Wtedy dla każdej produkcji $B \rightarrow \eta$

$$S \xrightarrow[\gamma]{} \delta Aax \xrightarrow[\gamma]{} \delta \alpha B\beta ax \xrightarrow[\gamma]{} \delta \alpha \eta \beta ax \xrightarrow[\gamma]{} \delta \alpha \eta by$$

Czyli dla żywotnego prefiksu γ dopuszczalna jest także sytuacja $[B \rightarrow \bullet \eta, b]$, gdzie:

$$b \in FIRST_1(\beta ax) = FIRST_1(\beta a)$$

Algorytm domykania zbioru sytuacji dopuszczalnych

We: Zbiór I sytuacji dopuszczalnych dla pewnego żywotnego prefiku
(w gramatyce uzupełnionej G')

Wy: Zbiór I będący domknięciem wejściowego zbioru sytuacji dopuszczalnych

Metodę ilustruje funkcja CLOSURE(I);

function CLOSURE(I);

begin

repeat

for każda sytuacja $[A \rightarrow \alpha \bullet B \beta, a] \in I$ do

for każda produkcja $(B \rightarrow \eta) \in P'$ do

for każdy $b \in FIRST_1(\beta a)$ do

$I := I \cup \{[B \rightarrow \bullet \eta, b]\};$

until nic nowego nie dodano do I ;

return (I);

end;

Funkcja GOTO

Funkcja GOTO:

$$\{I(\gamma) : \gamma - \text{żywotny prefiks}\} \times \{X : X \in (V \cup \Sigma)\} \alpha \{I(\gamma') : \gamma' \in (V \cup \Sigma)^+, \gamma' = \gamma X\}$$

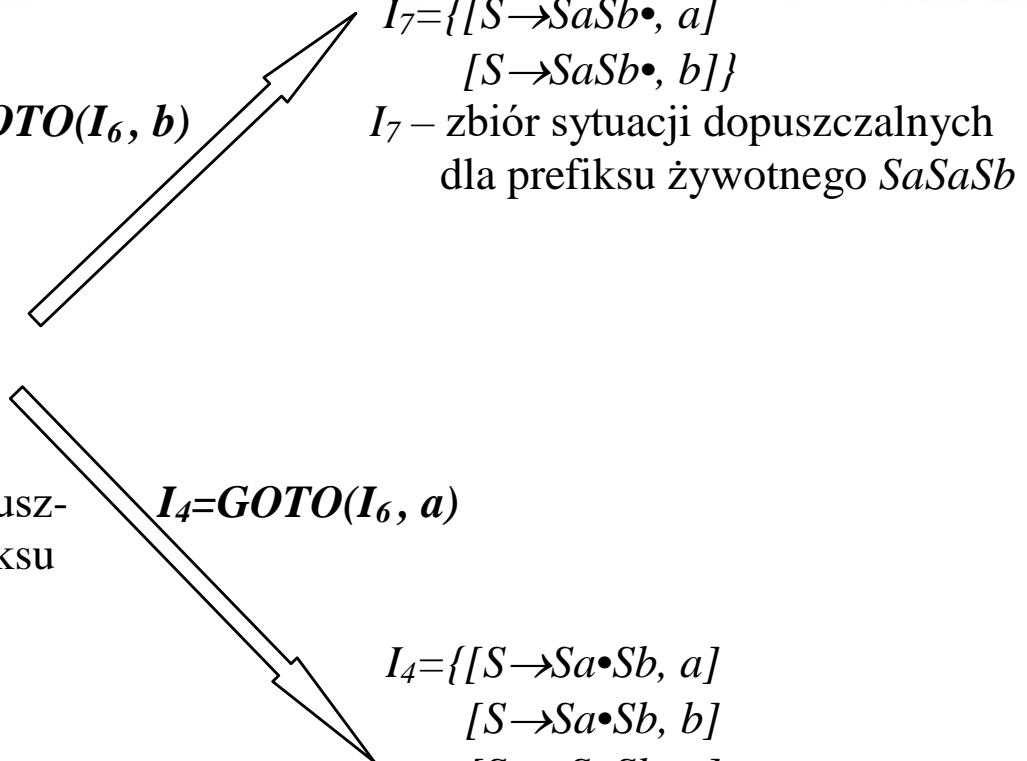
gdzie: $I(\gamma)$ – zbiór wszystkich sytuacji dopuszczalnych dla prefiku żywotnego γ

Przykład: $S \rightarrow S a S b / \epsilon$

- (a) $S \xrightarrow[R]{*} S a S a S \underset{\uparrow}{g} b b$
- (b) $S \xrightarrow[R]{*} S a S a S \underset{\uparrow}{g} a b b b$
- Prefiksem żywotnym obu form zdaniowych jest $S a S a S$

Przykład funkcji GOTO

$I_6 = \{ [S \rightarrow SaS \bullet b, a]$
 $[S \rightarrow SaS \bullet b, b]$
 $[S \rightarrow S \bullet aSb, a]$
 $[S \rightarrow S \bullet aSb, b] \}$
 $I_6 - \text{zbiór sytuacji dopuszczalnych dla prefiku żywotnego } SaSaS$



$I_7 = \{ [S \rightarrow SaSb \bullet, a]$
 $[S \rightarrow SaSb \bullet, b] \}$
 $I_7 - \text{zbiór sytuacji dopuszczalnych dla prefiku żywotnego } SaSaSb$

$I_4 = \{ [S \rightarrow Sa \bullet Sb, a]$
 $[S \rightarrow Sa \bullet Sb, b]$
 $[S \rightarrow • SaSb, a]$
 $[S \rightarrow • SaSb, b]$
 $[S \rightarrow •, a]$
 $[S \rightarrow •, b] \}$
 $I_4 - \text{zbiór sytuacji dopuszczalnych dla prefiku żywotnego } SaSaSa$

Wyznaczanie funkcji GOTO

We: I – zbiór wszystkich sytuacji dopuszczalnych dla prefiksu aktywnego γ , $X \in (V \cup \Sigma)$

Wy: J – zbiór wszystkich sytuacji dopuszczalnych dla prefiksu aktywnego γX

Metodę ilustruje funkcja GOTO (I , X)

function GOTO (I , X);

begin

$J := \emptyset$;

for każda_sytuacja $[A \rightarrow \alpha \bullet X \beta, a] \in I$ do

$J := J \cup \{ [A \rightarrow \alpha X \bullet \beta, a] \};$

return CLOSURE (J);

end;



Kanoniczny system zbiorów LR(1)-sytuacji dopuszczalnych

J – kanoniczny system zbiorów LR(1) sytuacji dopuszczalnych

J – jest zbiorem wszystkich zbiorów $I(\gamma)$ LR(1)-sytuacji dopuszczalnych,

gdzie : γ – żywotny prefiks w gramatyce G'

Konstrukcja kanonicznego systemu zbiorów LR(1)-sytuacji dopuszczalnych

We: G' – gramatyka uzupełniona $\langle V', \Sigma, P', S' \rangle$

dla gramatyki $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

Wy: J – kanoniczny system zbiorów LR(1) sytuacji dopuszczalnych dla G .

Metodę ilustruje funkcja ITEMS (G'):

function ITEMS (G');

begin

$J := \{\text{CLOSURE}(\{[S' \rightarrow \bullet S, \$]\})\};$

repeat

for każdy zbiór $I \in J$ do

for każdy $X \in (V \cup \Sigma)$ do

if $\text{GOTO}(I, X) \neq \emptyset$ then

$J := J \cup \{\text{GOTO}(I, X)\};$

until nic nowego nie dodano do J ;

return J ;

end;

Przykład

Wyznaczanie systemu kanonicznego zbiorów LR(1)-sytuacji dopuszczalnych dla gramatyki uzupełnionej

$$S' \rightarrow S$$

$$S \rightarrow SaSb / \varepsilon$$

$$I_0 = I(\varepsilon) = \{[S' \rightarrow \bullet S, \$]\}, \longrightarrow \text{FIRST}_1(\$) = \{\$\}$$

(domykamy)

$$[S \rightarrow \bullet SaSb, \$],$$

$$[S \rightarrow \bullet, \$],$$

(domykamy powtórnie)

$$[S \rightarrow \bullet SaSb, a]$$

$$[S \rightarrow \bullet, a]\}$$

$$\longrightarrow \text{FIRST}_1(aSb\$) = \{a\}$$

$$\longrightarrow \text{FIRST}_1(aSba) = \{a\}$$

(nic nowego nie da się dołączyć)

Przykład c.d.

Uproszczenie zapisu:

zamiast: $[A \rightarrow \beta_1 \bullet \beta_2, x], [A \rightarrow \beta_1 \bullet \beta_2, y]$

piszemy: $[A \rightarrow \beta_1 \bullet \beta_2, x / y])$

Ostatecznie:

$$I_o = \{[S \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a]\}$$

Przykład c.d.

$$I_o = \{[S \rightarrow \bullet S, \$], [S \rightarrow \bullet SaSb, \$ / a], [S \rightarrow \bullet, \$ / a]\}$$

$$I_1 = I(S) = GOTO(I_o, S) = GOTO(I(\varepsilon), S)$$

$$I_1 = \{[S' \rightarrow S\bullet, \$], [S \rightarrow S\bullet aSb, \$ / a]\}$$

(domknięcie nie daje nowych sytuacji)

$$GOTO(I_o, a) = GOTO(I(\varepsilon), a) = \emptyset$$

$$GOTO(I_o, b) = GOTO(I(\varepsilon), b) = \emptyset$$

(gdyż ani „a” ani „b” nie są żywotnymi prefiksami w tej gramatyce)

Przykład c.d.

$$I_1 = \{[S' \rightarrow S\bullet, \$], [S \rightarrow S\bullet a S b, \$ / a]\}$$

$$I_2 = I(Sa) = GOTO(I(S), a) = GOTO(I_1, a)$$

$$I_2 = \{[S \rightarrow Sa\bullet Sb, \$ / a], \quad | \quad \text{FIRST}_1(b\$) = \{b\}, \\ \quad | \quad \text{FIRST}_1(ba) = \{b\} \\ \quad | \quad (\text{domykamy})$$

$$[S \rightarrow \bullet Sa S b, b] \quad | \quad \text{FIRST}_1(a S b b) = \{a\}$$

$$[S \rightarrow \bullet, b], \quad | \quad (\text{domykamy powtórnie})$$

$$[S \rightarrow \bullet Sa S b, a] \quad | \quad \text{FIRST}_1(a S b a) = \{a\}$$

$$[S \rightarrow \bullet, a] \quad | \quad (\text{nic nowego nie da się dołączyć})$$

$$I_2 = \{[S \rightarrow Sa\bullet Sb, \$ / a], [S \rightarrow \bullet Sa S b, b / a], [S \rightarrow \bullet, b / a]\}$$

Przykład c.d.

$$I_1 = \{ [S' \rightarrow S\bullet, \$], [S \rightarrow S\bullet a S b, \$ / a] \}$$

$$\text{GOTO}(I_1, S) = \text{GOTO}(I(S), S) = \emptyset$$

$$\text{GOTO}(I_1, b) = \text{GOTO}(I(S), b) = \emptyset$$

(gdyż ani „SS” ani „Sb” nie są żywotnymi prefiksami w tej gramatyce)

$$I_2 = \{ [S \rightarrow S a \bullet S b, \$ / a], [S \rightarrow \bullet S a S b, b / a], [S \rightarrow \bullet, b / a] \}$$

$$I_3 = I(S a S) = \text{GOTO}(I(S a), S) = \text{GOTO}(I_2, S)$$

$$I_3 = \{ [S \rightarrow S a S \bullet b, \$ / a], [S \rightarrow S \bullet a S b, b / a] \}$$

(domknięcie nie daje nowych sytuacji)

$$\text{GOTO}(I_2, a) = \text{GOTO}(I_2, b) = \emptyset$$

Przykład c.d.

$$I_3 = \{[S \rightarrow SaS \bullet b, \$ / a], [S \rightarrow S \bullet aSb, b / a]\}$$

$$I_4 = I(SaSa) = GOTO(I(SaS), a) = GOTO(I_3, a)$$

$$I_4 = \{[S \rightarrow Sa \bullet Sb, b / a], [S \rightarrow \bullet SaSb, b / a], [S \rightarrow \bullet, b / a]\}$$

(Uwaga: Podobny zbiór sytuacji miał nr 2 i różnił się tylko postacią prawych stron sytuacji)

$$I_5 = I(SaSb) = GOTO(I(SaS), b) = GOTO(I_3, b)$$

$$I_5 = \{[S \rightarrow SaSb \bullet, \$ / a]\}$$

$$GOTO(I_3, S) = \emptyset$$

$$I_6 = I(SaSaS) = GOTO(I(SaSa), S) = GOTO(I_4, S)$$

$$I_6 = \{[S \rightarrow SaS \bullet b, b / a], [S \rightarrow S \bullet aSb, b / a]\}$$

(Porównaj zbiór I_3)

Przykład c.d.

$$I_4 = \{[S \rightarrow Sa \bullet Sb, b/a], [S \rightarrow \bullet SaSb, b/a], [S \rightarrow \bullet, b/a]\}$$

$$I_5 = \{[S \rightarrow SaSb \bullet, \$/a]\}$$

$$I_6 = \{[S \rightarrow SaS \bullet b, b/a], [S \rightarrow S \bullet aSb, b/a]\}$$

$$\text{GOTO}(I_4, a) = \text{GOTO}(I_4, b) = \emptyset$$

$$\text{GOTO}(I_5, a) = \text{GOTO}(I_5, b) = \text{GOTO}(I_5, \$) = \emptyset$$

$$I_7 = I(SaSaSb) = \text{GOTO}(I(SaSaS), b) = \text{GOTO}(I_6, b)$$

$$I_7 = \{[S \rightarrow SaSb \bullet, b/a]\} \quad (\text{Porównaj zbiór } I_5)$$

$$I(SaSaSa) = \text{GOTO}(I(SaSaS), a) = \text{GOTO}(I_6, a) = I_4$$

(**Uwaga:** tutaj otrzymaliśmy zbiór sytuacji identycznych ze zbiorem I_4 dla prefiksu aktywnego „SaSa”)

$$\text{GOTO}(I_6, S) = \emptyset$$

$$\text{GOTO}(I_7, S) = \text{GOTO}(I_7, a) = \text{GOTO}(I_7, b) = \emptyset$$

Zgodny zbiór sytuacji dopuszczalnych

$G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

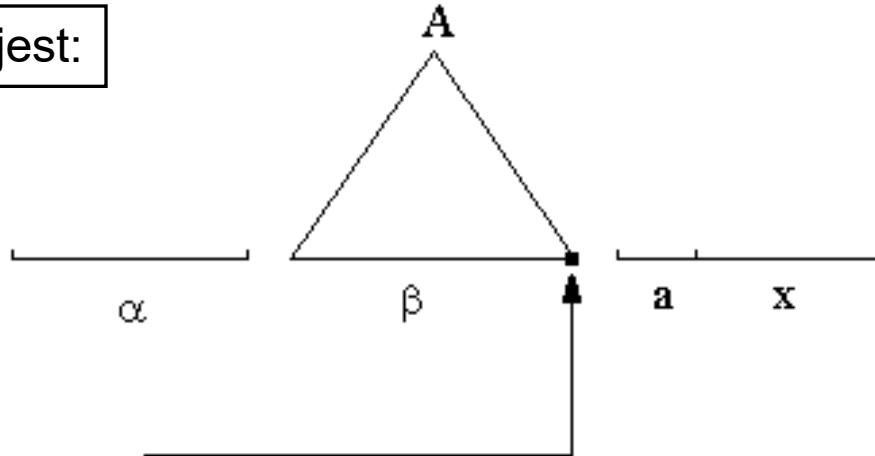
I – zbiór LR(1)-sytuacji dla gramatyki G

Definicja: Zbiór sytuacji I jest zgodny
 \Leftrightarrow nie zawiera dwu różnych sytuacji:

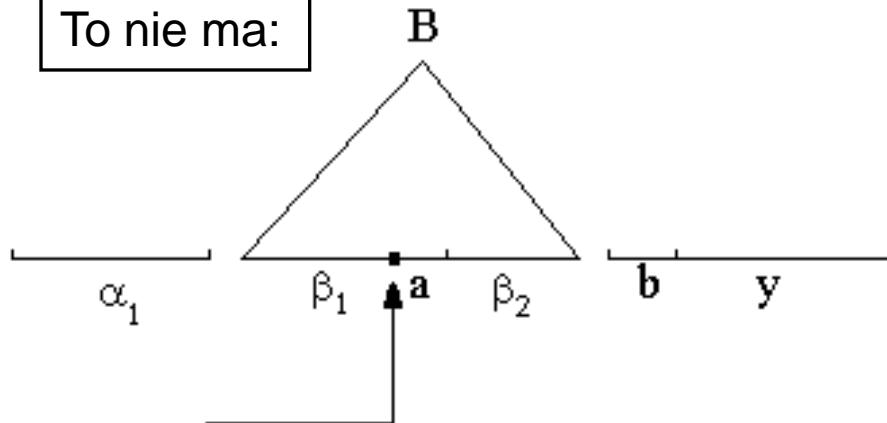
- (i) typu $[A \rightarrow \beta^\bullet, a]$
i $[B \rightarrow \beta_1^\bullet a \beta_2, b]$
- (ii) typu $[A \rightarrow \beta^\bullet, a]$
i $[B \rightarrow \gamma^\bullet, a]$

Zgodny zbiór sytuacji dopuszczalnych

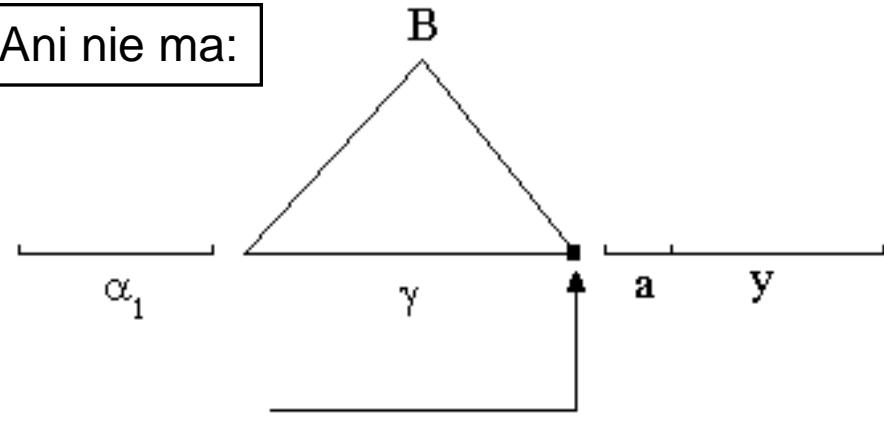
Jeśli jest:



To nie ma:



Ani nie ma:



Czy gramatyka jest LR(1)?

Twierdzenie: Niech $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

J – kanoniczny system zbiorów LR(1)-sytuacji dopuszczalnych dla G

G jest LR(1)-gramatyką $\Leftrightarrow (\forall I \in J) (I - \text{jest zgodny})$

Twierdzenie to stanowi najczęściej stosowany sposób przeprowadzenia weryfikacji, czy gramatyka G jest klasy LR(1).

Konstruowanie tablic parsera LR(1)

We: G' – gramatyka uzupełniona dla gramatyki bezkontekstowej G
 Tablica parsera LR(1) – funkcje f i g

Metoda:

1. Konstruujemy \mathbf{J} – kanoniczny system zbiorów LR(1) sytuacji dopuszczalnych dla G'
2. Badamy zgodność każdego zbioru $I \in \mathbf{J}$

Jeśli choć jeden zbiór I nie jest zgodny, gramatyka nie jest LR(1) \leftrightarrow STOP !!!

Numerujemy następnie produkcję gramatyki G' .

Konstruowanie tablic parsera LR(1)

3. for każdy zbiór $I_j \in J$ do

begin

utwórz w tablicy parsera stan $T_j \in \mathfrak{T}$ dla analizowanego $I_j \in J$;

for każda sytuacja ze zbioru I_j do

begin

- (a) if $[A \rightarrow \alpha \bullet a\beta, b] \in I_j$ and $a \in \Sigma$ and $GOTO(I_j, a) = I_k$ then
 $f(T_j, a) := \underline{shift} - k$;
- (b) if $[A \rightarrow \alpha \bullet, a] \in I_j$ and $A \neq S'$ and i -numer produkcji $(A \rightarrow \alpha) \in P$ then
 $f(T_j, a) := \underline{red} - i$;
- (c) if $[S' \rightarrow S \bullet, \$] \in I_j$ then
 $f(T_j, \$) := \underline{acc}$;
 end;

for każdy $A \in V$ do

if $GOTO(I_j, A) \neq \emptyset$ and $GOTO(I_j, A) = I_k$ then
 $g(T_j, A) := T_k$;

end;

Konstruowanie tablic parsera LR(1)

4. for każdy $T_j \in \mathfrak{T}$ do
begin
 for każdy $a \in \Sigma \cup \{\$\}$ do
 if $f(T_j, a)$ nieokreślone then
 $f(T_j, a) := \underline{err}$;
 for każdy $A \in V$ do
 if $g(T_j, A)$ nieokreślone then
 $g(T_j, A) := \underline{err}$;
 end;
5. Stanem początkowym parsera jest ten stan, który odpowiada zbiorowi $I \in \mathbb{J}$, dla którego $[S' \rightarrow \bullet S, \$] \in I$;

Gramatyki i parsery LR(0)

Omawiane postępowanie dotyczy gramatyk LR(1). Gramatyki LR(0) definiują stosunkowo wąską klasę języków bezkontekstowych posiadających własność przedrostkową. Poza tym dają z reguły tablice parsera o wyraźnie większym rozmiarze niż LR(1). Ponieważ postępowanie prowadzące do konstrukcji parsera LR(0) różni się nieco od przedstawionego powyżej (wymaga innego zdefiniowania funkcji f i g parsera), nie będzie przedmiotem naszego zainteresowania.

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

(1) $S \rightarrow SaSb$

(2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = \{ [S' \rightarrow S \bullet, \$], \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_1 = GOTO(I_o, S)$$

$$I_2 = GOTO(I_1, a)$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	$\$$	<i>S</i>
T_o	<u><i>red - 2</i></u>		<u><i>red - 2</i></u>	T_1
T_1	<u><i>shift - 2</i></u>		<u><i>acc</i></u>	
T_2	<u><i>red - 2</i></u>	<u><i>red - 2</i></u>		T_3
...				

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = GOTO(I_o, S)$$

$$I_1 = \{ [S' \rightarrow S \bullet, \$], \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = GOTO(I_1, a)$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	<i>\$</i>	<i>S</i>
T_o	<u>red</u> - 2		<u>red</u> - 2	$\textcolor{blue}{T}_1$
T_1	<u>shift</u> - 2		<u>acc</u>	
T_2	<u>red</u> - 2	<u>red</u> - 2		T_3
...				

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = \{ \textcolor{blue}{[S' \rightarrow S \bullet, \$]}, \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_1 = GOTO(I_o, S)$$

$$I_2 = GOTO(I_1, a)$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	$\$$	<i>S</i>
T_o	<u>red</u> - 2		<u>red</u> - 2	T_1
T_1	<u>shift</u> - 2		<u>acc</u>	
T_2	<u>red</u> - 2	<u>red</u> - 2		T_3
...				

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = \{ [S' \rightarrow S \bullet, \$], \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_1 = GOTO(I_o, S)$$

$$\textcolor{blue}{I_2 = GOTO(I_1, a)}$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	$\$$	<i>S</i>
T_o	<u>red</u> - 2		<u>red</u> - 2	T_1
T_1	<u>shift</u> - 2		<u>acc</u>	
T_2	<u>red</u> - 2	<u>red</u> - 2		T_3
...				

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = \{ [S' \rightarrow S \bullet, \$], \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_1 = GOTO(I_o, S)$$

$$I_2 = GOTO(I_1, a)$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	$\$$	<i>S</i>
T_o	<u>red</u> - 2		<u>red</u> - 2	T_1
T_1	<u>shift</u> - 2		<u>acc</u>	
T_2	<u>red</u> - 2	<u>red</u> - 2		T_3
...				

Przykład

Konstrukcja tablicy parsera LR(1) dla gramatyki: (0) $S' \rightarrow S$

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow \epsilon$

$$I_o = \{ [S' \rightarrow \bullet S, \$], \\ [S \rightarrow \bullet SaSb, \$ / a], \\ [S \rightarrow \bullet, \$ / a] \}$$

$$I_1 = \{ [S' \rightarrow S \bullet, \$], \\ [S \rightarrow S \bullet aSb, \$ / a] \}$$

$$I_2 = \{ [S \rightarrow Sa \bullet Sb, \$ / a], \\ [S \rightarrow \bullet SaSb, a / b], \\ [S \rightarrow \bullet, a / b] \}$$

$$I_1 = GOTO(I_o, S)$$

$$I_2 = GOTO(I_1, a)$$

$$I_3 = GOTO(I_2, S)$$

	<i>f</i>		<i>g</i>	
	<i>a</i>	<i>b</i>	$\$$	<i>S</i>
T_o	<u>red</u> - 2		<u>red</u> - 2	T_1
T_1	<u>shift</u> - 2		<u>acc</u>	
T_2	<u>red</u> - 2	<u>red</u> - 2		T_3
...				



AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Parsery SLR(1)

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Podklasty gramatyk LR(1)

$$G_{LR(0)} \subset G_{LR(1)}$$

Wiele gramatyk nie spełnia wymagań LR(0), ale spełnia wymagania LR(1) z nadmiarem. Z drugiej strony, konstrukcja parsera LR(1) jest procesem dość złożonym, a sama tablica LR(1) zajmuje stosunkowo duży obszar pamięci. Stąd pojawiły się gramatyki pośrednie SLR(1) oraz LALR(1). Obejmują one dostatecznie szeroką podklasę języków LR(1), zaś rozmiary tablic parserów SLR(1) i LALR(1) są znacznie mniejsze niż w przypadku tablic kanonicznego LR(1).

$$\begin{aligned} G_{LR(0)} &\subset G_{SLR(1)} \subset G_{LALR(1)} \subset G_{LR(1)} \\ G_{LR(0)} &\neq G_{SLR(1)} \neq G_{LALR(1)} \neq G_{LR(1)} \end{aligned}$$

Nazwa gramatyki: SLR(k)

S L R (k)

Proste
(Simple)

Przeglądanie
wejścia od
lewej strony
do prawej

Odtwarzanie
wywodu
prawostronnego

Wystarcza znajomość
"k" następnych symboli
łańcucha wejściowego i
historii
dotychczasowych
redukacji, aby wyznaczyć
jednoznacznie osnowę i
dokonać jej redukcji

Przedrostki żywotne Sytuacje dopuszczalne

Żywotny przedrostek (viable prefix)

γ - żywotny (aktywny) prefiks gramatyki G

$\Leftrightarrow \gamma$ - prefiks łańcucha $\alpha\beta$

*

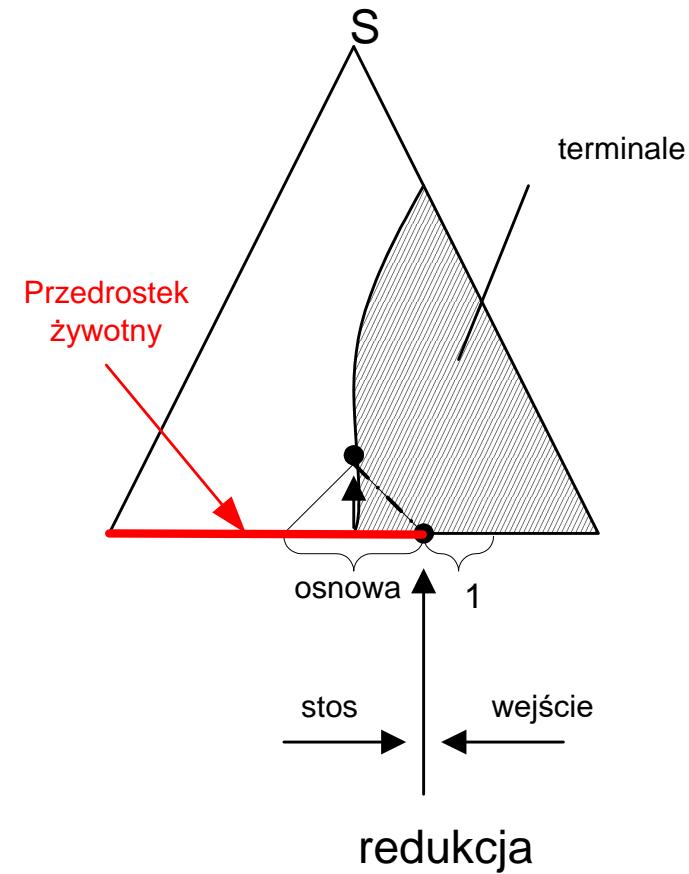
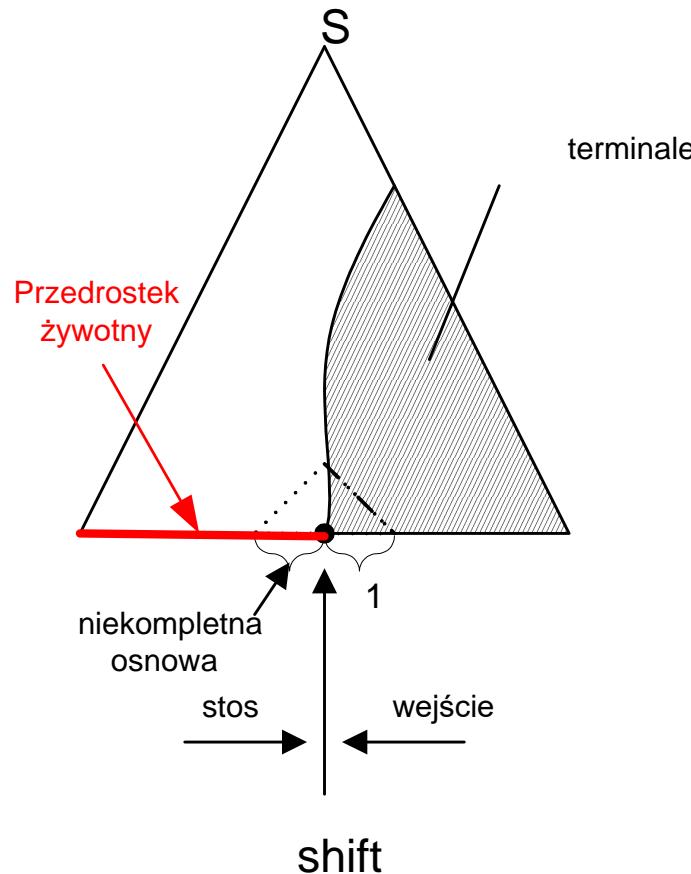
$$S \xrightarrow{R} \alpha A w \xrightarrow{R} \alpha \beta w$$

gdzie: $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$ $w \in \Sigma^*$ $A \in V$

Żywotny przedrostek jest to łańcuch będący przedrostkiem pewnej prawostronnie wyprowadzalnej formy zdaniowej, nie wychodzący poza prawy koniec jej osnowy.

Przedrostki żywotne

Sytuacje dopuszczalne



Przedrostki żywotne Sytuacje dopuszczalne

LR(0) – sytuacja

$[A \rightarrow \beta_1 \bullet \beta_2]$ - jest LR(0)-sytuacją, gdy $(A \rightarrow \beta_1 \beta_2) \in P$

LR(0) - sytuacja dopuszczalna

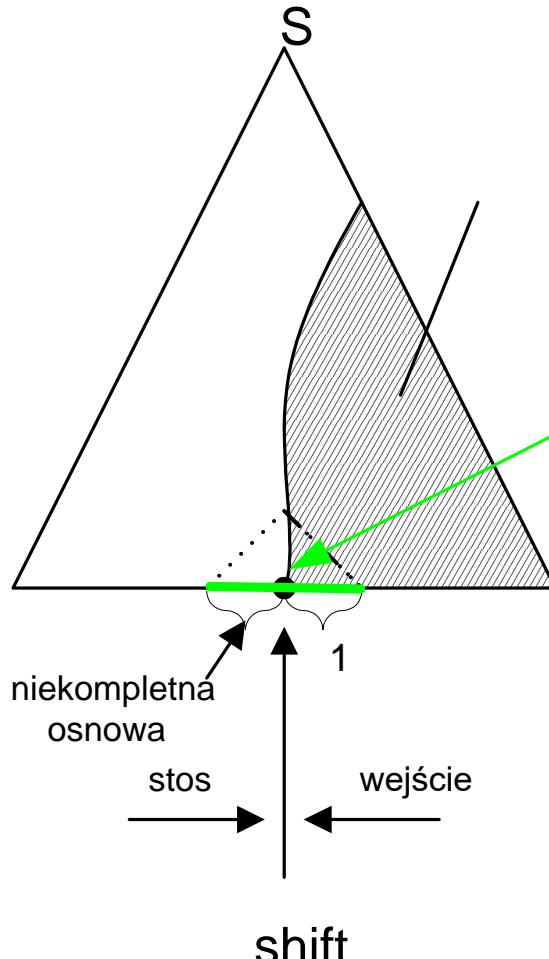
$[A \rightarrow \beta_1 \bullet \beta_2]$ - LR(0)-sytuacja jest sytuacją dopuszczalną dla żywotnego prefiku $\alpha\beta$, wtedy i tylko wtedy gdy

\exists wywód:

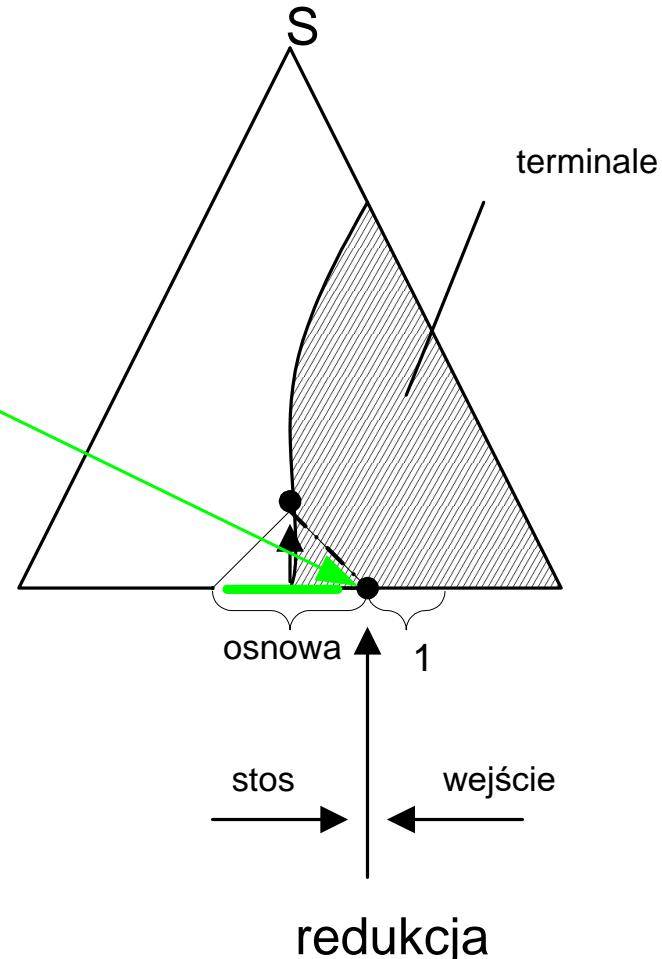
$$S \xrightarrow[R]{*} \alpha A w \xrightarrow[R]{*} \alpha \beta_1 \beta_2 w$$

Przedrostki żywotne

Sytuacje dopuszczalne



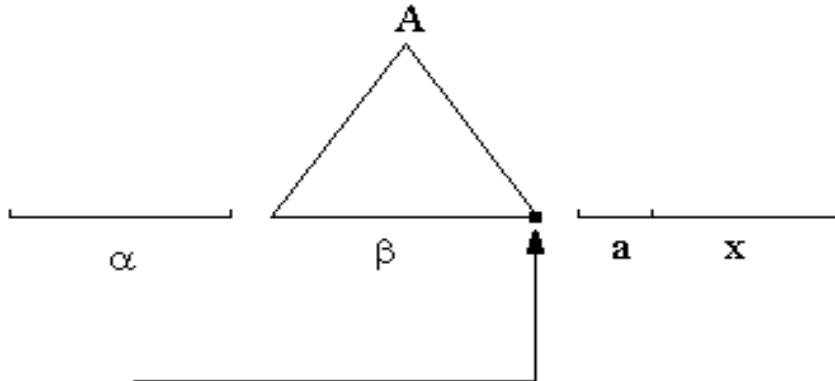
sytuacja dopuszczalna
ilustruje okolice
wierzchołka stosu
a właściwie podaje interpretację
tego obszaru przez parser



redukcja

Przedrostki żywotne

Sytuacje dopuszczalne

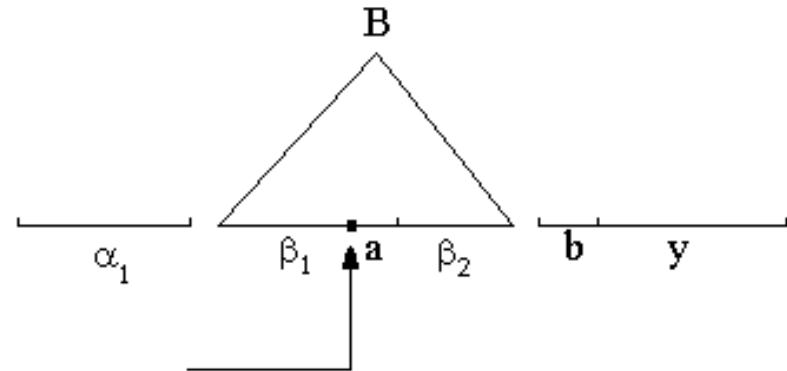


Sytuacja $[A \rightarrow \beta \bullet]$
 dopuszczalna dla żywotnego
 przedrostka $\alpha\beta$

Decyzja: redukcja wg produkcji
 $A \rightarrow \beta$

**Parser SLR wykona tę redukcję,
 gdy $a \in FOLLOW_1(A)$**

Efekt: nowa konfiguracja z
 żywotnym przedrostkiem αA



Sytuacja $[B \rightarrow \beta_1 \bullet a \beta_2]$
 dopuszczalna dla żywotnego przedrostka
 $\alpha_1\beta_1$

Decyzja: przesunięcie (shift) terminala a
 z wejścia na stos

Efekt: nowa konfiguracja opisana
 sytuacją: $[B \rightarrow \beta_1 a \bullet \beta_2]$
 dopuszczalną dla żywotnego przedrostka
 $\alpha_1\beta_1 a$

Przykład – gramatyka jednoznaczna

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

$E \bullet$
 $E + T \bullet$
 $E + T^* F \bullet$
 $E + T^* id \bullet$
 $E + T^* \bullet id$
 $E + T \bullet$ *id $[E \rightarrow E + T \bullet]$ lub $[T \rightarrow T \bullet^* F]$ dla prefiku $E + T$
 $E + F \bullet^* id$
 $E + id \bullet^* id$
 $E + \bullet id^* id$
 $E \bullet + id^* id$
 $T \bullet + id^* id$
 $F \bullet + id^* id$
 $id \bullet + id^* id$
 $\bullet id + id^* id$

Przykład – gramatyka jednoznaczna

stan	f						g		
	\$	+	*	()	<u>id</u>	E	T	F
T ₀				<u>shift-4</u>		<u>shift-5</u>	T ₁	T ₂	T ₃
T ₁	<u>acc</u>	<u>shift-6</u>							
T ₂	<u>red-2</u>	<u>red-2</u>	<u>shift-7</u>		<u>red-2</u>				
T ₃	<u>red-4</u>	<u>red-4</u>	<u>red-4</u>		<u>red-4</u>				
T ₄				<u>shift-4</u>		<u>shift-5</u>	T ₈	T ₂	T ₃
T ₅	<u>red-6</u>	<u>red-6</u>	<u>red-6</u>		<u>red-6</u>				
T ₆				<u>shift-4</u>		<u>shift-5</u>		T ₉	T ₃
T ₇				<u>shift-4</u>		<u>shift-5</u>			T ₁₀
T ₈		<u>shift-6</u>			<u>shift-11</u>				
T ₉	<u>red-1</u>	<u>red-1</u>	<u>shift-7</u>		<u>red-1</u>				
T ₁₀	<u>red-3</u>	<u>red-3</u>	<u>red-3</u>		<u>red-3</u>				
T ₁₁	<u>red-5</u>	<u>red-5</u>	<u>red-5</u>		<u>red-5</u>				

Kanoniczny system zbiorów LR(0)-sytuacji dopuszczalnych

J_0 – Kanoniczny system zbiorów LR(0)-sytuacji
dopuszczalnych

J_0 jest zbiorem wszystkich zbiorów $I(\gamma)$
LR(0)-sytuacji dopuszczalnych;

gdzie: γ - żywotny prefiks w gramatyce G' .

Wyznaczanie kanonicznego systemu zbiorów
LR(0)-sytuacji jest podobne jak w przypadku LR(1).
Podajemy tylko “funkcje” stanowiące podstawę
odpowiednich algorytmów.

Algorytm domykania zbioru sytuacji dopuszczalnych

We: Zbiór I sytuacji dopuszczalnych dla pewnego żywotnego prefiksu (w gramatyce uzupełnionej G')

Wy: Zbiór I będący domknięciem wejściowego zbioru sytuacji dopuszczalnych

Metodę ilustruje funkcja CLOSURE(I);

function CLOSURE(I);

begin

repeat

for każda sytuacja $[A \rightarrow \alpha \bullet B \beta] \in I$ do

for każda produkcja $(B \rightarrow \eta) \in P'$ do

$I := I \cup \{[B \rightarrow \bullet \eta]\};$

until nic nowego nie dodano do I ;

return (I);

end;

Wyznaczanie funkcji GOTO

We: I – zbiór wszystkich sytuacji dopuszczalnych dla prefiksu aktywnego γ , $X \in (V \cup \Sigma)$

Wy: J – zbiór wszystkich sytuacji dopuszczalnych dla prefiksu aktywnego γX

Metodę ilustruje funkcja $\text{GOTO}(I, X)$

function $\text{GOTO}(I, X);$

begin

$J := \emptyset;$

for każda_sytuacja $[A \rightarrow \alpha \bullet X \beta] \in I$ do

$J := J \cup \{ [A \rightarrow \alpha X \bullet \beta] \};$

return $\text{CLOSURE}(J);$

end;

Konstrukcja kanonicznego systemu zbiorów LR(0)-sytuacji dopuszczalnych

We: G' – gramatyka uzupełniona $\langle V', \Sigma, P', S' \rangle$

dla gramatyki $G = \langle V, \Sigma, P, S \rangle \in G_{BK}$

Wy: J_0 – kanoniczny system zbiorów
LR(0)-sytuacji dopuszczalnych dla G .

Metodę ilustruje funkcja ITEMS (G'):

function ITEMS (G');

begin

$J_0 := \{\text{CLOSURE}(\{[S'] \rightarrow^\bullet S\})\};$

repeat

for każdy zbiór $I \in J_0$ do

for każdy $X \in (V \cup \Sigma)$ do

if GOTO(I, X) $\neq \emptyset$ then

$J_0 := J_0 \cup \{\text{GOTO}(I, X)\};$

until nic nowego nie dodano do J_0 ;

return J_0 ;

end;

Gramatyki SLR(1)

$$G = \langle V, \Sigma, P, S \rangle \in G_{BK}$$

J_0 – kanoniczny system zbiorów LR(0)-sytuacji dopuszczalnych dla G

G jest gramatyką SLR(1) \Leftrightarrow

$$(\forall I \in J_0) \left(\begin{array}{l} \left[A \rightarrow \alpha g \beta \right] \in I \\ \left[B \rightarrow \gamma g \delta \right] \in I \end{array} \right) \} \text{ różne LR}(0)\text{-sytuacje!}$$

Zachodzi dokładnie jeden z poniższych warunków:

$$(1) \beta \neq \varepsilon \wedge \delta \neq \varepsilon$$

$$(2) \beta \neq \varepsilon \wedge \delta = \varepsilon \wedge \beta = a \beta_l \wedge a \in \Sigma \wedge \text{FOLLOW}_1(B) \cap \{a\} = \emptyset$$

$$(3) \beta = \varepsilon \wedge \delta \neq \varepsilon \wedge \delta = a \delta_l \wedge a \in \Sigma \wedge \text{FOLLOW}_1(A) \cap \{a\} = \emptyset$$

$$(4) \beta = \varepsilon \wedge \delta = \varepsilon \wedge \text{FOLLOW}_1(A) \cap \text{FOLLOW}_1(B) = \emptyset$$



AGH

Tworzenie tablicy parsera SLR(1)

Algorytm działania parsera SLR(1) jest identyczny jak LR(1). Inny jest sposób tworzenia tablicy parsera.

Konstrukcja tablicy parsera SLR(1)

We: G' - gramatyka uzupełniona dla gramatyki bezkontekstowej G .
 Tablica parsera SLR(1) - funkcje f i g

Metoda:

- (1) Konstruujemy J_0 - kanoniczny system zbiorów LR(0)-sytuacji dopuszczalnych dla G' .
- (2) Numerujemy produkcje gramatyki G' .



Tworzenie tablicy parsera SLR(1)

AGH

(3) for każdy zbiór $I_j \in J_0$ do
begin

utwórz w tablicy parsera stan $T_j \in \mathfrak{T}$ dla analizowanego $I_j \in J_0$;

for każda sytuacja ze zbioru I_j do

begin

(a) if $[A \rightarrow \alpha \bullet a \beta] \in I_j$ and $a \in \Sigma$ and $GOTO(I_j, a) = I_k$
 then $f(T_j, a) := \underline{shift-k}$

(b) if $[A \rightarrow \alpha \bullet] \in I_j$ and $A \neq S'$ and i - numer produkcji $(A \rightarrow \alpha) \in P$
 then for każdy $a \in FOLLOW_1(A)$ do $f(T_j, a) := \underline{red-i};$

(c) if $[S' \rightarrow S \bullet] \in I_j$ then $f(T_j, \$) := \underline{acc};$
 end;

for każdy $A \in V$ do

if $GOTO(I_j, A) \neq \emptyset$ and $GOTO(I_j, A) = I_k$ then $g(T_j, A) := T_k;$
end;



AGH

Tworzenie tablicy parsera SLR(1)

if w jakiejkolwiek pozycji tablicy parsera SLR(1) jest więcej niż jeden zapis then STOP; /* gramatyka nie jest SLR(1) */

(4) for każdy $T_j \in \mathfrak{T}$ do

begin

for każdy $a \in \Sigma \cup \{\$\}$ do

if $f(T_j, a)$ nieokreślone then $f(T_j, a) := \underline{err}$;

for każdy $A \in V$ do

if $g(T_j, A)$ nieokreślone then $g(T_j, A) := \underline{err}$;

end;

(5) Stanem początkowym parsera jest ten stan, który

odpowiada zbiorowi $I \in J_0$, dla którego $[S' \rightarrow \bullet S] \in I$;

Przykład

$$A_0 = \{[E' \rightarrow \bullet E], \\ [E \rightarrow \bullet E + T], \\ [E \rightarrow \bullet T], \\ [T \rightarrow \bullet T * F], \\ [T \rightarrow \bullet F], \\ [F \rightarrow \bullet (E)], \\ [F \rightarrow \bullet \underline{id}]\}$$

$$A_1 = GOTO(A_0, E) = \\ \{[E' \rightarrow E \bullet], \\ [E \rightarrow E \bullet + T]\}$$

$$A_2 = GOTO(A_0, T) = \\ \{[E \rightarrow T \bullet], \\ [T \rightarrow T \bullet * F]\}$$

$$A_3 = GOTO(A_0, F) = \\ \{[T \rightarrow F \bullet]\}$$

$$A_4 = GOTO(A_0, ()) = \\ \{[F \rightarrow (\bullet E)], \\ [E \rightarrow \bullet E + T] \\ [E \rightarrow \bullet T] \\ [T \rightarrow \bullet T * F] \\ [T \rightarrow \bullet F] \\ [F \rightarrow \bullet (E)] \\ [F \rightarrow \bullet \underline{id}]\}$$

$$A_5 = GOTO(A_0, \underline{id}) = \\ \{[F \rightarrow id \bullet]\}$$

- | | |
|-----|-----------------------|
| (0) | $E' \rightarrow E$ |
| (1) | $E \rightarrow E + T$ |
| (2) | $E \rightarrow T$ |
| (3) | $T \rightarrow T * F$ |
| (4) | $T \rightarrow F$ |
| (5) | $F \rightarrow (E)$ |
| (6) | $F \rightarrow id$ |

Przykład c.d.

$$A_6 = GOTO(A_1, +) = \{[E \rightarrow E + \bullet T], [T \rightarrow \bullet T^* F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet \underline{id}]\}$$

$$A_7 = GOTO(A_2, *) = \{[T \rightarrow T^*.F], [F \rightarrow .(E)], [F \rightarrow .\underline{id}]\}$$

$$A_8 = GOTO(A_4, E) = \{[F \rightarrow (E.)], [E \rightarrow E.+T]\}$$

$$A_2 = GOTO(A_4, T)$$

$$A_3 = GOTO(A_4, F)$$

$$A_4 = GOTO(A_4, ())$$

$$A_5 = GOTO(A_4, \underline{id})$$

$$A_9 = GOTO(A_6, T) = \{[E \rightarrow E + T.], [T \rightarrow T.*F]\}$$

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \underline{id}$

Przykład c.d.

$$A_3 = GOTO(A_6, F)$$
$$A_4 = GOTO(A_6, ()$$
$$A_5 = GOTO(A_6, \underline{id})$$
$$A_{10} = GOTO(A_7, F) =$$
$$\{[T \rightarrow T^*F \bullet]\}$$
$$A_4 = GOTO(A_7, ())$$
$$A_5 = GOTO(A_7, \underline{id})$$
$$A_{11} = GOTO(A_8, ()) = \\ \{[F \rightarrow (E) \bullet]\}$$
$$A_6 = GOTO(A_8, +)$$
$$A_7 = GOTO(A_9, *)$$

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^*F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

Przykład – gramatyka jednoznaczna

$$FIRST_1(E) = \{ (, \underline{id} \}$$

$$FIRST_1(T) = \{ (, \underline{id} \}$$

$$FIRST_1(F) = \{ (, \underline{id} \}$$

$$FOLLOW_1(E') = \{ \$ \}$$

$$FOLLOW_1(E) = \{ \$, +,) \}$$

$$FOLLOW_1(T) = \{ \$, +, *,) \}$$

$$FOLLOW_1(F) = \{ \$, +, *,) \}$$

$$A_0 = \{ [E' \rightarrow \bullet E],$$

$$[E \rightarrow \bullet E + T],$$

$$[E \rightarrow \bullet T],$$

$$[T \rightarrow \bullet T^* F],$$

$$[T \rightarrow \bullet F],$$

$$[F \rightarrow \bullet (E)],$$

$$[F \rightarrow \bullet \underline{id}] \}$$

$$A_1 = GOTO(A_0, E) =$$

$$\{ [E' \rightarrow E \bullet],$$

$$[E \rightarrow E \bullet + T] \}$$

$$A_2 = GOTO(A_0, T) =$$

$$\{ [E \rightarrow T \bullet],$$

$$[T \rightarrow T \bullet * F] \}$$

.....

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

stan	f						g		
	\$	+	*	()	id	E	T	F
T_0				<u>shift-4</u>		<u>shift-5</u>	T_1	T_2	T_3
T_1	<u>acc</u>	<u>shift-6</u>							
T_2	<u>red-2</u>	<u>red-2</u>	<u>shift-7</u>		<u>red-2</u>				

.....

Przykład – gramatyka jednoznaczna

stan	f						g		
	\$	+	*	()	<u>id</u>	E	T	F
T ₀				<u>shift-4</u>		<u>shift-5</u>	T ₁	T ₂	T ₃
T ₁	<u>acc</u>	<u>shift-6</u>							
T ₂	<u>red-2</u>	<u>red-2</u>	<u>shift-7</u>		<u>red-2</u>				
T ₃	<u>red-4</u>	<u>red-4</u>	<u>red-4</u>		<u>red-4</u>				
T ₄				<u>shift-4</u>		<u>shift-5</u>	T ₈	T ₂	T ₃
T ₅	<u>red-6</u>	<u>red-6</u>	<u>red-6</u>		<u>red-6</u>				
T ₆				<u>shift-4</u>		<u>shift-5</u>		T ₉	T ₃
T ₇				<u>shift-4</u>		<u>shift-5</u>			T ₁₀
T ₈		<u>shift-6</u>			<u>shift-11</u>				
T ₉	<u>red-1</u>	<u>red-1</u>	<u>shift-7</u>		<u>red-1</u>				
T ₁₀	<u>red-3</u>	<u>red-3</u>	<u>red-3</u>		<u>red-3</u>				
T ₁₁	<u>red-5</u>	<u>red-5</u>	<u>red-5</u>		<u>red-5</u>				

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^*F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

Symulacja działania parsera SLR dla gramatyki jednoznacznej

Stos	Wejście	Wyjście
T_0	<u>id</u> + <u>id</u> \$	ϵ
$T_0\text{id}T_5$	+ <u>id</u> \$	ϵ
T_0FT_3	+ <u>id</u> \$	6
T_0TT_2	+ <u>id</u> \$	64
T_0ET_1	+ <u>id</u> \$	642
$T_0ET_1+T_6$	<u>id</u> \$	642
$T_0ET_1+T_6\text{id}T_5$	\$	642
$T_0ET_1+T_6FT_3$	\$	6426
$T_0ET_1+T_6TT_9$	\$	64264
T_0ET_1	\$	642641
akceptacja		

Przykład gramatyki niejednoznacznej

Rozważana gramatyka:

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + E$
- (2) $E \rightarrow E * E$
- (3) $E \rightarrow (E)$
- (4) $E \rightarrow \underline{id}$

jest uproszczoną, a zarazem niejednoznaczną wersją gramatyki SLR(1) z poprzedniego przykładu:

- (0) $E' \rightarrow E$
 - (1) $E \rightarrow E + T$
 - (2) $E \rightarrow E * F$
 - (3) $F \rightarrow (E)$
 - (4) $F \rightarrow \underline{id}$
- (*) (2) $E \rightarrow T$
(4) $T \rightarrow F$

(*) W rozważanej gramatyce nie ma produkcji łańcuchowych, więc gdyby udało się skonstruować dla niej parser SLR(1) to rozbiór syntaktyczny byłby jeszcze szybszy, a także rozmiar tablicy parsera byłby mniejszy.

Przykład gramatyki niejednoznacznej

Sposób postępowania:

- (1) Konstruujemy system zbiorów LR(0)-sytuacji
- (2) Próbuając zbudować tablicę dla parsera SLR(1)
znajdujemy konflikty
- (3) Staramy się usunąć konflikty wykorzystując dodatkowe wiadomości i wymagania związane z językiem generowanym przez rozważaną gramatykę niejednoznaczną.

Przykład gramatyki niejednoznacznej

$A_0:$

$$E' \rightarrow \bullet E$$

$$E \rightarrow \bullet E + E$$

$$E \rightarrow \bullet E * E$$

$$E \rightarrow \bullet (E)$$

$$E \rightarrow \bullet \underline{id}$$

$A_1:$

$$E' \rightarrow E \bullet$$

$$E \rightarrow E \bullet + E$$

$$E \rightarrow E \bullet * E$$

$A_2:$

$$E \rightarrow (\bullet E)$$

$$E \rightarrow \bullet E + E$$

$$E \rightarrow \bullet E * E$$

$$E \rightarrow \bullet (E)$$

$$E \rightarrow \bullet \underline{id}$$

$A_3:$ $E \rightarrow \underline{id} \bullet$

$A_4:$

$$E \rightarrow E + \bullet E$$

$$E \rightarrow \bullet E + E$$

$$E \rightarrow \bullet E * E$$

$$E \rightarrow \bullet (E)$$

$$E \rightarrow \bullet \underline{id}$$

$A_5:$

$$E \rightarrow E * \bullet E$$

$$E \rightarrow \bullet E + E$$

$$E \rightarrow \bullet E * E$$

$$E \rightarrow \bullet (E)$$

$$E \rightarrow \bullet \underline{id}$$

$A_6:$

$$E \rightarrow (E \bullet)$$

$$E \rightarrow E \bullet + E$$

$$E \rightarrow E \bullet * E$$

$A_7:$

$$\textcolor{blue}{E \rightarrow E + E \bullet}$$

$$\textcolor{blue}{E \rightarrow E \bullet + E}$$

$$\textcolor{blue}{E \rightarrow E \bullet * E}$$

$A_8:$

$$\textcolor{blue}{E \rightarrow E * E \bullet}$$

$$\textcolor{blue}{E \rightarrow E \bullet + E}$$

$$\textcolor{blue}{E \rightarrow E \bullet * E}$$

$A_9:$

$$E \rightarrow (E) \bullet$$

$\boxed{\text{FOLLOW}_1(E) = \{\$,), +, *\}}$

Przykład gramatyki niejednoznacznej

$$\begin{aligned}A_7: \quad & E \rightarrow E + E^\bullet \\& E \rightarrow E^\bullet + E \\& E \rightarrow E^\bullet * E\end{aligned}$$

Konflikty dla A_7 :

a) ponieważ: $\{ +, * \} \subset FOLLOW_I(E)$ więc:

$$\begin{aligned}f(T_7, +) &= \underline{\text{red}} \ 1 \\f(T_7, *) &= \underline{\text{red}} \ 1\end{aligned}$$

b) ponieważ: $\{ +, * \} \subset \Sigma$ więc:

$$\begin{aligned}f(T_7, +) &= \underline{\text{shift}} \\f(T_7, *) &= \underline{\text{shift}}\end{aligned}$$

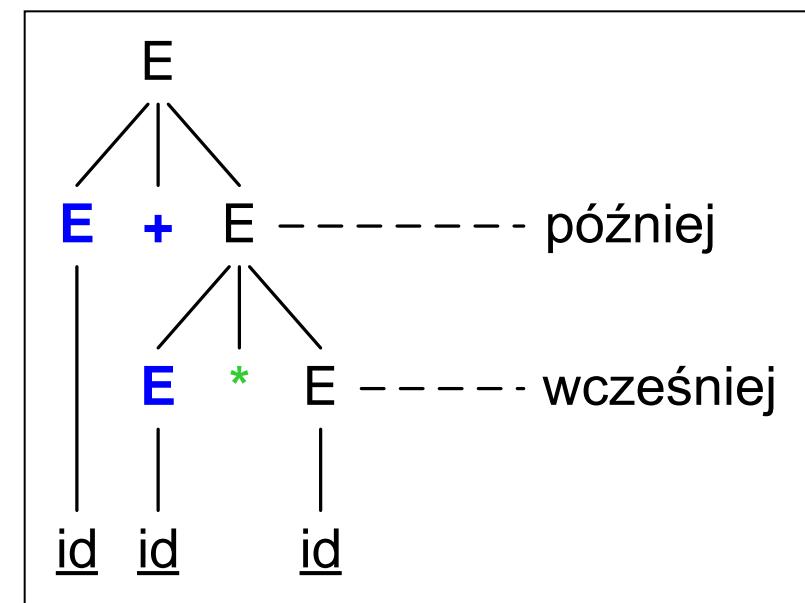
Konflikty:

$$\begin{array}{ll}f(T_7, +) = \underline{\text{red}} \ 2 & f(T_7, +) = \underline{\text{shift}} \\f(T_7, *) = \underline{\text{red}} \ 2 & f(T_7, *) = \underline{\text{shift}}\end{array}$$

Przykład – gramatyka niejednoznaczna, usuwanie konfliktów

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E+E$
- (2) $E \rightarrow E^*E$
- (3) $E \rightarrow (E)$
- (4) $E \rightarrow id$

$E \bullet$
 $E+E \bullet$
 $E+E^*E \bullet$
 $E+E^*id \bullet$
 $E+E^*\bullet id$
 $E+E \bullet^* id$ [$E \rightarrow E+E \bullet$]; [$E \rightarrow E \bullet + E$] lub [$E \rightarrow E \bullet^* E$]
 $E+id \bullet^* id$
 $E+\bullet id^* id$
 $E \bullet + id^* id$
 $id \bullet + id^* id$
 $\bullet id + id^* id$



Przykład gramatyki niejednoznacznej

Rozważamy ciąg wejściowy: *id*+*id***id*

Ponieważ “*” ma wyższy priorytet niż “+”, więc “*” powinna być wcześniej redukowana niż “+”

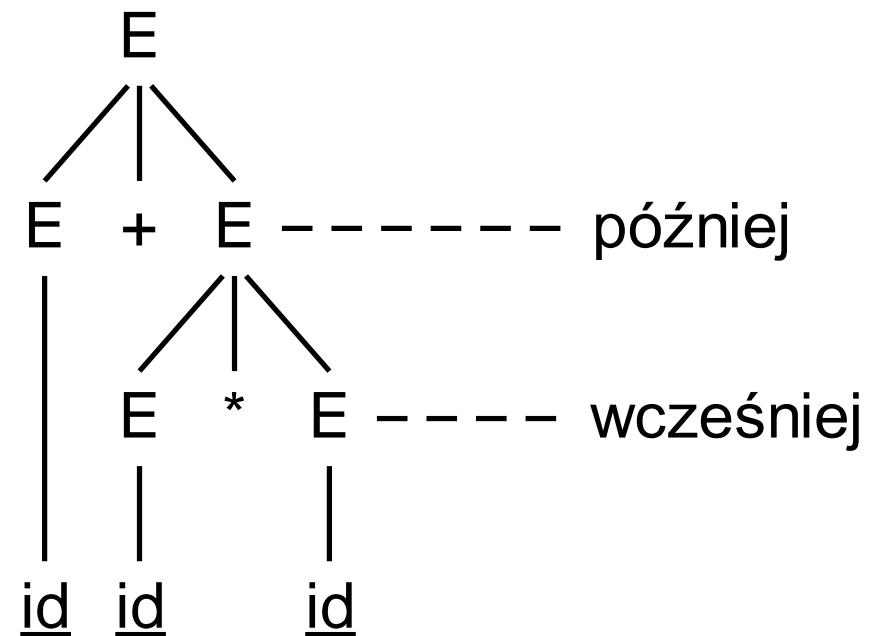
Stos:

$T_0ET_1 + T_4ET_7$

we:

* *id* \$

DECYZJA: $f(T_7, *) = \underline{\text{shift}}\ 5$



Przykład gramatyki niejednoznacznej

Rozważamy ciąg wejściowy: *id*+*id*+*id*

Ponieważ “+” jest lewostronnie łączny , więc najpierw powinna nastąpić redukcja lewego “+”

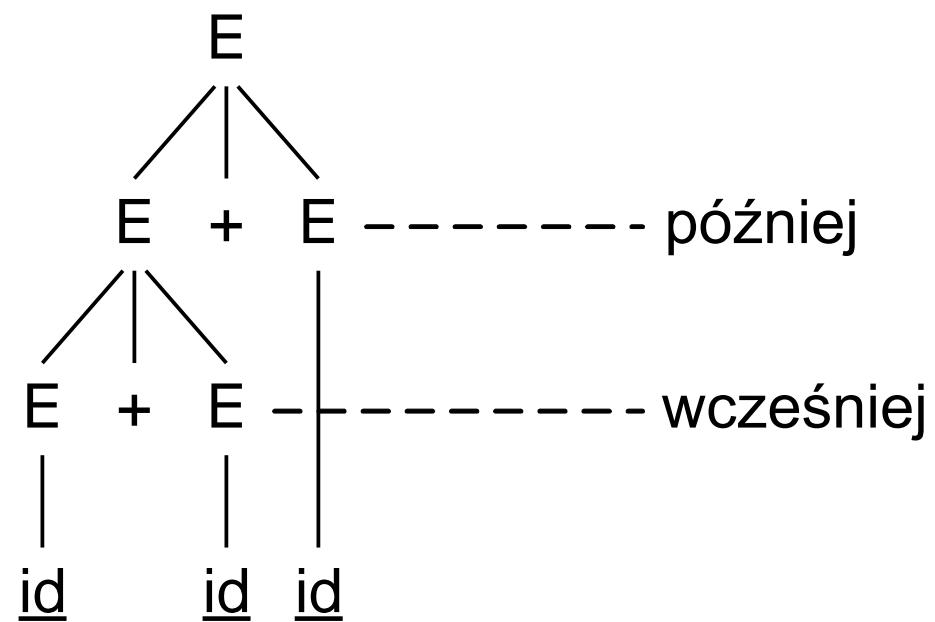
Stos:

$T_0ET_1 + T_4ET_7$

we:

+ *id* \$

DECYZJA: $f(T_7, +) = \text{red } 1$



Przykład – gramatyka niejednoznaczna, usuwanie konfliktów

stan	<u>id</u>	+	*	()	\$	E
T ₀	<u>shift</u> 3			<u>shift</u> 2			T ₁
T ₁		<u>shift</u> 4	<u>shift</u> 5			<u>acc</u>	
T ₂	<u>shift</u> 3			<u>shift</u> 2			T ₆
T ₃		<u>red</u> 4	<u>red</u> 4		<u>red</u> 4	<u>red</u> 4	
T ₄	<u>shift</u> 3			<u>shift</u> 2			T ₇
T ₅	<u>shift</u> 3			<u>shift</u> 2			T ₈
T ₆		<u>shift</u> 4	<u>shift</u> 5		<u>shift</u> 9		
T ₇		<u>shift</u> 4 <u>red</u> 1	<u>shift</u> 5 <u>red</u> 1		<u>red</u> 1	<u>red</u> 1	
T ₈		<u>shift</u> 4 <u>red</u> 2	<u>shift</u> 5 <u>red</u> 2		<u>red</u> 2	<u>red</u> 2	
T ₉			<u>red</u> 3	<u>red</u> 3	<u>red</u> 3	<u>red</u> 3	

Symulacja działania parsera SLR dla gramatyki niejednoznacznej

Stos	Wejście	Wyjście
T_0	<u>id</u> + <u>id</u> \$	ϵ
$T_0\underline{id}T_3$	+ <u>id</u> \$	ϵ
T_0ET_1	+ <u>id</u> \$	4
$T_0ET_1+T_4$	<u>id</u> \$	4
$T_0ET_1+T_4\underline{id}T_3$	\$	4
$T_0ET_1+T_4ET_7$	\$	44
T_0ET_1	\$	441
akceptacja		

Przypomnienie: symulacja działania parsera LL dla gramatyki jednoznacznej po usunięciu lewostronnej rekurencji

Stos	Wejście	Wyjście
E	<u>id</u> + <u>id</u> \$	ϵ
E'T	<u>id</u> + <u>id</u> \$	1
E'T'F	<u>id</u> + <u>id</u> \$	14
E'T' <u>id</u>	<u>id</u> + <u>id</u> \$	148
E'T'	+ <u>id</u> \$	148
E'	+ <u>id</u> \$	1486
E'T+	+ <u>id</u> \$	14862
E'T	<u>id</u> \$	14862
E'T'F	<u>id</u> \$	148624
E'T' <u>id</u>	<u>id</u> \$	1486248
E'T'	\$	1486248
E'	\$	14862486
ϵ	\$	148624863

akceptacja

Porównanie działania parserów LL i LR

... dla wejścia id+id i odpowiednich gramatyk

Rodzaj parsera	Liczba kroków	Długość wyjścia
LL dla gramatyki jednoznacznej po usunięciu lewostronnej rekurencji	12	9
SLR dla gramatyki jednoznacznej	9	6
SLR dla gramatyki niejednoznacznej	6	3



AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Parsery LALR(1)

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Gramatyki LALR(k) (Look Ahead LR(k))

Duża klasa gramatyk mających praktyczne zastosowanie nie należy do SLR(1), ale dla tych gramatyk daje się skonstruować parser o rozmiarze tablicy sterującej (funkcja "f" i "g") identycznym z SLR(1), tyle że przy pomocy trochę bardziej skomplikowanego algorytmu. Klasa ta nazywa się LALR(1) i z praktycznego punktu widzenia jest najważniejszą podklasą LR(1), gdyż:

- 1) Jest dostatecznie szeroka, żeby objąć znaczną większość języków programowania.
- 2) Rozmiar tablicy sterującej parsera jest jeszcze do przyjęcia (w odróżnieniu od kanonicznego LR(1)).

Jądro zbioru LR(1)-sytuacji

CORE – jądro zbioru LR(1) – sytuacji

A - zbiór LR(1) – sytuacji

$$CORE(A) = \{ [A \rightarrow \alpha \bullet \beta] / [A \rightarrow \alpha \bullet \beta, u] \in A \}$$

Twierdzenie

J_0 – kanoniczny system zbiorów LR(0) – sytuacji w G

J_1 – kanoniczny system zbiorów LR(1) – sytuacji w G

$$J_0 = \{ CORE(A) : A \in J_1 \}$$

CORE(GOTO(...))

Twierdzenie

$$\begin{aligned} \text{CORE}(\text{GOTO}_1(A, x)) &= \text{GOTO}_0(\text{CORE}(A), x) \\ x \in (V \cup \Sigma) \end{aligned}$$

gdzie:

GOTO_1 – funkcja GOTO dla zbioru LR(1)-sytuacji

GOTO_0 – funkcja GOTO dla zbioru LR(0)-sytuacji

e – relacja równości jąder

$$e \subset J_1 \times J_1 : A_1 e A_2 \stackrel{!}{\Leftrightarrow} \text{CORE}(A_1) = \text{CORE}(A_2)$$

Gramatyka LALR(1)

$$G = \langle V, \Sigma, P, S \rangle \in G_{BK}$$

J_1 – kanoniczny system zbiorów LR(1) – sytuacji dla G

$$J = \{ B([A]_e) : A \in J_1 \};$$

gdzie:

$$B([A]_e) = \bigcup_{C \in [A]_e} (C \in J_1)$$

G – gramatyka LALR(1) $\Leftrightarrow (\forall B \in J) (B - \text{zgodny})$

Przykład (łączenie zbiorów LR(1)-sytuacji)

$J_1 = \{ A_1, A_2, A_3, A_4, A_5, A_6 \}$

$e = \{ (A_1, A_1), (A_2, A_2), (A_3, A_3), (A_4, A_4), (A_2, A_3),$
 $(A_3, A_2), (A_2, A_4), (A_4, A_2), (A_3, A_4), (A_4, A_3),$
 $(A_5, A_5), (A_6, A_5), (A_5, A_6), (A_6, A_6) \}$

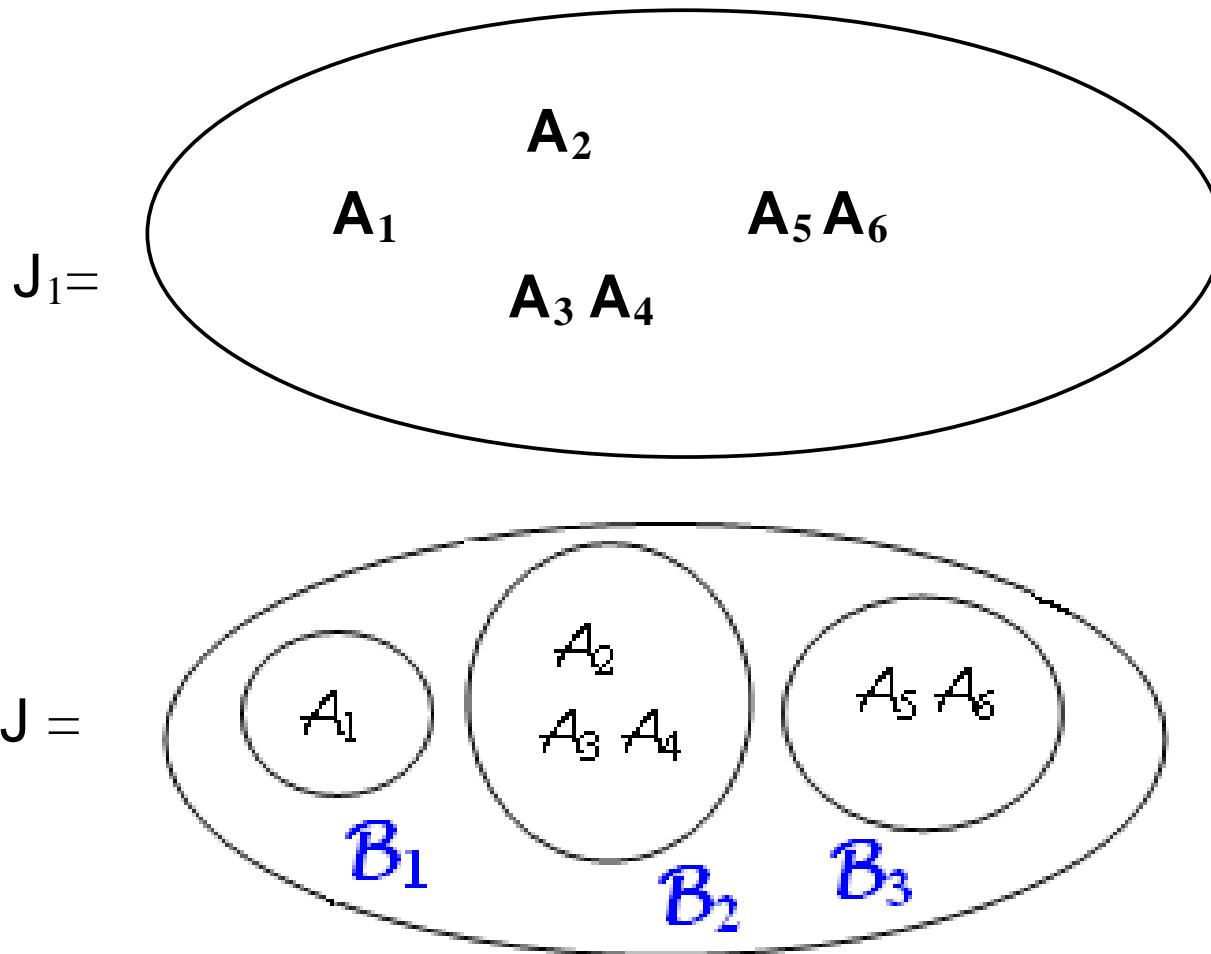
$\mathbf{B}_1 = A_1$

$\mathbf{B}_2 = A_2 \cup A_3 \cup A_4$

$\mathbf{B}_3 = A_5 \cup A_6$

$J = \{ B_1, B_2, B_3 \}$

Przykład c.d.



Konstrukcja tablicy parsera LALR(1)

WEJŚCIE: $G = \langle V, T, P, S \rangle \in G_{BK}$

WYJŚCIE: Tablica LALR(1)

ALGORYTM:

1. Wyznaczamy J_1 – kanoniczny system zbiorów LR(1)-sytuacji dopuszczalnych
2. Konstruujemy zbiór klas abstrakcji dla relacji e :

$$J := \{ B([A]_e) : A \in J_1 \};$$

$$/* J = \{ B_0, B_1, \dots, B_n \}$$

$$\text{gdzie } B_i = A_{(i1)} \cup A_{(i2)} \cup \dots \cup A_{(i m_i)} \\ i = 0, 1, \dots, n$$

$$[A_{(i1)}]_e = [A_{(i2)}]_e = \dots = [A_{(i m_i)}]_e */$$

3. Określamy funkcje f i g dla $J = \{ B_0, B_1, \dots, B_n \}$ identycznie, jak to miało miejsce w przypadku kanonicznego LR(1);

Przykład

$$G' = \langle V' = \{ S', S, L, R \}, T = \{ =, *, \underline{id} \}, P', S' \rangle$$
$$P' = \{$$

- (0) $S' \rightarrow S$
- (1) $S \rightarrow L = R$
- (2) $S \rightarrow R$
- (3) $L \rightarrow *R$
- (4) $L \rightarrow \underline{id}$
- (5) $R \rightarrow L$

$$\}$$
$$FIRST_1(S) = \{ *, \underline{id} \}$$
$$FIRST_1(L) = \{ *, \underline{id} \}$$
$$FIRST_1(R) = \{ *, \underline{id} \}$$
$$FOLLOW_1(S) = \{ \$ \}$$
$$FOLLOW_1(L) = \{ \$, = \}$$
$$FOLLOW_1(R) = \{ \$, = \}$$

Przykład

Sprawdzenie czy poniższa gramatyka G' jest SLR(1)?

Próbujemy skonstruować J_0 – kanoniczny system LR(0) – sytuacji

$$A_0 = \{ [S' \rightarrow \bullet S], \\ [S \rightarrow \bullet L = R], \\ [S \rightarrow \bullet R], \\ [L \rightarrow \bullet *R], \\ [L \rightarrow \bullet \underline{id}], \\ [R \rightarrow \bullet L] \}$$

$$A_1 = GOTO(A_0, S) = \{ [S' \rightarrow S \bullet] \}$$

$$A_2 = GOTO(A_0, L) = \{ [S \rightarrow L \bullet = R], \\ [R \rightarrow L \bullet] \}$$

- 1) $S' \rightarrow S$
- 2) $S \rightarrow L = R$
- 3) $S \rightarrow R$
- 4) $L \rightarrow *R$
- 5) $L \rightarrow \underline{id}$
- 6) $R \rightarrow L$

$$FOLLOW_1(S) = \{ \$ \}$$

$$FOLLOW_1(L) = \{ \$, = \}$$

$$FOLLOW_1(R) = \{ \$, = \}$$

Przykład

$$A_2 = GOTO(A_0, L) = \{ [S \rightarrow L^\bullet = R] , \\ [R \rightarrow L^\bullet] \\ \}$$

Ponieważ:

$$FOLLOW_1(R) = \{ \$, = \}$$

Więc:

$$\left. \begin{array}{l} f(T_2, =) = \underline{\text{red-6}} \\ f(T_2, =) = \underline{\text{shift}} \end{array} \right\} \text{konflikt!}$$

gramatyka nie jest SLR(1)!

Przykład

Konstruujemy J_1 – kanoniczny system zbiorów
LR(1)–sytuacji.

$$A_0 = \{ [S' \rightarrow \bullet S, \$], [S \rightarrow \bullet L = R, \$], [S \rightarrow \bullet R, \$], [L \rightarrow \bullet *R, \$/ =], [L \rightarrow \bullet \underline{id}, \$/ =], [R \rightarrow \bullet L, \$] \}$$

$$A_1 = \{ [S' \rightarrow S \bullet, \$] \}$$

$$A_2 = \{ [S \rightarrow L \bullet = R, \$] \} \\ [R \rightarrow L \bullet, \$] \}$$

- | | |
|----|--------------------------------|
| 1) | $S' \rightarrow S$ |
| 2) | $S \rightarrow L = R$ |
| 3) | $S \rightarrow R$ |
| 4) | $L \rightarrow *R$ |
| 5) | $L \rightarrow \underline{id}$ |
| 6) | $R \rightarrow L$ |

$$A_1 = GOTO(A_0, S)$$

$$A_2 = GOTO(A_0, L)$$

Przykład

$$A_3 = \{ [S \rightarrow R^\bullet, \$] \}$$

$$A_3 = GOTO(A_0, R)$$

$$A_4 = \{ [L \rightarrow *^\bullet R, \$/=],$$

$$A_4 = GOTO(A_0, *)$$

$$[R \rightarrow \bullet L, \$/=],$$

$$[L \rightarrow \bullet * R, \$/=],$$

$$[L \rightarrow \bullet \underline{id}, \$/=]\}$$

$$A_5 = \{ [L \rightarrow \underline{id}^\bullet, \$/=] \}$$

$$A_5 = GOTO(A_0, \underline{id})$$

$$A_6 = \{ [S \rightarrow L =^\bullet R, \$],$$

$$A_6 = GOTO(A_2, =)$$

$$[R \rightarrow \bullet L, \$],$$

$$[L \rightarrow \bullet * R, \$],$$

$$[L \rightarrow \bullet \underline{id}, \$] \}$$

- 1) $S' \rightarrow S$
- 2) $S \rightarrow L = R$
- 3) $S \rightarrow R$
- 4) $L \rightarrow * R$
- 5) $L \rightarrow \underline{id}$
- 6) $R \rightarrow L$

Przykład

$$A_7 = \{ [L \rightarrow *R^\bullet, \$/=] \}$$

$$A_7 = GOTO(A_4, R)$$

$$A_8 = \{ [R \rightarrow L^\bullet, \$/=] \}$$

$$A_8 = GOTO(A_4, L)$$

$$A_9 = \{ [S \rightarrow L=R^\bullet, \$] \}$$

$$A_9 = GOTO(A_6, R)$$

$$A_{10} = \{ [L \rightarrow *R^\bullet, \$],$$

$$A_{10} = GOTO(A_6, *)$$

$$[R \rightarrow L^\bullet, \$],$$

$$[L \rightarrow *R^\bullet, \$],$$

$$[L \rightarrow id^\bullet, \$] \}$$

- 1) $S' \rightarrow S$
- 2) $S \rightarrow L = R$
- 3) $S \rightarrow R$
- 4) $L \rightarrow *R$
- 5) $L \rightarrow id$
- 6) $R \rightarrow L$

Przykład

$$A_{11} = \{[L \rightarrow \underline{id} \bullet, \$]\}$$

$$A_{12} = \{[R \rightarrow L \bullet, \$]\}$$

$$A_{13} = \{[L \rightarrow *R \bullet, \$]\}$$

$$A_{11} = GOTO(A_6, \underline{id})$$

$$A_{12} = GOTO(A_6, L)$$

$$A_{13} = GOTO(A_{10}, R)$$

$$A_{12} = GOTO(A_{10}, L)$$

$$A_{10} = GOTO(A_{10}, *)$$

$$A_{11} = GOTO(A_{10}, \underline{id})$$

- 1) $S' \rightarrow S$
- 2) $S \rightarrow L = R$
- 3) $S \rightarrow R$
- 4) $L \rightarrow *R$
- 5) $L \rightarrow \underline{id}$
- 6) $R \rightarrow L$

Przykład

$B_0 = A_0$

$B_1 = A_1$

$B_2 = A_2$

$B_3 = A_3$

$B_4 = A_4 \cup A_{10}$

$B_5 = A_5 \cup A_{11}$

$B_6 = A_6$

$B_7 = A_7 \cup A_{13}$

$B_8 = A_8 \cup A_{12}$

$B_9 = A_9$

$B_1 = \text{GOTO}(B_0, S)$

$B_2 = \text{GOTO}(B_0, L)$

$B_3 = \text{GOTO}(B_0, R)$

$B_4 = \text{GOTO}(B_0, *)$

$B_5 = \text{GOTO}(B_0, \underline{\text{id}})$

$B_6 = \text{GOTO}(B_2, =)$

$B_7 = \text{GOTO}(B_4, R)$

$B_8 = \text{GOTO}(B_4, L)$

$B_4 = \text{GOTO}(B_4, *)$

$B_5 = \text{GOTO}(B_4, \underline{\text{id}})$

$B_9 = \text{GOTO}(B_6, R)$

$B_4 = \text{GOTO}(B_6, *)$

$B_5 = \text{GOTO}(B_6, \underline{\text{id}})$

$B_8 = \text{GOTO}(B_6, L)$

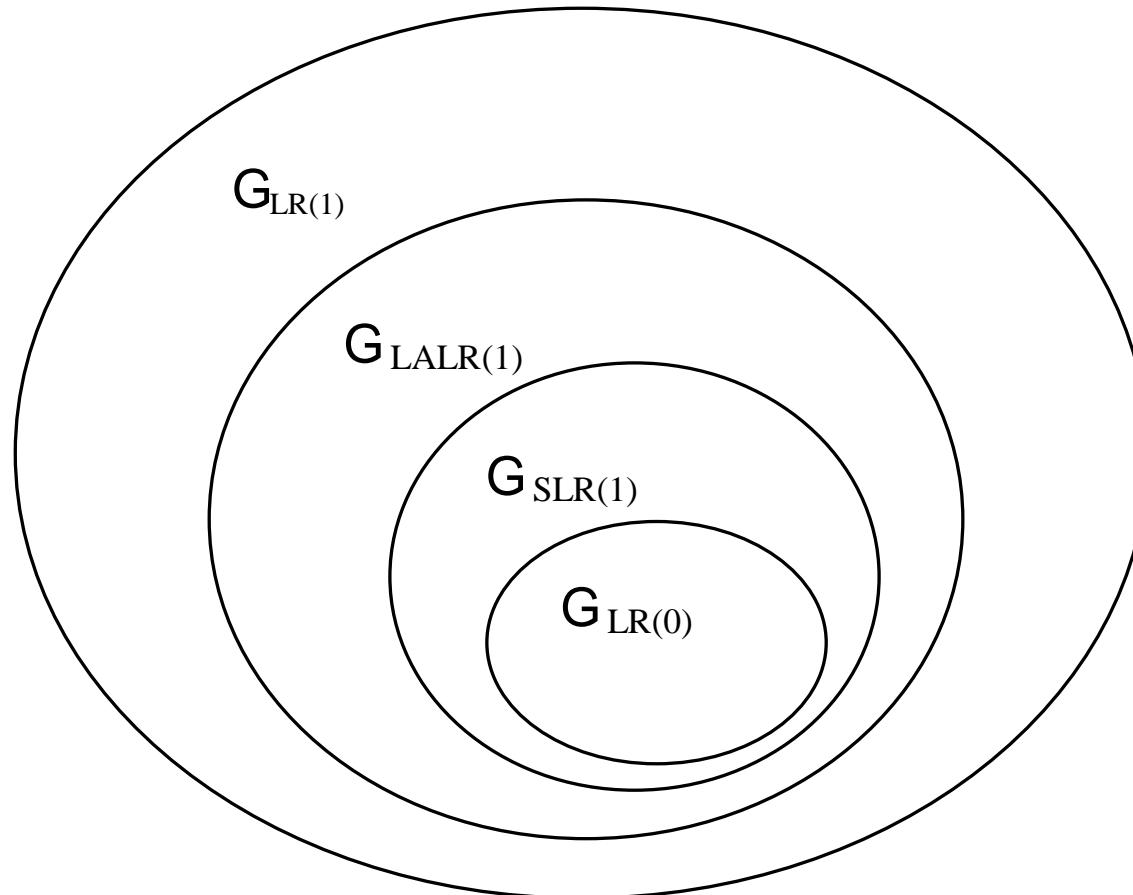
Przykład

TABLICA LALR(1)

stan	f					g		
	\$	=	*	<u>id</u>	S	L	R	
T ₀			<u>shift-4</u>	<u>shift-5</u>	T ₁	T ₂	T ₃	
T ₁	<u>acc</u>							
T ₂	<u>red-5</u>	<u>shift-6</u>						
T ₃	<u>red-2</u>							
T ₄			<u>shift-4</u>	<u>shift-5</u>		T ₈	T ₇	
T ₅	<u>red-4</u>	<u>red-4</u>						
T ₆			<u>shift-4</u>	<u>shift-5</u>		T ₈	T ₉	
T ₇	<u>red-3</u>	<u>red-3</u>						
T ₈	<u>red-5</u>	<u>red-5</u>						
T ₉	<u>red-1</u>							

(Kanoniczny LR(1) ma o 4 stany więcej)

Zależności pomiędzy klasami gramatyk



$$G_{LR(0)} \subset G_{SLR(1)} \subset G_{LALR(1)} \subset G_{LR(1)}$$

$$G_{LR(0)} \neq G_{SLR(1)} \neq G_{LALR(1)} \neq G_{LR(1)}$$

Każda gramatyka LR(0)
jest gramatyką SLR(1).
Każda gramatyka SLR(1)
jest gramatyką LALR(1).
Każda gramatyka LALR(1)
jest gramatyką LR(1).

Języki bezkontekstowe zdeterminowane a języki LR

Język bezkontekstowy nazywamy zdeterminowanym, gdy jest on akceptowany przez deterministyczny automat ze stosem.

$$L_{BK\ DET} = L_{LR(1)}$$

Każdy język bezkontekstowy zdeterminowany ma swoją gramatykę LR(1).

Każdy język bezkontekstowy zdeterminowany posiadający własność przedrostkową ma swoją gramatykę LR(0).

Języki LL a języki LR

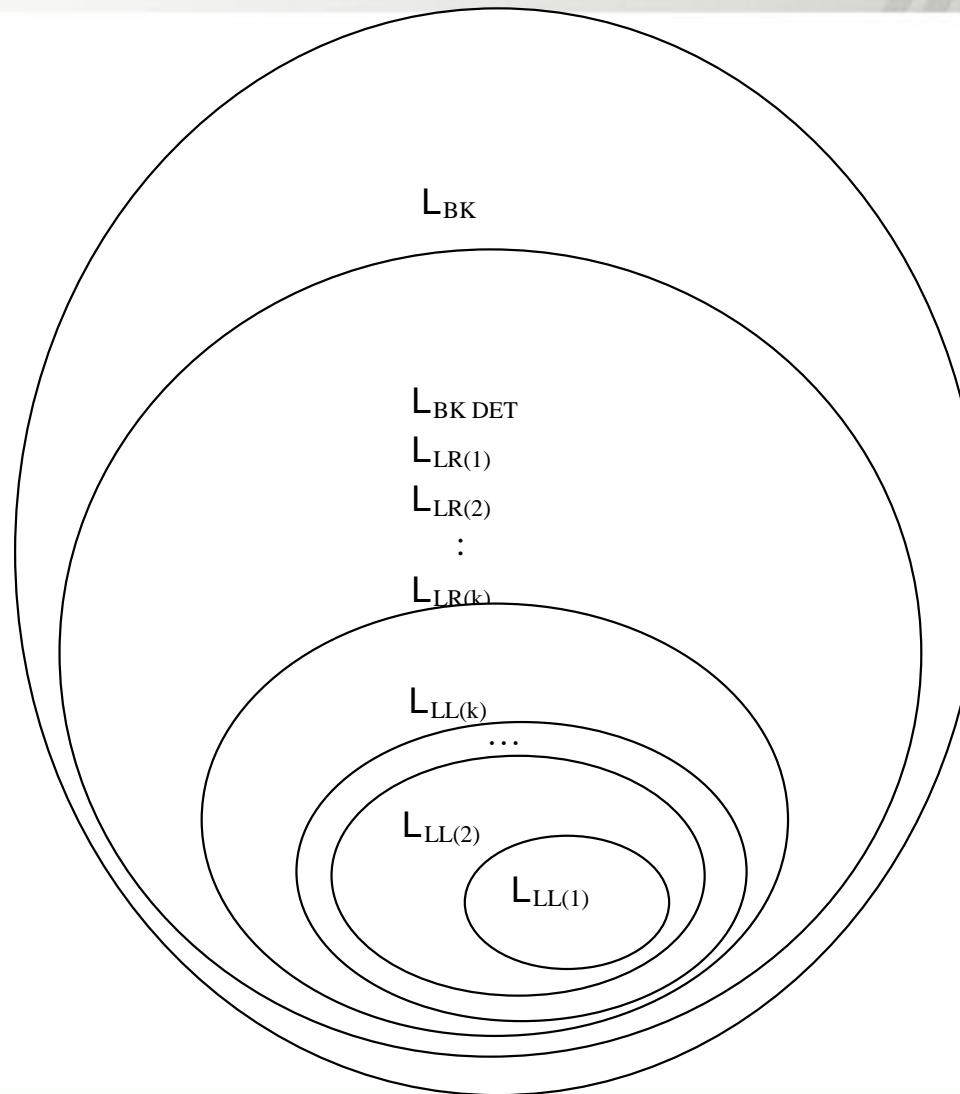
$$L_{LL(K)} \subset L_{LR(K)}$$

$$L_{LL(K)} \neq L_{LR(K)}$$

Każda gramatyka LL(k) jest gramatyką LR(k).
Istnieją języki LR(k), które nie są generowane
przez żadną gramatykę LL(k).

Języki LL(0) są jednoelementowe.

Podkasy języków bezkontekstowych





AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Gramatyki z pierwszeństwem operatorów Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki



Gramatyka operatorowa

Definicja:

$G = \langle V, \Sigma, P, S \rangle \in \mathcal{G}_{BK}$ jest gramatyką operatorową \Leftrightarrow

- (i) G – jest gramatyką prawidłową (bez ε -produkcyj)

$$\forall (A \rightarrow x_1 \dots x_n) \in P :$$

- (ii) $x_i \in V \Rightarrow x_{i+1} \in \Sigma$

$$x_{i+1} \in V \Rightarrow x_i \in \Sigma$$

(czyli dwa symbole nieterminalne nigdy nie sąsiadują ze sobą w prawych stronach produkcji). W analizie gramatyk operatorowych przyjmuje się termin „operator” dla określenia terminala i termin „operand” dla określenia nieterminala.



Relacje pierwszeństwa

Dla terminali określa się relacje pierwszeństwa

$$\langle \bullet, \sqsubseteq, \bullet \rangle \subset (\Sigma \cup \{\$\}) \times (\Sigma \cup \{\$\})$$
 w następujący sposób:

$$(1) a \sqsubseteq b \text{ gdy } (A \rightarrow \alpha a \gamma b \beta) \in P \wedge \gamma \in V \cup \{\varepsilon\}, \quad a, b \in \Sigma$$

$$(2) a < \bullet b \text{ gdy } (A \rightarrow \alpha a B \beta) \in P \wedge B \stackrel{+}{\Rightarrow} \gamma b \delta \wedge \gamma \in V \cup \{\varepsilon\}$$

$$(3) a \bullet > b \text{ gdy } (A \rightarrow \alpha B b \beta) \in P \wedge B \stackrel{+}{\Rightarrow} \delta a \gamma \wedge \gamma \in V \cup \{\varepsilon\}$$

$$(4) \$ < \bullet a \text{ gdy } S \stackrel{+}{\Rightarrow} \gamma a \alpha \wedge \gamma \in V \cup \{\varepsilon\}$$

$$(5) a \bullet > \$ \text{ gdy } S \stackrel{+}{\Rightarrow} \alpha a \gamma \wedge \gamma \in V \cup \{\varepsilon\}$$

We wszystkich powyższych określeniach $a, b \in \Sigma$.



Gramatyka z pierwszeństwem operatorów

Definicja:

Gramatyka $G = \langle V, \Sigma, P, S \rangle \in \mathcal{G}_{BK}$ jest gramatyką z pierwszeństwem operatorów \Leftrightarrow

- (i) G – jest gramatyką operatorową,
- (ii) dla każdej pary „operatorów” (terminali) zachodzi co najwyżej jedna z relacji pierwszeństwa: $\langle \bullet, \sqsubseteq, \bullet \rangle$



Przykład

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \underline{id}$$

	+	*	()	<u>id</u>	\$
+	•>	<•	<•	•>	<•	•>
*	•>	•>	<•	•>	<•	•>
(<•	<•	<•	■	<•	
)	•>	•>		•>		•>
<u>id</u>	•>	•>		•>		•>
\$	<•	<•	<•		<•	

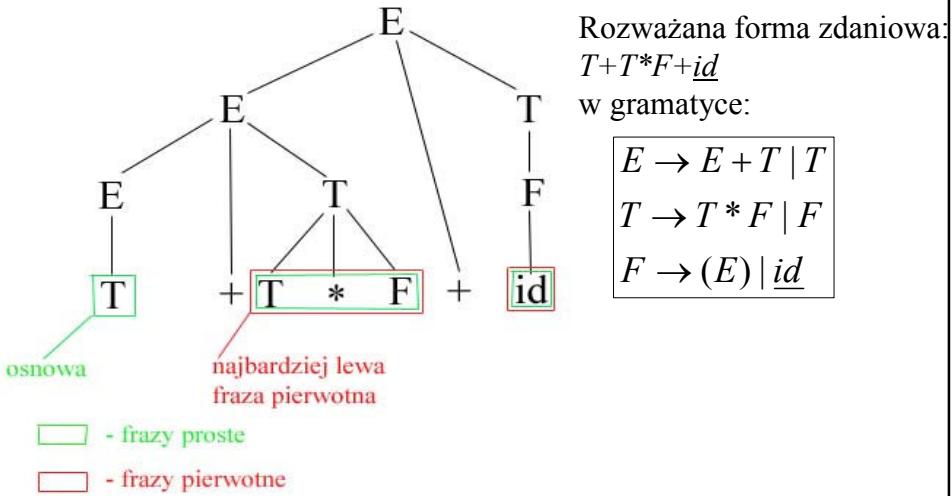


Fraza pierwotna

Mając daną gramatykę pierwszeństw operatorów i znając relacje pierwszeństw dla terminali tej gramatyki możemy w każdej formie zdaniowej określić frazę pierwotną.

Frazą pierwotną formy zdaniowej nazywamy dowolną frazą zawierającą co najmniej jeden terminal i nie zawierającą żadnej innej frazy pierwotnej (oprócz siebie).

Przykład



Relacje pierwszeństw

Twierdzenie:

Niech $G = \langle V, \Sigma, P, S \rangle$ - gramatyka pierwszeństw operatorów i

$$\$S\$ \xrightarrow[R]{*} \alpha A \omega \xrightarrow[R]{*} \alpha \beta \omega$$

Wówczas:

- (1) Relacje \prec lub \sqsubseteq spełnione są między kolejnymi terminalami (i symbolem \\$) łańcucha α
- (2) Relacja \prec spełniona jest między skrajnym prawym terminalem łańcucha α i skrajnym lewym terminalem łańcucha β
- (3) Relacja \sqsubseteq spełniona jest między kolejnymi terminalami łańcucha β
- (4) Relacja \succ spełniona jest między skrajnym prawym terminalem łańcucha β i pierwszym symbolem (oczywiście też terminalem, bo rozważamy wywód prawostronny) łańcucha ω



Przykład

	+	*	()	<u>id</u>	\$
+	•>	<•	<•	•>	<•	•>
*	•>	•>	<•	•>	<•	•>
(<•	<•	<•	≡	<•	
)	•>	•>		•>		•>
<u>id</u>	•>	•>		•>		•>
\$	<•	<•	<•		<•	

Przykład:

$$\underbrace{\$E\$}_{\$\$\$} \xrightarrow{R} \underbrace{\$E}_{\alpha} \underbrace{+}_{A} \underbrace{T}_{\omega} \underbrace{+ \underline{id}\$}_{\omega} \Rightarrow \underbrace{\$E}_{\alpha} \underbrace{+}_{\beta} \underbrace{T * F}_{\omega} \underbrace{+ \underline{id}\$}_{\omega}$$

$$\begin{array}{ccccccc} \$ & E & + & T & * & F & + & id & \$ \\ \$ & <• & + & <• & * & •> & + & <•^{**} & \underline{id} & •>^{**} & \$ \end{array}$$

przy czym:

** - relacje wewnętrz łańcucha ω , istotne później

β - fraza pierwotna (tutaj także osnowa)



Gramatyka szkieletowa

Definicja:

Niech $G = \langle V, \Sigma, P, S \rangle$ - gramatyka operatorowa.

Gramatyką szkieletową G_S dla gramatyki G nazywamy

gramatykę $G_S = \langle \{S\}, \Sigma, P', S \rangle$ zawierającą produkcję

typu: $(S \rightarrow X_1 \dots X_m) \in P'$ dla każdej produkcji:

$(A \rightarrow Y_1 \dots Y_m) \in P$, przy czym dla $1 \leq i \leq m$:

(1) $X_i = Y_i$ gdy $Y_i \in \Sigma$

(2) $X_i = S$ gdy $Y_i \in V$

oraz P' nie zawiera produkcji $S \rightarrow S$.

G_S - na ogół nie jest jednoznaczna.

$$L(G) \subseteq L(G_S)$$



Gramatyka szkieletowa

G_S - na ogół nie jest jednoznaczna.

$$L(G) \subseteq L(G_S)$$

Przy rozborze w oparciu o pierwszeństwa „operatorów” nie troszczymy się o nieterminale, gdyż są dla nas nieroróżnicialne, bo:

- nie znamy relacji pomiędzy nieterminałami
- nie potrafimy wykryć osnowy, gdy nie zawiera ona terminala
- nie umiemy w związku z tym redukować wg produkcji łańcuchowych



Gramatyka szkieletowa - przykład

Tworzymy gramatykę szkieletową zastępując wszystkie nieterminale symbolem początkowym S i eliminując produkcje łańcuchowe.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \underline{id}$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

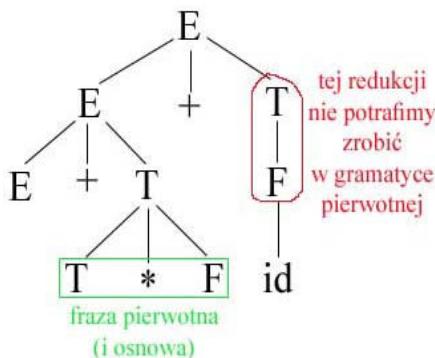
$$E \rightarrow (E)$$

$$E \rightarrow \underline{id}$$

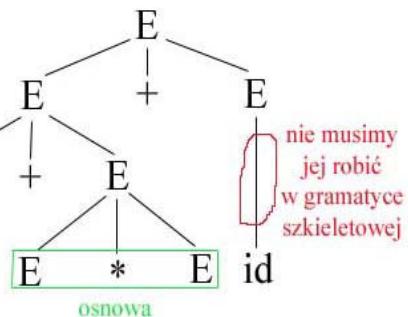


Gramatyka szkieletowa

Drzewo syntaktyczne w gramatyce pierwotnej



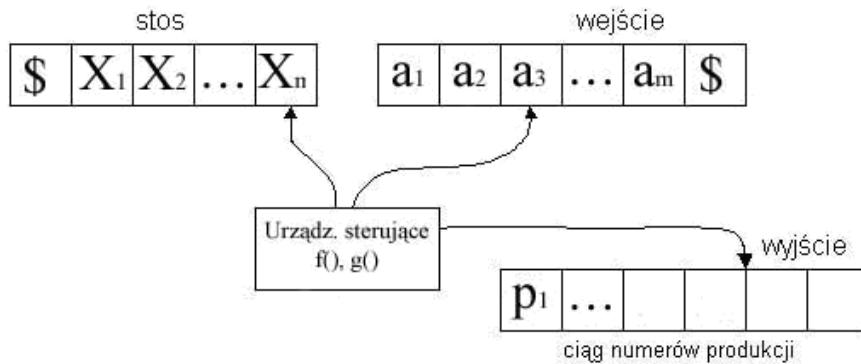
Drzewo syntaktyczne w gramatyce szkieletowej



Parsing szkieletowy

Otrzymana gramatyka szkieletowa jest na ogół niejednoznaczna i generuje słowa nie należące do języka gramatyki pierwotnej. JEDNOZNACZNY jest natomiast algorytm parsingu, który dodatkowo uwzględnia informacje z tablicy relacji pierwszeństw terminali i akceptuje słowa gramatyki pierwotnej. Dążymy do tego, aby algorytm parsingu akceptował słowa gramatyki pierwotnej i tylko te słowa, ale tak być nie musi. Rozbiór polega na odtworzeniu szkieletowego drzewa syntaktycznego poprzez redukcje osłów gramatyki szkieletowej (co sprowadza się do redukcji fraz pierwotnych w gramatyce pierwotnej).

Algorytm shift-reduce dla pierwszeństw operatorów



Algorytm shift-reduce dla pierwszeństw operatorów

We: $G_S = <\{S\}, \Sigma, P', S>$ oraz relacje pierwszeństw „operatorów” dla $G = <V, \Sigma, P, S>$

Wy: drzewo rozbiór syntaktycznego dla G_S

β – oznacza S lub ϵ

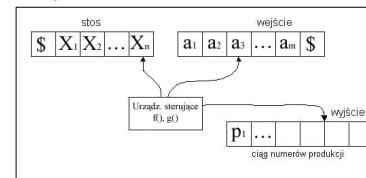
a, b – oznacza terminal lub $\$$

Działanie parsera:

Działanie parsera opisują funkcje f i g

Funkcja $f(\dots)$ jest funkcją określającą działanie parsera.

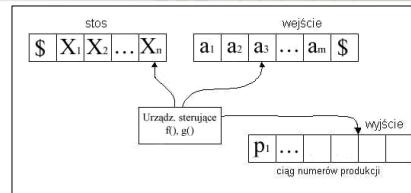
Funkcja $g(\dots)$, wywoływaną, gdy $f(\dots) = \text{red}$, jest odpowiedzialna za notowanie numerów produkcji.



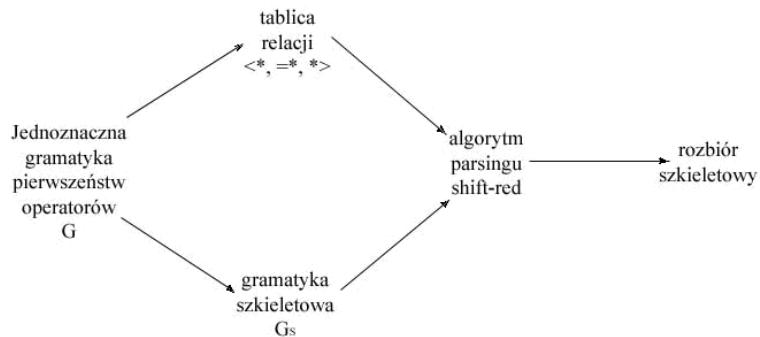
Algorytm shift-reduce dla pierwszeństw operatorów

Działanie parsera:

- (1) $f(a\beta, b) = \text{shift}$ gdy $a < \cdot b$ lub $a \equiv b$
- (2) $f(a\beta, b) = \text{red}$ gdy $a \cdot > b$
- (3) $f(\$, \$) = \text{acc}$
- (4) $f(\alpha, \omega) = \text{err}$ w pozostałych przypadkach
- (5) $g(a\beta b\gamma, \omega) = i$ gdy:
 - (a) β jest S lub ε
 - (b) $a < \cdot b$
 - (c) relacja \equiv jest spełniona dla kolejnych terminali łańcucha γ
 - (d) $S \rightarrow \beta b\gamma$ jest produkcją w G_S numerze i
- (6) $g(\alpha, \omega) = \text{err}$ w pozostałych przypadkach



Parsing wykorzystujący pierwszeństwa operatorów



Powyższa procedura może być uproszczona w następujący sposób:

- (1) konstruuje się tylko gramatykę szkieletową
- (2) określa się tablicę relacji pierwszeństw „operatorów” na podstawie zakładanych własności języka
- (3) buduje się na tej podstawie algorytm parsingu



Przykład

- (1) $E \rightarrow -E$ (minus unarny)
- (2) $E \rightarrow E \uparrow E$ (potęgowanie)
- (3) $E \rightarrow E * E$ (mnożenie)
- (4) $E \rightarrow E / E$ (dzielenie)
- (5) $E \rightarrow E + E$ (dodawanie)
- (6) $E \rightarrow E - E$ (odejmowanie)
- (7) $E \rightarrow (E)$
- (8) $E \rightarrow id$

Formułujemy własności operatorów:

Operator	Priorytet	Liczba operandów	Usytuowanie	Łączność
\neg	1 (najwyższy)	unarny	prefix	prawostronna
\uparrow	2	binarny		prawostronna
$*$, $/$	3	binarny		lewostronna
$+$, $-$	4	binarny		lewostronna



Przykład c.d.

Ustalamy relacje między terminalami:

- (1) Jeśli operator binarny Θ_1 ma wyższy priorytet niż operator Θ_2 , to: $\Theta_1 \bullet > \Theta_2$ i $\Theta_2 < \bullet \Theta_1$
np.: $* \bullet > +$, $+ < \bullet *$
- (2) Jeśli operator binarny Θ_1 ma taki sam priorytet jak operator Θ_2 , to:
 - (a) $\Theta_1 \bullet > \Theta_2$ i $\Theta_2 \bullet > \Theta_1$ dla lewostronne łącznych
 - (b) $\Theta_1 < \bullet \Theta_2$ i $\Theta_2 < \bullet \Theta_1$ dla prawostronnie łącznych

np.: lewostronne łączne: $+ \bullet > -$, $- \bullet > +$, $+ \bullet > +$, $- \bullet > -$
prawostronnie łączne: $\uparrow < \bullet \uparrow$



Przykład c.d.

(3) Jeśli ϑ jest unarnym prefixem, to:

- (a) $\Theta <\!\!> \vartheta$ dla każdego operatora Θ - binarnego lub unarnego
- (b) $\vartheta >\!\!> \Theta$ gdy ϑ ma wyższy priorytet od Θ
- (c) $\vartheta <\!\!< \Theta$ gdy ϑ ma niższy priorytet od Θ

np.: $\neg <\!\!> \neg$, $+ <\!\!> \neg$, $\neg >\!\!> +$



Przykład c.d.

(4) Pozostałe relacje (Θ oznacza tutaj dowolny operator):

$\Theta <\!\!< \underline{id}$, $\Theta <\!\!< (\dots) >\!\!> \Theta$, $\Theta >\!\!> \$$, $\underline{id} >\!\!> \Theta$, $(<\!\!> \Theta >\!\!>)$, $\$ <\!\!< \Theta$, $\dot{(=)}$, $(<\!\!> (\dots) >\!\!>)$, $\$ <\!\!< (\dots) >\!\!> \$$, $\$ <\!\!< \underline{id}$, $\underline{id} >\!\!> \$$, $(<\!\!> \underline{id}, \underline{id} >\!\!>)$



Przykład c.d.

Tworzymy tabelę relacji:

	+	-	*	/	↑	<u>id</u>	()	¬	\$
+	•>	•>	<•	<•	<•	<•	<•	•>	<•	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	<•	•>
*	•>	•>	•>	•>	•>	<•	<•	•>	<•	•>
/	•>	•>	•>	•>	•>	<•	<•	•>	<•	•>
↑	•>	•>	•>	•>	•>	<•	<•	•>	<•	•>
<u>id</u>	•>	•>	•>	•>	•>	err 3	err 3	•>	•>	•>
(<•	<•	<•	<•	<•	<•	<•	≡	<•	err 4
)	•>	•>	•>	•>	•>	err 3	err 3	•>	•>	•>
¬	•>	•>	•>	•>	•>	<•	<•	•>	<•	•>
\$	<•	<•	<•	<•	<•	<•	<•	err 2	<•	err 1



Przykład c.d.

Precyzujemy algorytm parsera: (β oznacza E lub ε)

$$\begin{aligned}f(a\beta, b) &= \underline{\text{shift}}, \text{ gdy } a <\cdot b \text{ lub } a = b \\f(a\beta, b) &= \underline{\text{red}}, \text{ gdy } a \cdot > b \\f(\$, \$) &= \underline{\text{acc}}, \\f(\dots, \dots) &= \underline{\text{err}}, \text{ w pozostałych przypadkach.} \\g(b\beta \rightarrow E) &= 1, \text{ gdy } b <\cdot \neg \\g(bE \uparrow E) &= 2, \text{ gdy } b <\cdot E \\g(bE^* E) &= 3, \text{ gdy } b <\cdot * \\&\dots\dots \\&\dots\dots \\g(b\beta(E)) &= 7, \text{ gdy } b <\cdot (\\g(b\beta \underline{id}) &= 8, \text{ gdy } b <\cdot \underline{id} \\g(\dots, \dots) &= \underline{\text{err}}, \text{ w pozostałych przypadkach.}\end{aligned}$$



Przykład c.d.

Przykład rozbioru szkieletowego dokonanego przez parser:

\$	<u>id</u> * \neg (<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	ε	$\mapsto [\$, \text{shift}]$
\$ <u>id</u>	\ast \neg (<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	ε	$\mapsto [\text{id} \bullet > \ast, \text{red-8}]$
\$E	\ast \neg (<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	8	$\mapsto [\$, \text{shift}]$
\$E*	\neg (<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	8	$\mapsto [\ast \bullet > \neg, \text{shift}]$
\$E*\neg	(<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	8	$\mapsto [\neg \bullet > (\text{id}), \text{shift}]$
\$E*\neg(<u>id</u> + <u>id</u>) \uparrow <u>id</u> \$	8	$\mapsto [(\bullet > \text{id}), \text{shift}]$
\$E*\neg(id)	<u>id</u>) \uparrow <u>id</u> \$	8	$\mapsto [\text{id} \bullet > +, \text{red-8}]$
\$E*\neg(E	<u>id</u>) \uparrow <u>id</u> \$	88	$\mapsto [(<+ \bullet >, \text{shift}]$
\$E*\neg(E+	<u>id</u>) \uparrow <u>id</u> \$	88	$\mapsto [+ < \bullet > \text{id}, \text{shift}]$
\$E*\neg(E+id)) \uparrow <u>id</u> \$	88	$\mapsto [\text{id} \bullet >), \text{red-8}]$



Przykład c.d.

\$E*\neg(E+E) \uparrow <u>id</u> \$	888	$\mapsto [+ \bullet >), \text{red-5}]$
\$E*\neg(E) \uparrow <u>id</u> \$	8885	$\mapsto [(\bullet >, \text{shift}]$
\$E*\neg(E)	\uparrow <u>id</u> \$	8885	$\mapsto [\bullet > \uparrow, \text{red-7}]$
\$E*\neg E	\uparrow <u>id</u> \$	88857	$\mapsto [\neg \bullet > \uparrow, \text{red-1}]$
\$E*E	\uparrow <u>id</u> \$	888571	$\mapsto [* \bullet > \uparrow, \text{shift}]$
\$E*E \uparrow	<u>id</u> \$	888571	$\mapsto [\uparrow \bullet > \text{id}, \text{shift}]$
\$E*E \uparrow <u>id</u>	\$	888571	$\mapsto [\text{id} \bullet > \$, \text{red-8}]$
\$E*E \uparrow E	\$	8885718	$\mapsto [\uparrow \bullet > \$, \text{red-2}]$
\$E*E	\$	88857182	$\mapsto [* \bullet > \$, \text{red-3}]$
\$E	\$	888571823	$\mapsto \text{acc}$



Przykład c.d.

- (1) Błędy syntaktyczne wykrywane są wówczas, gdy nie jest określone pierwszeństwo terminali (por. tablica relacji pierwszeństw i definicja funkcji $f(\dots, \dots)$):

err 1: „missing operand”

Akcja: wprowadzenie „id” na wejście

err 2: „unbalanced right parenthesis”

Akcja: zlikwidowanie „)” z wejścia

	<u>id</u>	()	\$
<u>id</u>	err 3	err 3	•>	•>
(<•	<•	≡	err 4
)	err 3	err 3	•>	•>
\$	<•	<•	err 2	err 1

err 3: „missing operator”

Akcja: wprowadzenie „+” na wejście

err 4: „missing right parenthesis”

Akcja: zdjęcie „(” ze stosu wraz ze wszystkimi symbolami, które są między „(” a wierzchołkiem stosu z wierzchołkiem stosu włącznie



Przykład c.d.

- (2) Błędy syntaktyczne wykrywane wówczas, gdy badanie pierwszeństwa wskazuje na redukcję, a redukcji nie da się dokonać, gdyż zawartość stosu nie odpowiada prawej stronie żadnej produkcji (por. definicja funkcji $g(\dots, \dots)$):

err 5: „missing operand(s)”

Redukowany jest operator (+, -, /, *, ↑, ¬), a brak jednego lub obu nieterminali E .

Akcja: redukcja mimo wszystko, operator na stosie wskazuje, wg której produkcji redukować

err 6: „no expression between parenthesis”

Brak nieterminala E pomiędzy “(” a “)”.

Akcja: redukcja wg produkcji 7 mimo błędu



Przykład c.d.

err 7: „missing operator”

Redukowany jest id, po lewej stronie id na stosie nie ma terminala.

Akcja: zdjęcie ze stosu nieterminala E , poprzedzającego id, po czym redukcja wg produkcji 8

err 8: „missing operator”

Redukowane są „ (E) ”, po lewej stronie „ $($ ” na stosie jest E .

Akcja: zdjęcie ze stosu nieterminala E poprzedzającego „ $($ ”, po czym redukcja wg produkcji 7



Przykład c.d.

\$	+ - <u>id</u>) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [\$, <\bullet +, \text{shift}]$
\$+	- <u>id</u>) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [+ \bullet > -, \text{red-5}, \text{err-5}]$

ERROR 5: „missing operands”

\$E	- <u>id</u>) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [\$, <\bullet -, \text{shift}]$
\$E-	<u>id</u>) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [- <\bullet \text{id}, \text{shift}]$
\$E- <u>id</u>) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [\text{id} \bullet >), \text{red-8}]$
\$E-E) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [- \bullet >), \text{red-6}]$
\$E) <u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [\$?), \text{err-2}]$

ERROR 2: „unbalanced right parenthesis”

\$E	<u>id()</u> <u>id</u> <u>id\$</u>	$\mapsto [\$, <\bullet \text{id}, \text{shift}]$
-----	-----------------------------------	--



Przykład c.d.

\$E <u>id</u>	$\lambda() \underline{id} \underline{id\$}$	$\mapsto [\underline{id?}(, \underline{err-3})]$
ERROR 3: „missing operator”		
\$E <u>id</u>	$\lambda +() \underline{id} \underline{id\$}$	$\mapsto [\underline{id}\bullet>+, \underline{red-8} \underline{err-7}]$
ERROR 7: „missing operator”		
\$E	$\lambda +() \underline{id} \underline{id\$}$	$\mapsto [\$, \underline{shift}]$
\$E+	$\lambda () \underline{id} \underline{id\$}$	$\mapsto [+<\bullet(, \underline{shift}]$
\$E+($\lambda) \underline{id} \underline{id\$}$	$\mapsto [(\dot{=}), \underline{shift}]$
\$E+()	$\lambda) \underline{id} \underline{id\$}$	$\mapsto [)\bullet>, \underline{red-7}, \underline{err-6}]$
ERROR 6: „no expression between parenthesis”		
\$E+E	$\lambda) \underline{id} \underline{id\$}$	$\mapsto [+•>), \underline{red-5}]$
\$E	$\lambda) \underline{id} \underline{id\$}$	$\mapsto [\$?), \underline{err-2}]$
ERROR 2: “unbalanced right parenthesis”		
\$E	$\lambda \underline{id} \underline{id\$}$	$\mapsto [\$, <\bullet \underline{id}, \underline{shift}]$



Przykład c.d.

\$E <u>id</u>	$\lambda \underline{id\$}$	$\mapsto [\underline{id?}\underline{id}, \underline{err-3}]$
ERROR 3: “missing operator”		
\$E <u>id</u>	$\lambda +\underline{id\$}$?
\$E <u>id</u>	$\lambda +\underline{id\$}$	$\mapsto [\underline{id}\bullet>+, \underline{red-8}, \underline{err-7}]$
ERROR 7: „missing operator”		
\$E	$\lambda +\underline{id\$}$	$\mapsto [\$, \underline{shift}]$
\$E+	$\lambda \underline{id\$}$	$\mapsto [+<\bullet \underline{id}, \underline{shift}]$
\$E+ <u>id</u>	$\lambda \$$	$\mapsto [\underline{id}\bullet> \$, \underline{red-8}]$
\$E+E	$\lambda \$$	$\mapsto [+•> \$, \underline{red-5}]$
\$E	$\lambda \$$	$\mapsto \underline{error}, \text{ bo były błędy}$



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Gramatyki atrybutywne, część 1 (gramatyki S-atrybutywne) **Teoria komplikacji**

Dr inż. Janusz Majewski
Katedra Informatyki



Gramatyki atrybutywne

- Do przeprowadzenia poprawnego tłumaczenia, oprócz informacji na temat składni języka podlegającego tłumaczeniu, translator musi posiadać możliwość korzystania z wielu innych informacji dotyczących tłumaczonych konstrukcji. Informacje te mają charakter semantyczny. Dotyczą np. typu zmiennych, typu wyrażeń, wielkości pamięci dla zmiennych, adresów odpowiednich zapisów w tablicach, wartości wyrażeń w procesie interpretacji, itd.
- Informacje semantyczne mają charakter atrybutów stwarzyszonych z konstrukcjami języka podlegającego tłumaczeniu. Jednym ze sposobów formalnego określenia semantyki języka i zdefiniowania akcji podejmowanych w procesie translacji jest aparat gramatyk atrybutywnych.



Atrybutowane drzewa rozbioru

Analiza semantyczna i generowanie kodu oparte są na drzewie rozbioru syntaktycznego. Każdy wierzchołek drzewa (symbol gramatyki) jest „udekorowany” atrybutami opisującymi własności wierzchołka. Dla podkreślenia tego faktu takie drzewo nazywa się często „atrybutowanym drzewem rozbioru syntaktycznego”. Informację zgromadzoną w atrybutach wierzchołka wyprowadza się z jego otoczenia. Obliczenie atrybutów i sprawdzenie ich zgodności jest zadaniem analizy semantycznej. Optymalizację i generację kodu również można opisać w podobny sposób, używając atrybutów do sterowania przekształcaniem drzewa i w końcu do sterowania wyborem instrukcji maszynowych.



Wykorzystanie gramatyk atrybutywnych

Teoretyczny mechanizm gramatyk atrybutywnych może być wykorzystany:

- w analizie semantycznej do skontrolowania tych wszystkich aspektów kodu źródłowego, które nie zostały zbadane przez analizator leksykalny i syntaktyczny,
- przy wykonywaniu tłumaczenia do kodu pośredniego,
- przy obliczaniu pewnych wartości związanych z drzewem rozbioru (np. konstrukcja kalkulatora software'owego),
- do wykonywania pewnych akcji związanych z węzłami drzewa rozbioru (np. zapis lub odczyt z tablicy symboli, utworzenie lub wykorzystanie innych struktur danych, zapis i/lub odczyt z pliku, wydrukowanie czegoś, itp.); akcje te nie są „wyliczeniem” wartości atrybutów w sensie dosłownym.



Odwrotna notacja polska (notacja postfiksowa)

Przypomnienie:

notacja nawiasowa = notacja infiksowa
odwrotna notacja polska = notacja postfiksowa

Niech będą dane dwa zbiory:

OPERATOR – zbiór operatorów binarnych
OPERAND – zbiór pojedynczych operandów

1. Jeśli wyrażenie E w notacji infiksowej jest pojedynczym operandem ($E \in \text{OPERAND}$), to postać postfiksowa tego wyrażenia to E .
2. Jeśli $E_1 \Theta E_2$ jest wyrażeniem w postaci infiksowej, E_1 i E_2 są wyrażeniami w notacji infiksowej, $\Theta \in \text{OPERATOR}$, to:
 $E'_1 E'_2 \Theta$ jest zapisem postfiksowym wyrażenia $E_1 \Theta E_2$
przy czym: E'_1 i E'_2 – są odpowiednikami wyrażeń E_1 i E_2 w notacji postfiksowej
3. Jeśli (E) jest wyrażeniem infiksowym, to:
 E' jest zapisem wyrażenia (E) w notacji postfiksowej
przy czym E' jest odpowiednikiem wyrażenia E w notacji infiksowej.



Przykłady

Notacja infiksowa: **a+b*c**

Notacja postfiksowa: **abc*c+**

Notacja infiksowa: **a*b+c**

Notacja postfiksowa: **ab*c+**

Notacja infiksowa: **(a+b)*c**

Notacja postfiksowa: **ab+c***

Notacja infiksowa: **a*(b+c)**

Notacja postfiksowa: **abc+***



Przykład – translacja wyrażeń do odwrotnej notacji polskiej (ONP)

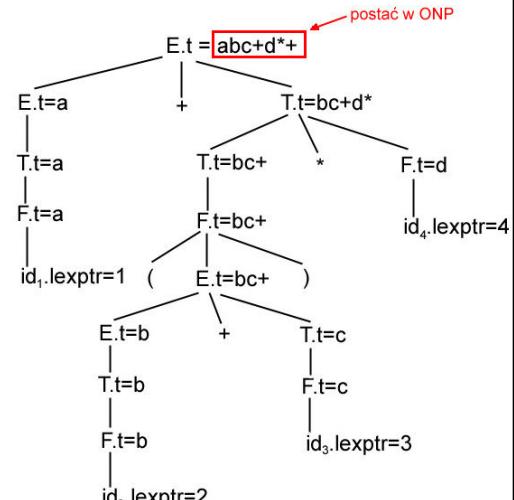
$E \rightarrow E_1 + T$	$E.t \leftarrow E_1.t \parallel T.t \parallel '+'$
$E \rightarrow T$	$E.t \leftarrow T.t$
$T \rightarrow T_1 * F$	$T.t \leftarrow T_1.t \parallel F.t \parallel '*'$
$T \rightarrow F$	$T.t \leftarrow F.t$
$F \rightarrow (E)$	$F.t \leftarrow E.t$
$F \rightarrow id$	$F.t \leftarrow name(id.lexptr)$

gdzie: \parallel - operator konkatenacji tekstów

Rozważane słowo źródłowe: $a+(b+c)*d$

Po analizie leksykalnej: $id_1+(id_2+id_3)*id_4$

<u>id.lexptr</u>	<u>name(id.lexptr)</u>
1	a
2	b
3	c
4	d



Przykład akcji – translacja instrukcji przypisania do kodu trójadresowego

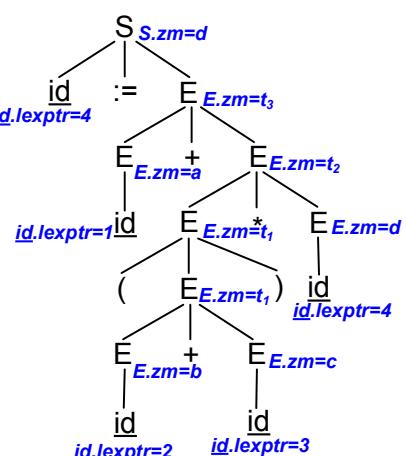
$S \rightarrow id := E$	$S.zm \leftarrow name(id.lexptr)$ $gen(S.zm) ::= \parallel E.zm$
$E \rightarrow E_1 + E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm) ::= \parallel E_1.zm \parallel '+' \parallel E_2.zm$
$E \rightarrow E_1 * E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm) ::= \parallel E_1.zm \parallel '*' \parallel E_2.zm$
$E \rightarrow (E_1)$	$E.zm \leftarrow E_1.zm$
$E \rightarrow id$	$E.zm \leftarrow name(id.lexptr)$

gdzie: \parallel - operator konkatenacji tekstów

Rozważane słowo źródłowe: $d:=a+(b+c)*d$

Po analizie leksykalnej: $id_4:=id_1+(id_2+id_3)*id_4$

<u>id.lexptr</u>	<u>name(id.lexptr)</u>
1	a
2	b
3	c
4	d





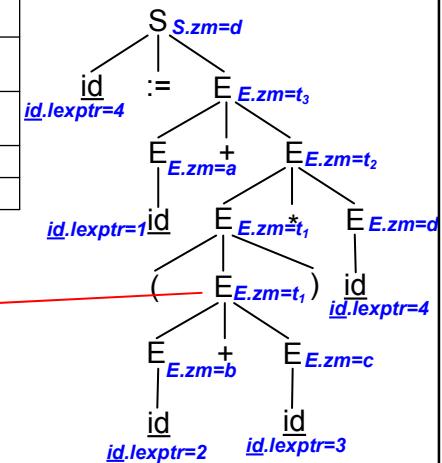
Przykład akcji – translacja instrukcji przypisania do kodu trójadresowego

$S \rightarrow \underline{id} := E$	$S.zm \leftarrow name(\underline{id}.lexptr)$ $gen(S.zm \parallel ":=\parallel E.zm)$
$E \rightarrow E_1 + E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel ":+\parallel E_1.zm \parallel "+" \parallel E_2.zm)$
$E \rightarrow E_1 * E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel ":\ast\parallel E_1.zm \parallel "\ast" \parallel E_2.zm)$
$E \rightarrow (E_1)$	$E.zm \leftarrow E_1.zm$
$E \rightarrow id$	$E.zm \leftarrow name(\underline{id}.lexptr)$

słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

t₁ := b + c



Przykład akcji – translacja instrukcji przypisania do kodu trójadresowego

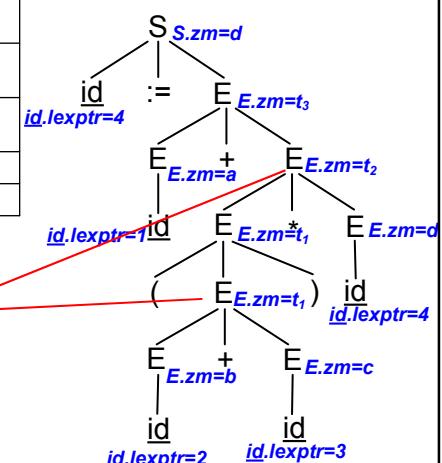
$S \rightarrow \underline{id} := E$	$S.zm \leftarrow name(\underline{id}.lexptr)$ $gen(S.zm \parallel ":=\parallel E.zm)$
$E \rightarrow E_1 + E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel ":+\parallel E_1.zm \parallel "+" \parallel E_2.zm)$
$E \rightarrow E_1 * E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel ":\ast\parallel E_1.zm \parallel "\ast" \parallel E_2.zm)$
$E \rightarrow (E_1)$	$E.zm \leftarrow E_1.zm$
$E \rightarrow id$	$E.zm \leftarrow name(\underline{id}.lexptr)$

słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

t₁ := b + c

t₂ := t₁ * d





Przykład akcji – translacja instrukcji przypisania do kodu trójadresowego

$S \rightarrow \underline{id} := E$	$S.zm \leftarrow name(\underline{id}.lexptr)$ $gen(S.zm \parallel " := " \parallel E.zm)$
$E \rightarrow E_1 + E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel " := " \parallel E_1.zm \parallel "+" \parallel E_2.zm)$
$E \rightarrow E_1 * E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel " := " \parallel E_1.zm \parallel "*" \parallel E_2.zm)$
$E \rightarrow (E_1)$	$E.zm \leftarrow E_1.zm$
$E \rightarrow id$	$E.zm \leftarrow name(\underline{id}.lexptr)$

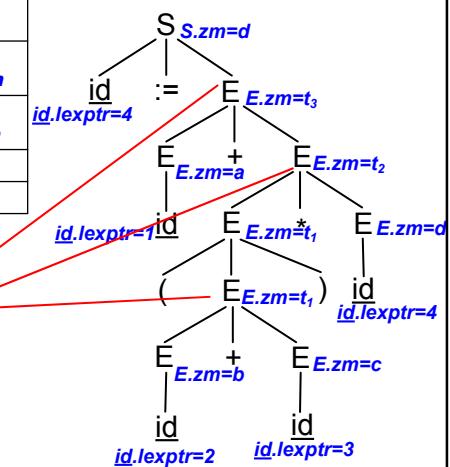
słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

$t_1 := b + c$

$t_2 := t_1 * d$

$t_3 := a + t_2$



Przykład akcji – translacja instrukcji przypisania do kodu trójadresowego

$S \rightarrow \underline{id} := E$	$S.zm \leftarrow name(\underline{id}.lexptr)$ $gen(S.zm \parallel " := " \parallel E.zm)$
$E \rightarrow E_1 + E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel " := " \parallel E_1.zm \parallel "+" \parallel E_2.zm)$
$E \rightarrow E_1 * E_2$	$E.zm \leftarrow new_temp()$ $gen(E.zm \parallel " := " \parallel E_1.zm \parallel "*" \parallel E_2.zm)$
$E \rightarrow (E_1)$	$E.zm \leftarrow E_1.zm$
$E \rightarrow id$	$E.zm \leftarrow name(\underline{id}.lexptr)$

słowo źródłowe: **d:=a+(b+c)*d**

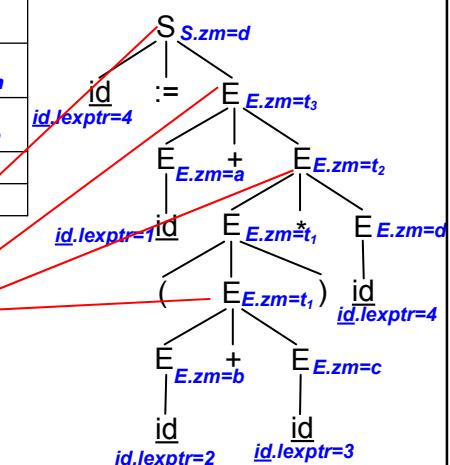
Tłumaczenie:

$t_1 := b + c$

$t_2 := t_1 * d$

$t_3 := a + t_2$

$d := t_3$





Przykłady reguł semantycznych

- Deklaracje typu:

$P \rightarrow D ; S$	
$D \rightarrow D ; D$	
$D \rightarrow id : T$	$\{addtype(id.entry, T.type)\}$
$T \rightarrow char$	$\{T.type \leftarrow char\}$
$T \rightarrow integer$	$\{T.type \leftarrow integer\}$
$T \rightarrow boolean$	$\{T.type \leftarrow boolean\}$
$T \rightarrow \uparrow T_1$	$\{T.type \leftarrow pointer(T_1.type)\}$
$T \rightarrow array [num] of T_1$	$\{T.type \leftarrow array(1..num.val, T_1.type)\}$



Przykłady reguł semantycznych

- Ustalanie typu wyrażeń:

$E \rightarrow num$	$\{E.type \leftarrow integer\}$
$E \rightarrow num . num$	$\{E.type \leftarrow real\}$
$E \rightarrow id$	$\{E.type \leftarrow lookup(id.lexptr)\}$
$E \rightarrow E_1 op E_2$	$\{E.type \leftarrow \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \text{ then integer}$ $\quad \quad \quad \text{else if } E_1.type = \text{integer} \text{ and } E_2.type = \text{real} \text{ then real}$ $\quad \quad \quad \text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{integer} \text{ then real}$ $\quad \quad \quad \text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{real} \text{ then real}$ $\quad \quad \quad \text{else type_error}\}$



Przykłady reguł semantycznych

- *Kontrola typu instrukcji:*

$S \rightarrow id := E$	$\{S.type \leftarrow \text{if } lookup(id.entry) = E.type \text{ then void else type_error}\}$
$S \rightarrow \text{if } E \text{ then } S_1$	$\{S.type \leftarrow \text{if } E.type = \text{boolean then } S_1.type \text{ else type_error}\}$
$S \rightarrow \text{while } E \text{ do } S_1$	$\{S.type \leftarrow \text{if } E.type = \text{boolean then } S_1.type \text{ else type_error}\}$
$S \rightarrow S_1 ; S_2$	$\{S.type \leftarrow \text{if } S_1.type = \text{void and } S_2.type = \text{void then void else type_error}\}$



Gramatyka atrybutywna (1)

- Gramatyka atrybutywna jest oparta na gramatyce bezkontekstowej $G = \langle V, \Sigma, P, S \rangle$.
- Łączy ona z każdym symbolem gramatyki $X \in (V \cup \Sigma)$ zbiór atrybutów $A(X)$.
- Każdy atrybut $X.a \in A(X)$ reprezentuje specyficzną własność symbolu X i może przyjąć którykolwiek wartość z pewnego zbioru wartości.
- $X.a$ oznacza atrybut „ a ” będący elementem zbioru $A(X)$.
Każdy wierzchołek w drzewie rozbiór syntaktycznego słowa z języka $L(G)$ jest związany ze zbiorem wartości atrybutów dla tego symbolu X , który stanowi etykietę tego wierzchołka.
Wartości atrybutów ustala się za pomocą reguł semantycznych.



Gramatyka atrybutywna (2)

Zbiór $R(p)$:

$$R(p) \stackrel{df}{=} \left\{ X_i.a \leftarrow f(X_j.b, \dots, X_k.c) \right\}$$

jest zbiorem reguł semantycznych dla produkcji $p \in P$ takiej, że:

$$p = (X_0 \rightarrow X_1 \dots X_n)$$

$$0 \leq i \leq n$$

$$0 \leq j \leq n$$

$$0 \leq k \leq n$$

Reguła semantyczna $X_i.a \leftarrow f(X_j.b, \dots, X_k.c)$ definiuje atrybut $X_i.a$ za pomocą atrybutów $X_j.b, \dots, X_k.c$ symboli z tej samej produkcji.



Gramatyka atrybutywna (3)

Gramatyka atrybutywna AG jest trójką:

$$AG = \langle G, A, R \rangle$$

gdzie:

$$G = \langle V, \Sigma, P, S \rangle \in \mathcal{G}_{BK} - \text{gramatyka bezkontekstowa}$$

$$A = \bigcup_{X \in (V \cup \Sigma)} A(X) - \text{skończony zbiór atrybutów}$$

$$R = \bigcup_{p \in P} R(p) - \text{skończony zbiór reguł semantycznych}$$

Jeśli $A(X) \cap A(Y) \neq \emptyset$ to $X=Y$

Dla każdego wystąpienia X w drzewie rozboru syntaktycznego słowa z języka $L(G)$, do obliczenia każdego atrybutu $X.a \in A(X)$ można zastosować co najwyżej jedną regułę semantyczną.



Gramatyka atrybutywna (4)

Dla każdej produkcji $p = (X_0 \rightarrow X_1 \dots X_n) \in P$ zbiorem definiujących wystąpień atrybutów jest:

$$AF(p) = \{X_i.a : X_i.a \leftarrow f(\dots) \in R(p)\}$$

Atrybut $X.a$ nazywamy syntetyzowanym $\stackrel{df}{\Leftrightarrow}$
 $\exists p = (X \rightarrow \chi) \in P : X.a \in AF(p)$

Atrybut $X.a$ nazywamy dziedzicznym $\stackrel{df}{\Leftrightarrow}$
 $\exists p = (Y \rightarrow \mu X v) \in P : X.a \in AF(p)$



Gramatyka atrybutywna (5)

$\forall X \in (V \cup \Sigma)$ zachodzi $AS(X) \cap AI(X) = \emptyset$

gdzie:

$$AS(X) \stackrel{df}{=} \{X.a : \exists p = (X \rightarrow \chi) \in P \wedge X.a \in AF(p)\}$$

$AS(X)$ – zbiór atrybutów syntetyzowanych dla $X \in (V \cup \Sigma)$

$$AI(X) \stackrel{df}{=} \{X.a : \exists q = (Y \rightarrow \mu X v) \in P \wedge X.a \in AF(q)\}$$

$AI(X)$ – zbiór atrybutów dziedziczonych dla $X \in (V \cup \Sigma)$

Ponadto istnieje co najwyżej jedna reguła $X.a \leftarrow f(\dots)$ w zbiorze $R(p)$ dla każdego $p \in P$ i $X.a \in A(X)$.



Gramatyka atrybutywna (6)

Definicja:

Gramatyka atrybutywna jest zupełna, jeśli dla wszystkich symboli $X \in (V \cup \Sigma)$ są spełnione następujące warunki:

$$\forall p = (X \rightarrow \chi) \in P : AS(X) \subseteq AF(p)$$

$$\forall q = (Y \rightarrow \mu X v) \in P : AI(X) \subseteq AF(q)$$

$$AS(X) \cup AI(X) = A(X)$$

Ponadto: $AI(S) = \emptyset$

Definicja

Gramatyka atrybutywna jest dobrze zdefiniowana jeśli dla każdego drzewa rozboru syntaktycznego słowa z $L(G)$ wszystkie atrybuty są efektywnie obliczalne.



Gramatyki S-atrybutywne

Gramatyką atrybutyną klasy S (gramatyką S-atrybutyną) nazywamy gramatykę atrybutyną zawierającą tylko i wyłącznie atrybuty syntetyzowane.

Mogą one być obliczone z wykorzystaniem drzewa rozboru syntaktycznego metodą bottom-up (\rightarrow por. poprzedni przykład z translacją wyrażeń do ONP; \rightarrow por. następny przykład kalkulatora).



Przykład – kalkulator

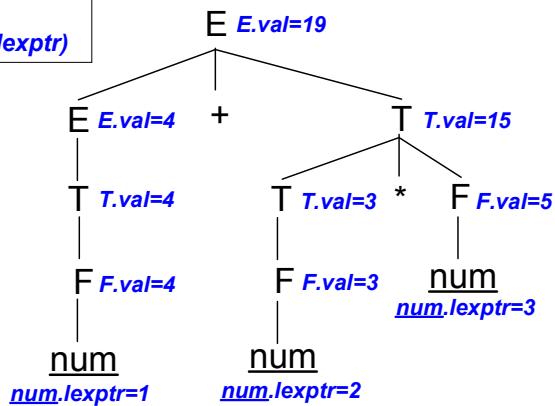
$E \rightarrow E_1 + T$	$E.\text{val} \leftarrow E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} \leftarrow T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} \leftarrow T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} \leftarrow F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} \leftarrow E.\text{val}$
$F \rightarrow \underline{\text{num}}$	$F.\text{val} \leftarrow \text{value}(\underline{\text{num}}.\text{lexptr})$

Słowo źródłowe: **4+3*5**

Po skanerze: **num₁+num₂*num₃**

Tablica symboli:

<u>num.lexptr</u>	value(<u>num.lexptr</u>)
1	4
2	3
3	5



Translacja „bottom-up” dla gramatyk S-trybutywnych

na stosie “state” faktycznie stany, a nie symbole gramatyki

		top
		top-1
		top-2
F	5	
*		
T	3	
⋮	⋮	
state		
attr		

Przed redukcją

$T \rightarrow T_1 * F$

produkcja

$T.\text{val} \leftarrow T_1.\text{val} * F.\text{val}$

reguła semantyczna

ntop:=top-2
attr[ntop]:=attr[top-2]*attr[top]

fragment kodu translatora

na stosie “attr” wartość atrybutu syntetyzowanego lub wskaznik, gdy jest więcej niż jeden atrybut

Po redukcji

		ntop
.		
T	15	
⋮	⋮	

Ogólnie:

ntop:=top-r+1
poprzedni wierzchołek stosu
długość prawej strony produkcji



Przykład – kalkulator c.d.

$L \rightarrow E\$$	$L.b \leftarrow print(E.val)$	$print(attr[top-1])$
$E \rightarrow E_I + T$	$E.val \leftarrow E_I.val + T.val$	$attr[ntop] := attr[top-2] + attr[top]$
$E \rightarrow T$	$E.val \leftarrow T.val$	$attr[ntop] := attr[top-2] * attr[top]$
$T \rightarrow T_I * F$	$T.val \leftarrow T_I.val * F.val$	$attr[ntop] := attr[top-1]$
$T \rightarrow F$	$T.val \leftarrow F.val$	$attr[ntop] := value(attr[top])$
$F \rightarrow (E)$	$F.val \leftarrow E.val$	
$F \rightarrow \underline{num}$	$F.val \leftarrow value(\underline{num}.lexptr)$	

Założenie: po shift num na stosie attr umieszczana jest wartość num.lexptr

Słowo wejściowe: num₁ * num₂ + num₃\$ (3*5+4)

<u>num.lexptr</u>	<u>value(num.lexptr)</u>
1	3
2	5
3	4



Przykład – kalkulator c.d.

Słowo wejściowe: num₁ * num₂ + num₃\$ (3*5+4)

wejście	state	attr	Produkcja
<u>num</u> * <u>num</u> + <u>num</u> \$			
* <u>num</u> + <u>num</u> \$	<u>num</u>	1	
* <u>num</u> + <u>num</u> \$	F	3	$F \rightarrow num$
* <u>num</u> + <u>num</u> \$	T	3	$T \rightarrow F$
<u>num</u> + <u>num</u> \$	T*	3□	
+ <u>num</u> \$	T*num	3□2	
+ <u>num</u> \$	T*F	3□5	$F \rightarrow num$
+ <u>num</u> \$	T	15	$T \rightarrow T * F$
+ <u>num</u> \$	E	15	$E \rightarrow T$
<u>num</u> \$	E+	15□	
\$	E+num	15□3	
\$	E+F	15□4	$F \rightarrow num$
\$	E+T	15□4	$T \rightarrow F$
\$	E	19	$E \rightarrow E + T$
E\$		19□	
L	(*)		

(*) – wydrukowanie wartości =19



AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Gramatyki atrybutywne, część 2 (gramatyki L-atrybutywne)

Teoria kompilacji

**Dr inż. Janusz Majewski
Katedra Informatyki**

Przypomnienie: rodzaje atrybutów

Dla każdej produkcji $p = (X_0 \rightarrow X_1 \dots X_n) \in P$ zbiorem definiujących wystąpień atrybutów jest:

$$AF(p) = \{X_i.a : X_i.a \leftarrow f(\dots) \in R(p)\}$$

Atrybut $X.a$ nazywamy syntetyzowanym $\stackrel{df}{\Leftrightarrow}$
 $\exists p = (X \rightarrow \chi) \in P : X.a \in AF(p)$

Atrybut $X.a$ nazywamy dziedziczonym $\stackrel{df}{\Leftrightarrow}$
 $\exists p = (Y \rightarrow \mu X \nu) \in P : X.a \in AF(p)$

Przykład – gramatyka atrybutywna z atrybutami dziedzicznymi

$D \rightarrow TL$	$L.in \leftarrow T.type$
$T \rightarrow \underline{int}$	$T.type \leftarrow integer$
$T \rightarrow \underline{float}$	$T.type \leftarrow real$
$L \rightarrow L_1, \underline{id}$	$L_1.in \leftarrow L.in$ $L.b \leftarrow addtype(\underline{id}.lexptr, L.in)$
$L \rightarrow \underline{id}$	$L.b \leftarrow addtype(\underline{id}.lexptr, L.in)$

Funkcja *addtype()* wstawia do tablicy symboli informację o typie zmiennej

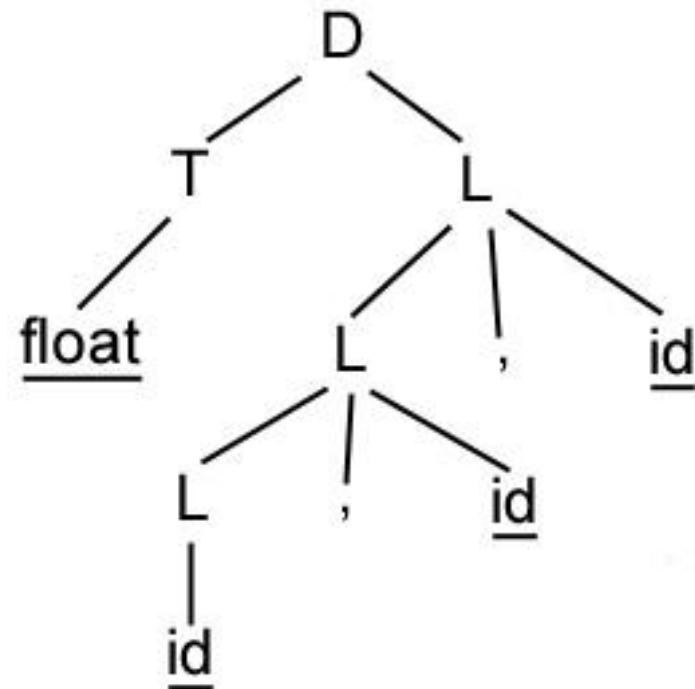
Analizowane słowo: *float id₁,id₂,id₃*

Przykład – gramatyka atrybutywna z atrybutami dziedzicznymi

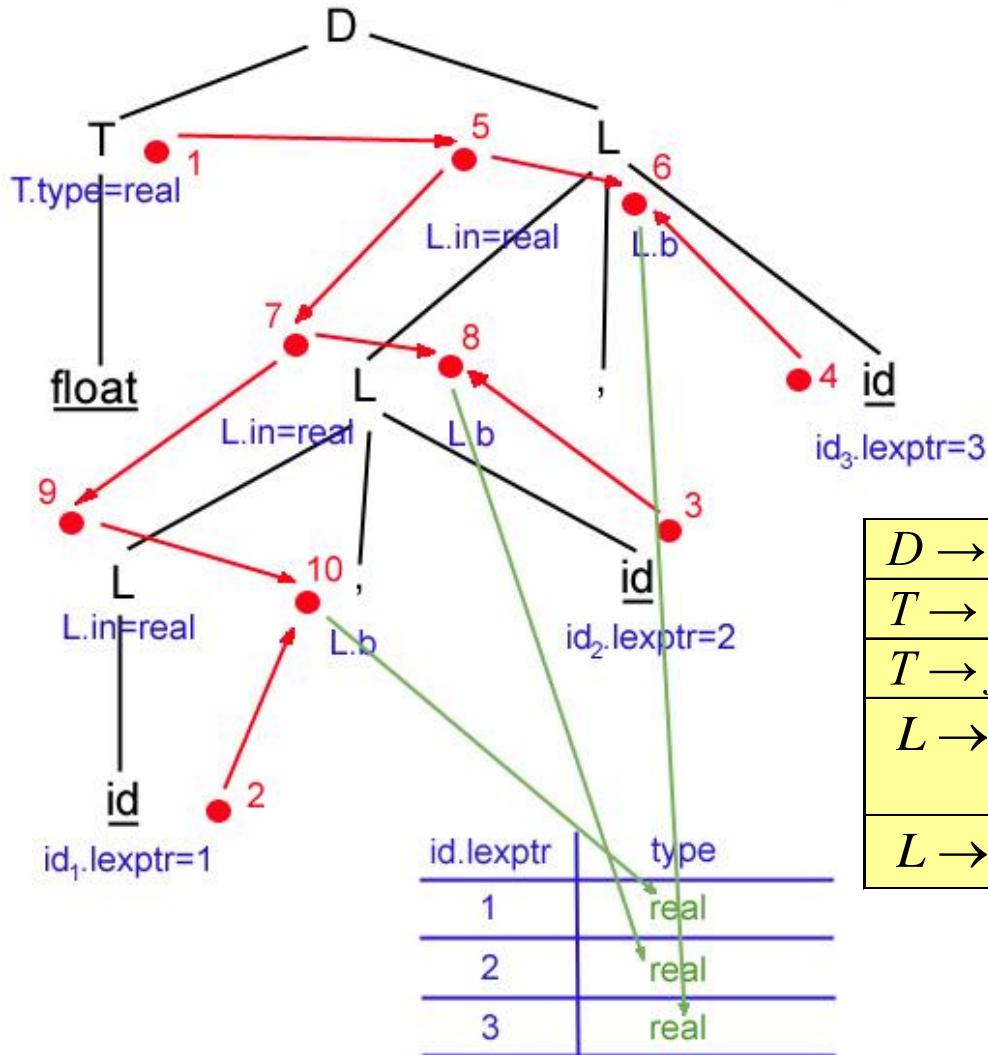
$D \rightarrow TL$	$L.in \leftarrow T.type$
$T \rightarrow \underline{int}$	$T.type \leftarrow integer$
$T \rightarrow \underline{float}$	$T.type \leftarrow real$
$L \rightarrow L_1, \underline{id}$	$L_1.in \leftarrow L.in$ $L.b \leftarrow addtype(\underline{id}.lexptr, L.in)$
$L \rightarrow \underline{id}$	$L.b \leftarrow addtype(\underline{id}.lexptr, L.in)$

Funkcja *addtype()* wstawia do tablicy symboli informację o typie zmiennej

Analizowane słowo: float *id*₁, *id*₂, *id*₃



Przykład – gramatyka atrybutywna z atrybutami dziedzicznymi



$D \rightarrow TL$	$L.in \leftarrow T.type$
$T \rightarrow int$	$T.type \leftarrow integer$
$T \rightarrow float$	$T.type \leftarrow real$
$L \rightarrow L_1, id$	$L_1.in \leftarrow L.in$ $L.b \leftarrow addtype(id.lexptr, L.in)$
$L \rightarrow id$	$L.b \leftarrow addtype(id.lexptr, L.in)$

Gramatyki L-atrybutywne

Definicja

Gramatyka atrybutywna jest gramatyką typu L (gramatyką L-atrybutową), gdy każdy dziedziczony atrybut symbolu $X_j, 1 \leq j \leq n$ występującego w prawej stronie produkcji

$X_0 \rightarrow X_1X_2\dots X_n$ zależy tylko od:

- (1) atrybutów symboli X_1, X_2, \dots, X_{j-1} pojawiających się z lewej strony symbolu X_j w prawej części produkcji,
- (2) dziedziczonych atrybutów symbolu X_0 występującego w lewej stronie produkcji.

Każda gramatyka S-atrybutywna jest zarazem gramatyką L-atrybutową.

Gramatyki L-atrybutywne

Dla gramatyk L-atrybutywnych możliwe jest obliczanie atrybutów podczas przeszukiwania drzewa rozbioru syntaktycznego w głąb metodą zejść rekurencyjnych (depth-first), zgodnie z ogólną procedurą DFVISIT:

procedure DFVISIST(n :wierzchołek);

begin

for każdy potomek „ m ” wierzchołka „ n ” od lewej strony do prawej do
 begin

 oblicz atrybuty dziedziczone dla „ m ”;
 DFVISIT(m);

end;

 oblicz atrybuty syntetyzowane dla „ n ”;

end;

Rozpoczęcie przeszukiwania=wywołanie:

 DFVISIT(korzeń_drzewa_rozbioru_syntaktycznego);

Gramatyki L-atrybutywne

Gramatyki L-atrybutywne bazujące na gramatykach LL(1) umożliwiają obliczanie atrybutów równolegle z parsingiem top-down.

Przekształcanie gramatyk S-atrybutywnych w gramatyki L-atrybutywne bez lewej rekursji i przedstawianie ich w postaci schematów tłumaczenia.

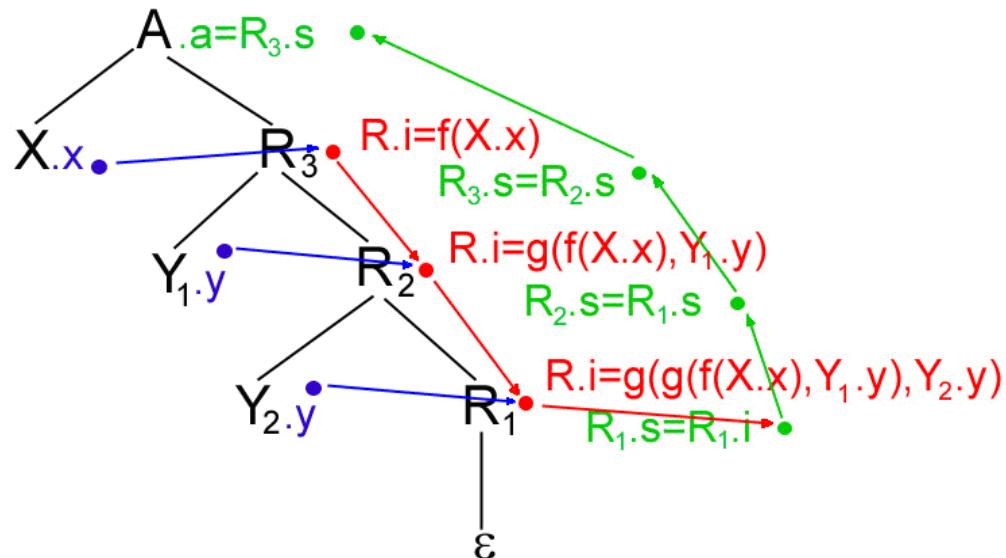
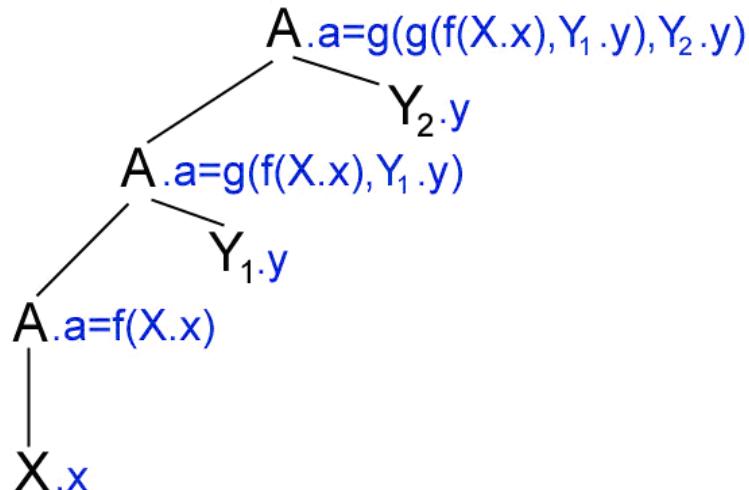
We: gramatyka S-atrybutywna z lewą rekursją, np.:

$$\begin{array}{ll} A \rightarrow A_1 Y & \{A.a \leftarrow g(A_1.a, Y.y)\} \\ A \rightarrow X & \{A.a \leftarrow f(X.x)\} \end{array}$$

Wy: gramatyka L-atrybutywna bez lewej rekursji w postaci schematu tłumaczenia:

$$\begin{array}{ll} A \rightarrow X & \{R.i \leftarrow f(X.x)\} \\ R \rightarrow Y & \{R_1.i \leftarrow g(R.i, Y.y)\} \\ R \rightarrow \varepsilon & \{R.s \leftarrow R.i\} \\ & \{A.a \leftarrow R.s\} \\ & \{R.s \leftarrow R_1.s\} \end{array}$$

Gramatyki L-atrybutywne



We: gramatyka S-atrybutywna z lewą rekursją, np.:

$$A \rightarrow A_1 Y \quad \{A.a \leftarrow g(A_1.a, Y.y)\}$$

$$A \rightarrow X \quad \{A.a \leftarrow f(X.x)\}$$

Wy: gramatyka L-atrybutywna bez lewej rekursji w postaci schematu tłumaczenia:

$$A \rightarrow X \quad \{R.i \leftarrow f(X.x)\} \quad R \{A.a \leftarrow R.s\}$$

$$R \rightarrow Y \quad \{R_1.i \leftarrow g(R.i, Y.y)\} \quad R_1 \{R.s \leftarrow R_1.s\}$$

$$R \rightarrow \epsilon \quad \{R.s \leftarrow R.i\}$$

Przykład – obliczanie wartości wyrażeń

$$E \rightarrow E_1 + T$$

$$\{E.val \leftarrow E_1.val + T.val\}$$

$$E \rightarrow E_1 - T$$

$$\{E.val \leftarrow E_1.val - T.val\}$$

$$E \rightarrow T$$

$$\{E.val \leftarrow T.val\}$$

$$T \rightarrow (E)$$

$$\{T.val \leftarrow E.val\}$$

$$T \rightarrow \underline{num}$$

$$\{T.val \leftarrow \underline{num}.val\}$$

Po usunięciu lewej rekursji:

$$E \rightarrow TR$$

$$R \rightarrow +TR$$

$$R \rightarrow -TR$$

$$R \rightarrow \varepsilon$$

$$T \rightarrow (E)$$

$$T \rightarrow \underline{num}$$

Przykład – obliczanie wartości wyrażeń

Po usunięciu lewej rekursji:

$$E \rightarrow TR$$

$$R \rightarrow +TR$$

$$R \rightarrow -TR$$

$$R \rightarrow \varepsilon$$

$$T \rightarrow (E)$$

$$T \rightarrow \underline{\text{num}}$$

i po przeróbce na gramatykę L-atrybutywną uzyskujemy schemat tłumaczenia:

$$E \rightarrow T \quad \{R.i \leftarrow T.val\} \quad R \quad \{E.val \leftarrow R.s\}$$

$$R \rightarrow +T \quad \{R_1.i \leftarrow R.i + T.val\} \quad R_1 \quad \{R.s \leftarrow R_1.s\}$$

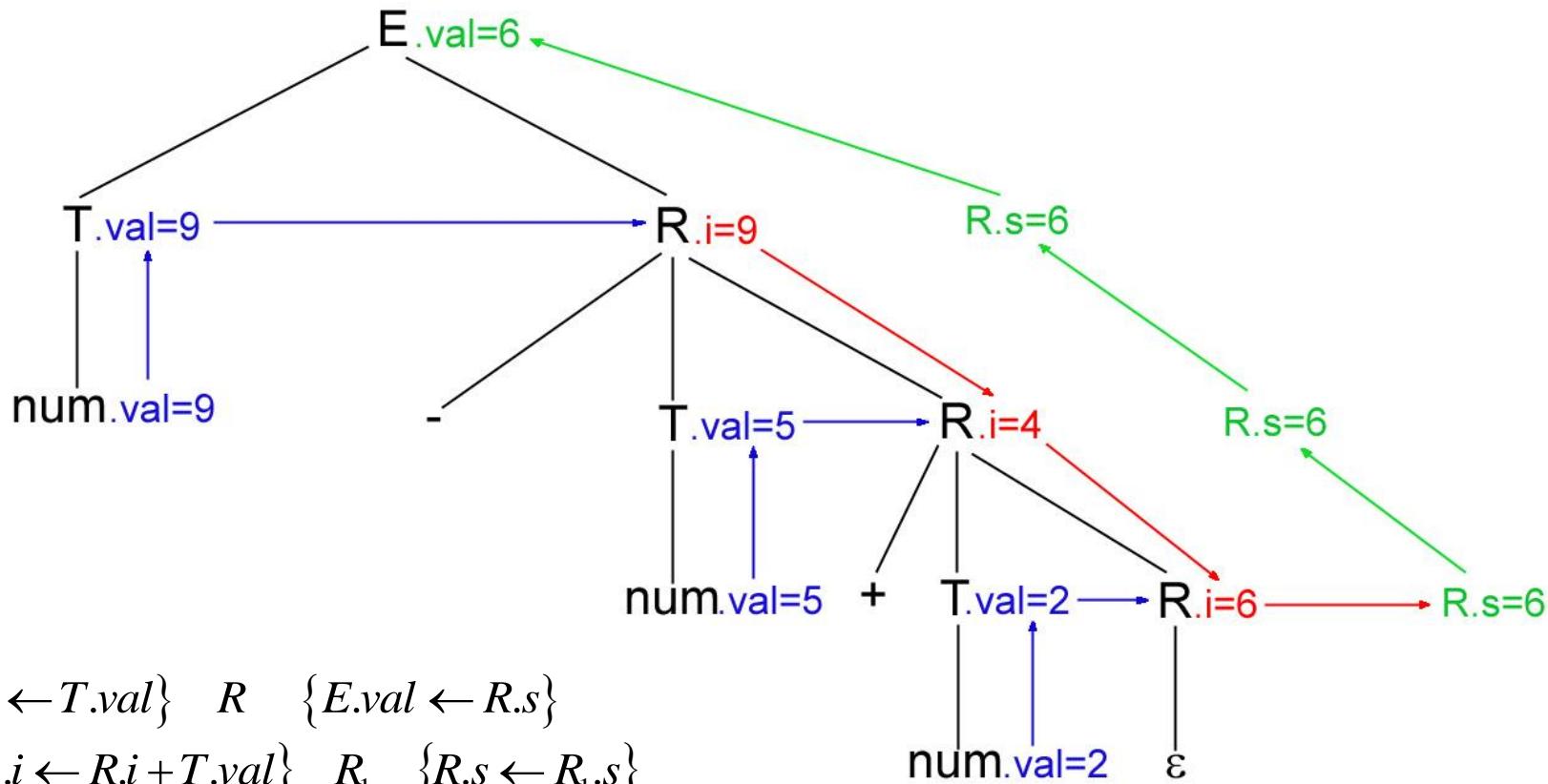
$$R \rightarrow -T \quad \{R_1.i \leftarrow R.i - T.val\} \quad R_1 \quad \{R.s \leftarrow R_1.s\}$$

$$R \rightarrow \varepsilon \quad \{R.s \leftarrow R.i\}$$

$$T \rightarrow (E) \quad \{T.val \leftarrow E.val\}$$

$$T \rightarrow \underline{\text{num}} \quad \{T.val \leftarrow \underline{\text{num}.val}\}$$

Przykład – obliczanie wartości wyrażeń



$$E \rightarrow T \quad \{R.i \leftarrow T.\text{val}\} \quad R \quad \{E.\text{val} \leftarrow R.s\}$$

$$R \rightarrow +T \quad \{R_1.i \leftarrow R.i + T.\text{val}\} \quad R_1 \quad \{R.s \leftarrow R_1.s\}$$

$$R \rightarrow -T \quad \{R_1.i \leftarrow R.i - T.\text{val}\} \quad R_1 \quad \{R.s \leftarrow R_1.s\}$$

$$R \rightarrow \varepsilon \quad \{R.s \leftarrow R.i\}$$

$$T \rightarrow (E) \quad \{T.\text{val} \leftarrow E.\text{val}\}$$

$$T \rightarrow \underline{\text{num}} \quad \{T.\text{val} \leftarrow \underline{\text{num}.val}\}$$

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu rekurencyjnego

Algorytm tworzenia funkcji rekurencyjnych dla obliczania atrybutów gramatyki L-atrybutywnej bazującej na gramatyce LL(1) (obliczanie podczas parsingu)

We: schemat tłumaczenia dla LL(1) gramatyki typu L-atrybutywnego;
tablica parsera LL(1)

Wy: fragment kodu translatora.

1. Dla każdego nieterminala $A \in V$ konstruuje się funkcję o nazwie A . Każdemu atrybutowi dziedziczonemu symbolu nieterminalnemu A odpowiadać ma jeden formalny parametr.

Funkcja ma zwracać wartość atrybutu syntetyzowanego dla A . (Przyjęte założenie, że A posiada tylko jeden atrybut syntetyzowany nie powoduje utraty ogólności, gdyż można zwrócić wskaźnik do struktury zawierającej wartości większej liczby atrybutów syntetyzowanych symbolu A). Funkcja powinna zawierać zmienną lokalną dla każdego atrybutu każdego symbolu gramatyki pojawiającego się we wszystkich produkcjach $A \rightarrow \dots$.

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu rekurencyjnego

2. W kodzie funkcji podejmuje się decyzję którą z produkcji typu $A \rightarrow \dots$ użyto w danym wierzchołku na podstawie bieżącego symbolu wejściowego z wykorzystaniem tablicy parsera LL(1). Każdej produkcji $A \rightarrow \dots$ odpowiada odrębna ścieżka w kodzie funkcji.

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu rekurencyjnego

3. W ramach kodu dla produkcji $A \rightarrow \dots$ rozważamy terminale, nieterminala i akcje prawej strony produkcji w takiej kolejności, w jakiej występują one w schemacie tłumaczenia analizowanym od lewej do prawej:
 - (a) dla terminala X z atrybutem (syntetyzowanym) $X.x$ obliczamy ten atrybut, zachowujemy jego wartość w zmiennej lokalnej i przesuwamy wejście (bierzemy następny symbol z wejścia),
 - (b) dla nieterminala B generujemy wywołanie odpowiadającej funkcji:
$$B.s \leftarrow B(b_1, \dots, b_k)$$
 gdzie:
 $B.s$ to zmienna lokalna dla atrybutu syntetyzowanego symbolu B
 $B()$ - wywołanie funkcji dla symbolu B
 b_1, \dots, b_k - zmienne lokalne dla atrybutów dziedziczonych symbolu B
 - (c) kopujemy akcje zastępując odwołania do atrybutów odpowiednimi zmiennymi lokalnymi. Ostatnią akcją będzie obliczenie atrybutu syntetyzowanego nieterminala A stojącego po lewej stronie produkcji $A \rightarrow \dots$. Wartość tę funkcja zwraca.

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu rekurencyjnego

Tak skonstruowane funkcje nie uwzględniają kontroli sytuacji błędnych (błędna postać ciągu wejściowego).

Rozpoczęcie parsingu i tłumaczenia następuje poprzez wywołanie funkcji dla symbolu początkowego gramatyki.

Przykład – obliczanie wartości wyrażeń

	<u>num</u>	()	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$				
R			$R \rightarrow \epsilon$	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \epsilon$
T	$T \rightarrow \underline{num}$	$T \rightarrow (E)$				

$$E \rightarrow T \{R.i \leftarrow T.val\} \quad R \quad \{E.val \leftarrow R.s\}$$

```

function E : val;      /*zwraca E.val*/
    var T_val_loc, R_i_loc, R_s_loc : val;
begin
    if (lookahead=num) or (lookahead=')' then
        begin /*produkcja E → TR*/
            T_val_loc := T;
            R_i_loc := T_val_loc;
            R_s_loc := R(R_i_loc);
            E := R_s_loc;
        end
    else error;
end;

```

Przykład – obliczanie wartości wyrażeń

	<u>num</u>	()	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$				
R			$R \rightarrow \epsilon$	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \epsilon$
T	$T \rightarrow num$	$T \rightarrow (E)$				

$R \rightarrow +T \{R_i.i \leftarrow R.i + T.val\} \quad R_i \quad \{R.s \leftarrow R_i.s\}$

```

function R(R_i : val) : val; /*zwroca R.s*/
    var R_i_loc, R_s_loc, T_val_loc : val;
begin
    if (lookahead = '+') then /* $R \rightarrow +TR$ */
        begin
            match (lookahead);
            T_val_loc:=T;
            R_i_loc:=R_i+T_val_loc;
            R_s_loc:=R(R_i_loc);
            R:=R_s_loc;
        end
        .....
    
```

Przykład – obliczanie wartości wyrażeń

	<u>num</u>	()	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$				
R			$R \rightarrow \epsilon$	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \epsilon$
T	$T \rightarrow num$	$T \rightarrow (E)$				

$R \rightarrow -T \{R_i.i \leftarrow R.i - T.val\}$ $R_i \quad \{R.s \leftarrow R_i.s\}$

$R \rightarrow \epsilon \{R.s \leftarrow R.i\}$

```

function R(R_i : val) : val; /*zwraca R.s*/
    var R_i_loc, R_s_loc, T_val_loc : val;
begin if (lookahead = '+') then ..... /* $R \rightarrow +TR$ */
        else if (lookahead='-) then /* $R \rightarrow -TR$  */
            begin
                match (lookahead);
                T_val_loc:=T;
                R_i_loc:=R_i-T_val_loc;
                R_s_loc:=R(R_i_loc);
                R:=R_s_loc;
            end
        else if (lookahead='$') or (lookahead=')' then /* $R \rightarrow \epsilon$  */
            R:=R_i
        else error;
end;

```

Przykład – obliczanie wartości wyrażeń

	<u>num</u>	()	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$				
R			$R \rightarrow \epsilon$	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \epsilon$
T	$T \rightarrow \underline{num}$	$T \rightarrow (E)$				

$T \rightarrow (E) \quad \{T.val \leftarrow E.val\}$

$T \rightarrow \underline{num} \quad \{T.val \leftarrow \underline{num}.val\}$

```

function T : val;      /*zwraca T.val*/
    var T_val_loc, E_val_loc, num_val_loc : val;
begin
    if (lookahead = num) then /* $T \rightarrow \underline{num}$ */
        begin
            T_val_loc := value (num);
            match (num);
            T := T_val_loc;
        end
    else if (lookahead='(') then ..... /*  $T \rightarrow (E)$  */
    else error;
end;

```

Przykład – obliczanie wartości wyrażeń

	<u>num</u>	()	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$				
R			$R \rightarrow \epsilon$	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \epsilon$
T	$T \rightarrow \underline{num}$	$T \rightarrow (E)$				

$T \rightarrow (E) \quad \{T.val \leftarrow E.val\}$

$T \rightarrow \underline{num} \quad \{T.val \leftarrow \underline{num}.val\}$

```

function T : val;      /*zwraca T.val*/
    var T_val_loc, E_val_loc, num_val_loc : val;
begin
    if (lookahead=num) then ..... /* $T \rightarrow \underline{num}$ */
    else if (lookahead='(') then /*  $T \rightarrow (E)$  */
        begin
            match ('(');
            E_val_loc:=E;
            match (')');
            T:=E_val_loc;
        end
    else error;
end;

```

Przykład – obliczanie wartości wyrażeń

var lookahead : token; /*zmienna globalna – token podglądzany na wejściu*/

procedure match (t:token); /*realizacja operacji „pop”*/

begin

if lookahead = t then

 lookahead := next_token

else

 error;

end;

function next_token : token; /*czyta i zwraca token z wejścia; niszczy przeczytany token*/

begin ... end;

function value (t : token) : val; /*zwraca wartość tokenu „num”*/

begin ... end;

Przykład – obliczanie wartości wyrażeń

procedure error;
begin ... end;

/*sygnalizacja i obsługa błędów*/

function Translate : val

/*zwraca wynik interpretacji czyli wartość analizowanego wyrażenia*/

begin

... /*czynności wstępne*/

lookahead := next_token;

/*czytanie pierwszego tokenu*/

Translate := E;

/*wywołanie funkcji rekurencyjnej dla symbolu początkowego gramatyki*/

end;



AGH

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu predykcyjnego (idea algorytmu)

Algorithm

LL(1) L-Attributed Evaluation

```
FOR each predicted production  $X_0 \rightarrow X_1X_2\dots X_n$ .  
  Push  $X_0$ 's inherited attributes onto semantic stack.  
  Push  $X_1$ 's inherited attributes onto semantic stack.  
  Parse  $X_1$ , then push  $X_1$ 's synthesized attributes onto  
    semantic stack.  
  Push  $X_2$ 's inherited attributes onto semantic stack.  
  Parse  $X_2$ , then push  $X_2$ 's synthesized attributes onto  
    semantic stack ...  
  Push  $X_n$ 's inherited attributes onto semantic stack.  
  Parse  $X_n$ , then push  $X_n$ 's synthesized attributes onto  
    semantic stack.  
  Pop attributes of  $X_1X_2,\dots,X_n$ .  
  Push synthesized attributes of  $X_0$ .  
ENDFOR
```

Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu predykcyjnego (przykład)

Gramatyka L-atrybutywna bez lewej rekursji w postaci schematu tłumaczenia:

$$A \rightarrow X \{R.i \leftarrow f(X.x)\} R \{A.s \leftarrow R.s\}$$

$$R \rightarrow Y \{R_1.i \leftarrow g(R.i, Y.y)\} R_1 \{R.s \leftarrow R_1.s\}$$

$$R \rightarrow \varepsilon \{R.s \leftarrow R.i\}$$

Zamieniamy reguły obliczające atrybuty w [**<symbole_akcji>**](#). Będą one umieszczane na stosie parsera jak terminale czy nieterminale i kojarzone z odpowiednimi regułami obliczającymi atrybuty. Przetwarzanie symboli akcji polega na obliczeniu atrybutu definiowanego przez regułę, zdjęciu ze stosu atrybutów niepotrzebnych argumentów reguły i położeniu na stosie atrybutów wartości obliczonego atrybutu.

$$A \rightarrow X <\!\!R.i\!\!> R <\!\!A.s\!\!>$$

$$R \rightarrow Y <\!\!R_1.i\!\!> R_1 <\!\!R.s\!\!>$$

$$R \rightarrow \varepsilon <\!\!R.s\!\!>$$

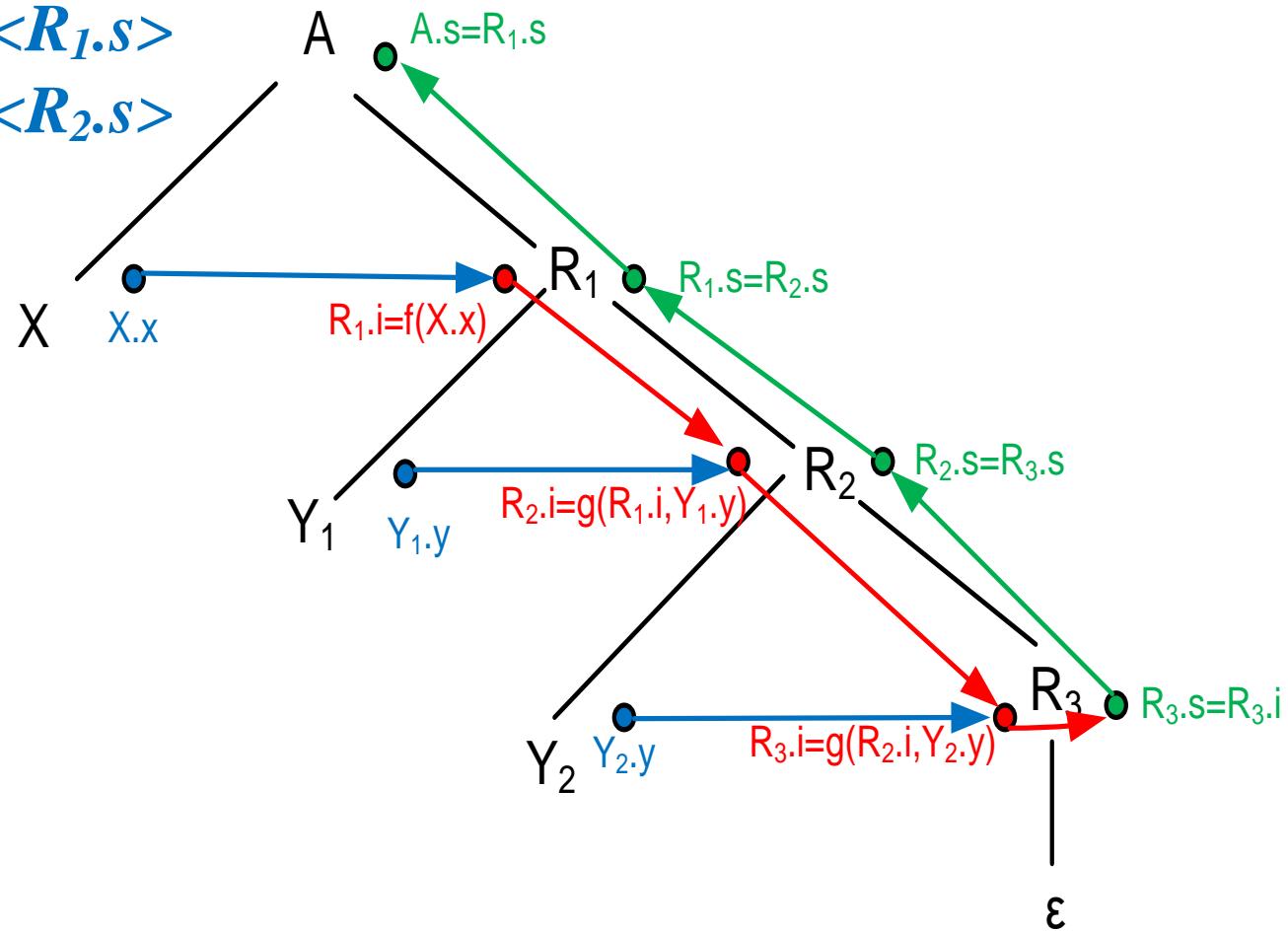
Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu predykcyjnego - przykład

$$A \rightarrow X < R_1.i > R_1 < A.s >$$

$$R_1 \rightarrow Y_1 < R_2.i > R_2 < R_1.s >$$

$$R_2 \rightarrow Y_2 < R_3.i > R_3 < R_2.s >$$

$$R_3 \rightarrow \varepsilon < R_3.s >$$



Algorytm obliczania atrybutów w gramatyce L-atrybutywnej podczas parsingu predykcyjnego - przykład c. d.

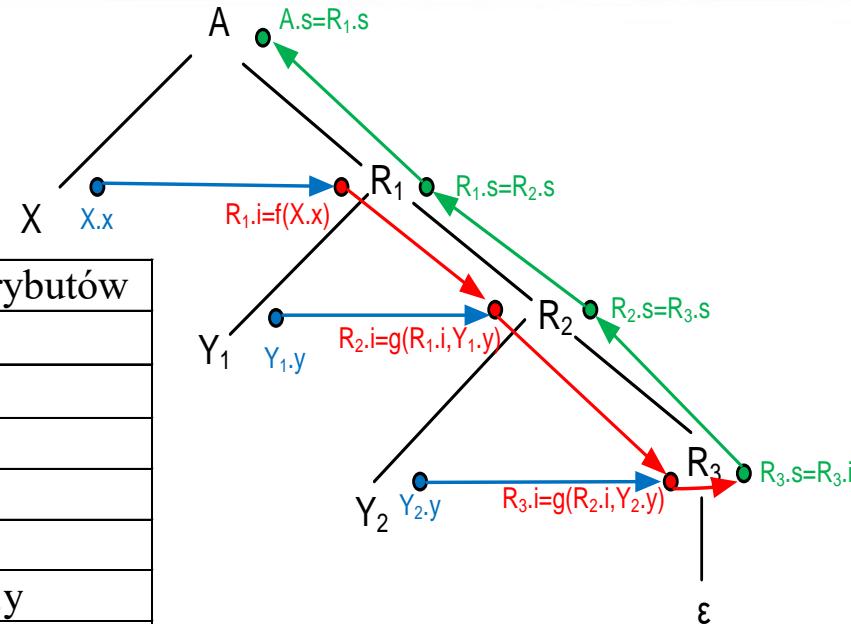
$$A \rightarrow X <R_1.i> R_1 < A.s >$$

$$R_1 \rightarrow Y_1 <R_2.i> R_2 < R_1.s >$$

$$R_2 \rightarrow Y_2 <R_3.i> R_3 < R_2.s >$$

$$R_3 \rightarrow \varepsilon < R_3.s >$$

Stos parsera	Wejście	Stos atrybutów
A	XY ₁ Y ₂ \$	ε
<A.s> R ₁ <R _{1.i} > X	X Y ₁ Y ₂ \$	ε
<A.s> R ₁ <R _{1.i} >	Y ₁ Y ₂ \$	X.x
<A.s> R ₁	Y ₁ Y ₂ \$	R _{1.i}
<A.s> <R _{1.s} > R ₂ <R _{2.i} > Y ₁	Y ₁ Y ₂ \$	R _{1.i}
<A.s> <R _{1.s} > R ₂ <R _{2.i} >	Y ₂ \$	R _{1.i} Y _{1.y}
<A.s> <R _{1.s} > R ₂	Y ₂ \$	R _{2.i}
<A.s> <R _{1.s} > <R _{2.s} > R ₃ <R _{3.i} > Y ₂	Y ₂ \$	R _{2.i}
<A.s> <R _{1.s} > <R _{2.s} > R ₃ <R _{3.i} >	\$	R _{2.i} Y _{2.y}
<A.s> <R _{1.s} > <R _{2.s} > R ₃	\$	R _{3.i}
<A.s> <R _{1.s} > <R _{2.s} > <R _{3.s} >	\$	R _{3.i}
<A.s> <R _{1.s} > <R _{2.s} >	\$	R _{3.s}
<A.s> <R _{1.s} >	\$	R _{2.s}
<A.s>	\$	R _{1.s}
ε	\$	A.s



Translacja bottom-up dla gramatyk L-atrybutywnych

Metoda poniższa może być stosowana do tych wszystkich gramatyk, co poprzednio (tzn. dla L-atrybutywnych definicji opartych na gramatykach LL(1)), a także dla wielu (choć nie wszystkich) gramatyk L-atrybutywnych opartych na gramatykach LR(1).

Na przykładzie zostanie przedstawione likwidowanie „akcji wewnętrznych” w schematach tłumaczenia poprzez wprowadzenie dodatkowych nieterminali.

Przykład: wypisywanie wyrażeń w odwrotnej notacji polskiej

Gramatyka atrybutywna typu L:

$$E \rightarrow TR$$

$$R \rightarrow +T \quad \{R_I.i \leftarrow \text{print}('+) \} \quad R_I$$

$$R \rightarrow -T \quad \{R_I.i \leftarrow \text{print}('-) \} \quad R_I$$

$$R \rightarrow \varepsilon$$

$$T \rightarrow \underline{\text{num}} \quad \{T.s \leftarrow \text{print}(\text{text}(\underline{\text{num}}.\text{val}))\}$$

Jest transformowana do postaci typu S:

$$E \xrightarrow{(1)} TR$$

$$R \xrightarrow{(2)} +TMR \mid -TNR \mid \varepsilon \quad \xrightarrow{(3)} TMR \mid NTR \mid \varepsilon \quad \xrightarrow{(4)} TMR \mid NTR \mid \varepsilon$$

$$T \xrightarrow{(5)} \underline{\text{num}} \quad \{T.s \leftarrow \text{print}(\text{text}(\underline{\text{num}}.\text{val}))\}$$

$$M \xrightarrow{(6)} \underline{\varepsilon} \quad \{M.s \leftarrow \text{print}('+')\}$$

$$N \xrightarrow{(7)} \underline{\varepsilon} \quad \{N.s \leftarrow \text{print}(' -')\}$$

Przykład: wypisywanie wyrażeń w odwrotnej notacji polskiej

state	we	wy	
<u>num</u>	9-5+2		
T	-5+2	9	<i>shift</i>
T-	-5+2	9	<i>red 5</i>
T- <u>num</u>	5+2	9	<i>shift</i>
T-T	+2	95	<i>shift</i>
T-TN	+2	95-	<i>red 5</i>
T-TN+	2	95-	<i>red 7</i>
T-TN+ <u>num</u>	ϵ	95-	<i>shift</i>
T-TN+T	ϵ	95-2	<i>shift</i>
T-TN+TM	ϵ	95-2+	<i>red 5</i>
T-TN+TMR	ϵ	95-2+	<i>red 6</i>
T-TNR	ϵ	95-2+	<i>red 4</i>
TR	ϵ	95-2+	<i>red 2</i>
E	ϵ	95-2+	<i>red 3</i>
			<i>acc</i>

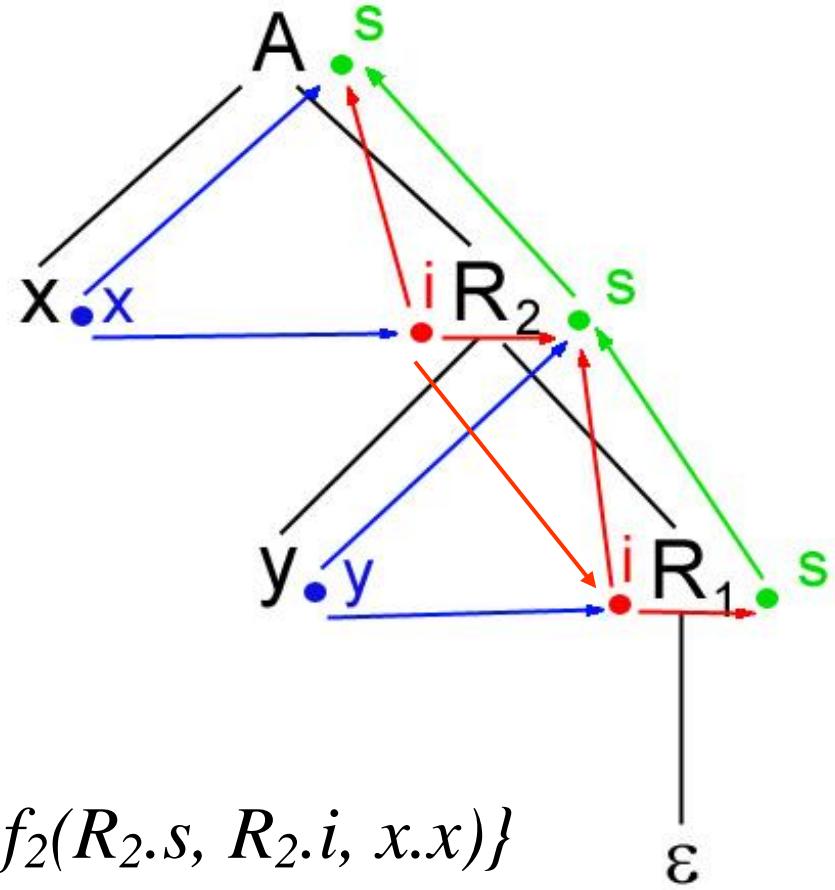
(1) $E \rightarrow TR$
 (2) $R \rightarrow +TMR$
 (3) $R \rightarrow -TNR$
 (4) $R \rightarrow \epsilon$
 (5) $T \rightarrow \underline{num}$ $\{print(text(\underline{num}.val))\}$
 (6) $M \rightarrow \epsilon$ $\{print ('+')\}$
 (7) $N \rightarrow \epsilon$ $\{print ('-')\}$

Przykład – tworzenie nowych nieterminali dla atrybutów dziedziczonych

$$A \rightarrow xR$$

$$R \rightarrow yR$$

$$R \rightarrow \epsilon$$



$$A \rightarrow x \{R_2.i \leftarrow f_1(x.x)\} R_2 \{A.s \leftarrow f_2(R_2.s, R_2.i, x.x)\}$$

$$R_2 \rightarrow y \{R_1.i \leftarrow f_3(y.y, R_2.i)\} R_1 \{R_2.s \leftarrow f_4(R_1.s, R_1.i, R_2.i, y.y)\}$$

$$R_1 \rightarrow \epsilon \{R_1.s \leftarrow f_5(R_1.i)\}$$

Przykład – tworzenie nowych nieterminali dla atrybutów dziedziczonych

Gramatyka:

$$A \rightarrow x \{R_2.i \leftarrow f_1(x.x)\} R_2 \{A.s \leftarrow f_2(R_2.s, R_2.i, x.x)\}$$

$$R_2 \rightarrow y \{R_1.i \leftarrow f_3(y.y, R_2.i)\} R_1 \{R_2.s \leftarrow f_4(R_1.s, R_1.i, R_2.i, y.y)\}$$

$$R_1 \rightarrow \varepsilon \{R_1.s \leftarrow f_5(R_1.i)\}$$

jest transformowana do postaci:

$$(1) \quad A \rightarrow x M R_2 \left\{ A.s \leftarrow f_2(R_2.s, R_2.i, x.x) \right\}$$

$$(2) \quad R_2 \rightarrow y N R_1 \left\{ R_2.s \leftarrow f_4(R_1.s, R_1.i, R_2.i, y.y) \right\}$$

$$(3) \quad R_1 \rightarrow \varepsilon \left\{ R_1.s \leftarrow f_5(R_1.i) \right\}$$

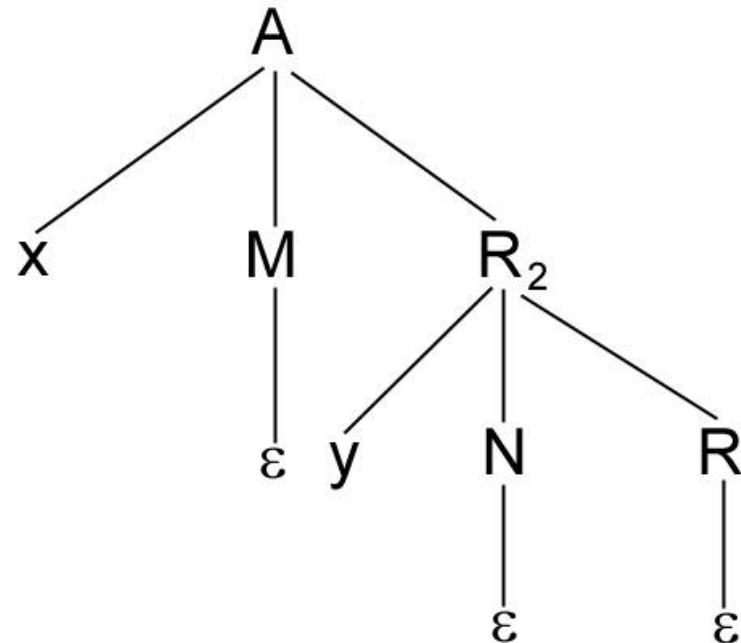
$$(4) \quad M \rightarrow \varepsilon \left\{ R_2.i \leftarrow f_1(x.x) \right\}$$

$$(5) \quad N \rightarrow \varepsilon \left\{ R_1.i \leftarrow f_3(y.y, R_2.i) \right\}$$

Przykład – tworzenie nowych nieterminali dla atrybutów dziedziczonych

- (1) $A \rightarrow x M R_2 \{ A.s \leftarrow f_2(R_2.s, R_2.i, x.x) \}$
- (2) $R_2 \rightarrow y N R_1 \{ R_2.s \leftarrow f_4(R_1.s, R_1.i, R_2.i, y.y) \}$
- (3) $R_1 \rightarrow \epsilon \{ R_1.s \leftarrow f_5(R_1.i) \}$
- (4) $M \rightarrow \epsilon \{ R_2.i \leftarrow f_1(x.x) \}$
- (5) $N \rightarrow \epsilon \{ R_1.i \leftarrow f_3(y.y, R_2.i) \}$

state	attr	we	
ϵ	ϵ	xy	shift
x	$x.x$	y	red 4
$x M$	$x.x R_2.i$	y	shift
$x M y$	$x.x R_2.i y.y$	ϵ	red 5
$x M y N$	$x.x R_2.i y.y R_1.i$	ϵ	red 3
$x M y N R_1$	$x.x R_2.i y.y R_1.i R_1.s$	ϵ	red 2
$x M R_2$	$x.x R_2.i R_2.s$	ϵ	red 1
A	$A.s$	ϵ	acc



Przykład – tworzenie nowych nieterminali dla atrybutów dziedziczonych

$$(2) \quad R_2 \rightarrow y \ N \ R_1 \left\{ R_2.s \leftarrow f_4(R_1.s, R_1.i, R_2.i, y.y) \right\}$$

Przed redukcją

R ₁	R ₁ .s	top
N	R ₁ .i	top-1
y	y.y	top-2
M	R ₂ .i	top-3
X	X.X	top-4
state	attr	

ntop:=top-2;
 R₂.s attr[ntop]:=
 R₁.s R₁.i
 f (attr[top], attr[top-1],
 attr[top-3]. attr[top-2]);
 R₂.i y.y

Po redukcji

R ₂	R ₂ .s	ntop
M	R ₂ .i	
X	X.X	
state	attr	

atrybut dziedziczony zawsze
 bezpośrednio pod atrybutem
 syntetyzowanym symbolu R,
 o ile symbol R znajduje się na stosie

Parsing bottom-up dla gramatyk L-atrybutywnych

Ogólny algorytm obliczania atrybutów w trakcie parsingu bottom-up dla gramatyk L-atrybutywnych opartych na gramatykach LL(1) (a także dla niektórych LR(1) nie będących LL(1))

- We: Gramatyka L-atrybutywna oparta na gramatyce LL(1); gramatyka syntaktyczna nie zawiera symbolu początkowego w prawych stronach produkcji.
- Zał: Symbol początkowy gramatyki nie posiada atrybutu dziedziczonego. Każdy symbol gramatyki posiada jeden atrybut syntetyzowany i jeden dziedziczony.
- Wy: Algorytm parsera bottom-up obliczającego atrybuty podczas analizy słowa wejściowego.

Parsing bottom-up dla gramatyk L-atrybutywnych

$A.i$ - atrybut dziedziczony A; $A.s$ - atrybut syntetyzowany A

$X_j.i$ - atrybut dziedziczony X_j ; $X_j.s$ - atrybut syntetyzowany X_j

Każda produkcja: $A \rightarrow X_1X_2\dots X_j\dots X_n$ jest przekształcana do postaci:

$A \rightarrow M_1X_1M_2X_2\dots M_jX_j\dots M_nX_n$

$M_1 \rightarrow \varepsilon$

$M_2 \rightarrow \varepsilon$

.....

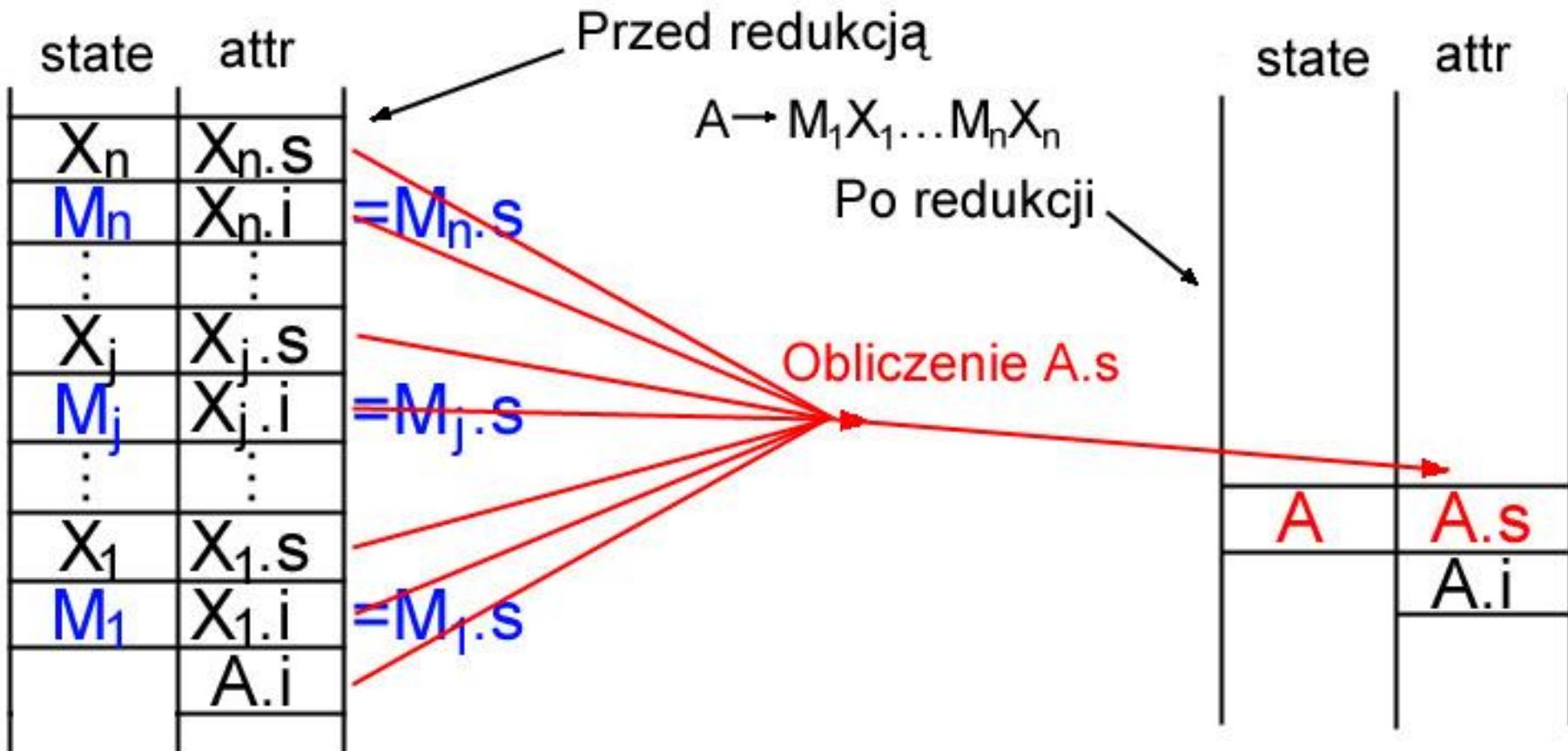
$M_j \rightarrow \varepsilon$

.....

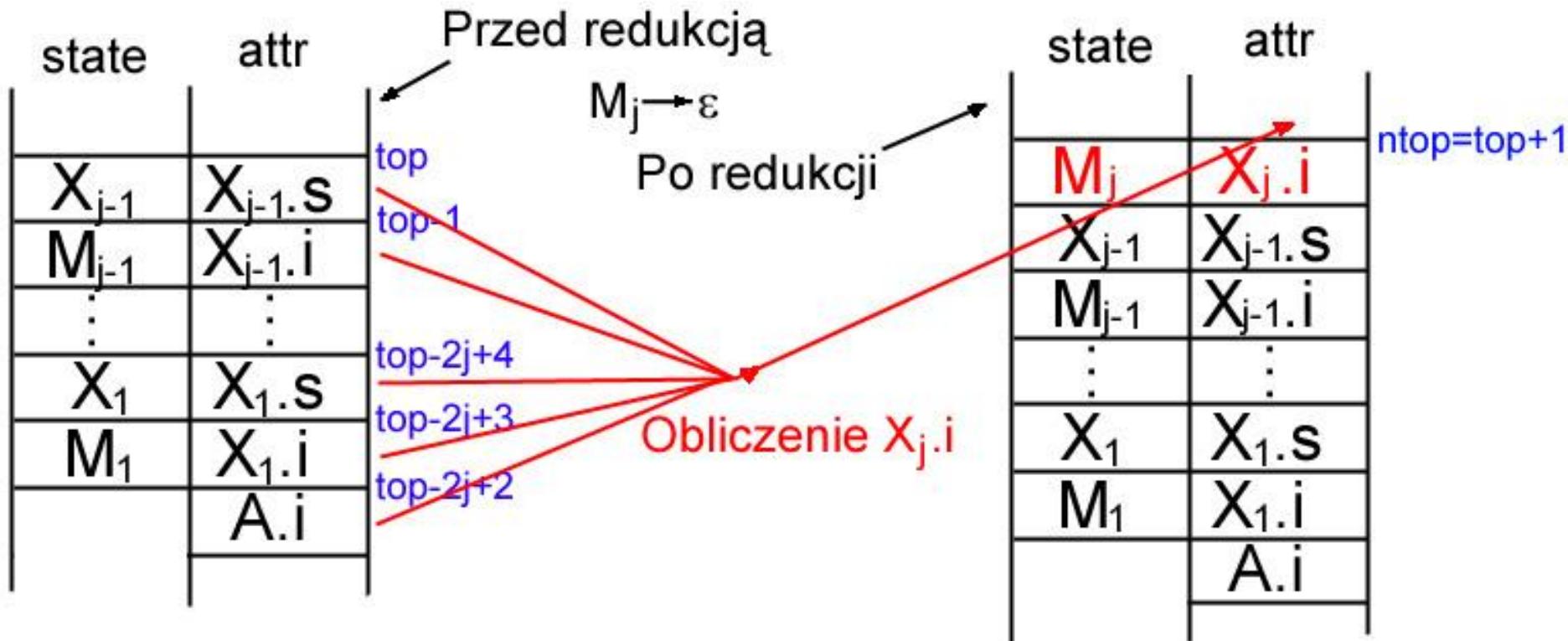
$M_n \rightarrow \varepsilon$

Atrybut dziedziczony $X_j.i$ każdego symbolu X_j jest utożsamiany z atrybutem syntetyzowanym $M_j.s$ dodatkowego nieterminala M_j .

Parsing bottom-up dla gramatyk L-atrybutywnych



Parsing bottom-up dla gramatyk L-atrybutywnych





AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Gramatyki atrybutywne - część 3

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki



Gramatyka zupełna i dobrze zdefiniowana

Definicja:

Gramatyka atrybutywna jest zupełna, jeśli dla wszystkich symboli $X \in (V \cup \Sigma)$ są spełnione następujące warunki:

$$\forall p = (X \rightarrow \chi) \in P : AS(X) \subseteq AF(p)$$

$$\forall q = (Y \rightarrow \mu X v) \in P : AI(X) \subseteq AF(q)$$

$$AS(X) \cup AI(X) = A(X)$$

Ponadto: $AI(S) = \emptyset$

Definicja

Gramatyka atrybutywna jest dobrze zdefiniowana jeśli dla każdego drzewa rozbiórku syntaktycznego słowa z $L(G)$ wszystkie atrybuty są efektywnie obliczalne.

Zbiór (graf) bezpośrednich zależności atrybutów

Definicja

Dla każdej produkcji $p = (X_0 \rightarrow X_1 X_2 \dots X_n) \in P$ zbiorem (relacją, grafem) bezpośrednich zależności atrybutów jest zbiór:

$$DDP(p) = \{(X_i.a \rightarrow X_j.b) : X_j.b \leftarrow f(\dots, X_i.a, \dots) \in R(p)\}$$

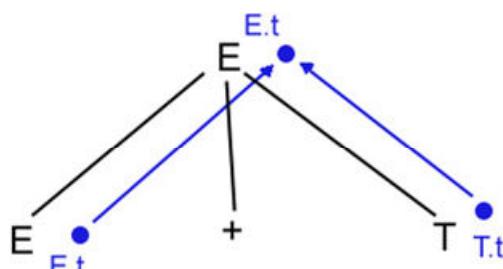
Gramatyka atrybutywna jest lokalnie acykliczna, jeśli graf $DDP(p)$ jest acykliczny dla każdej produkcji $p \in P$.

Graf bezpośrednich zależności atrybutów

Przykład

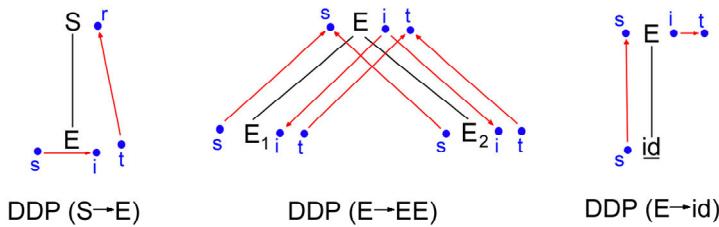
Produkcja $E \rightarrow E_1 + T$ Reguła semantyczna $E.t \leftarrow E_1.t \parallel T.t \parallel '+'$

Graf bezpośrednich zależności atrybutów dla produkcji
 $E \rightarrow E + T$



Przykład – grafy bezpośrednich zależności atrybutów

(1) $S \rightarrow E$	$E.i \leftarrow g(E.s)$ $S.t \leftarrow E.t$
(2) $E \rightarrow E_1 E_2$	$E.s \leftarrow f_s(E_1.s, E_2.s)$ $E_1.i \leftarrow f_{11}(E.i)$ $E_2.i \leftarrow f_{12}(E.i)$ $E.t \leftarrow f_t(E_1.t, E_2.t)$
(3) $E \rightarrow id$	$E.s \leftarrow \underline{id}.s$ $E.t \leftarrow h(E.i)$



Grafy bezpośrednich zależności atrybutów

Zbiory (relacje) bezpośrednich zależności atrybutów $DDP(p)$ dają informacje o własnościach lokalnych gramatyki atrybutywnej. Dla poprawnego obliczenia atrybutów potrzebne są informacje z bezpośredniego „otoczenia” poszczególnych produkcji, czyli informacje o bardziej globalnym charakterze.



Graf zależności

Definicja

Niech S będzie atrybutowanym drzewem rozbiór syntaktycznego słowa z $L(G)$ i niech K_0, \dots, K_n będą wierzchołkami odpowiadającymi zastosowaniu produkcji $p = (X_0 \rightarrow X_1 \dots X_n)$. Piszemy $K_i.a \rightarrow K_j.b$ jeśli $(X_i.a \rightarrow X_j.b) \in DDP(p)$. Zbiór $DT(S) = \{(K_i.a \rightarrow K_j.b)\}$ uwzględniający wszystkie zastosowania produkcji w drzewie S nazwiemy relacją zależności nad drzewem S , a odpowiadający mu graf zorientowany – grafem zależności.

Twierdzenie

Gramatyka atrybutywna jest dobrze zdefiniowana wtedy i tylko wtedy, gdy jest zupełna i graf $DT(S)$ jest acykliczny dla każdego drzewa S rozbiór syntaktycznego odpowiadającego słowu z języka $L(G)$.



Algorytm konstruowania grafu zależności

We: gramatyka atrybutywna $AG = \langle G, A, R \rangle$ oraz drzewo rozbiór syntaktycznego słowa z $L(G)$

Wy: graf zależności atrybutów dla danego drzewa rozbiór syntaktycznego

for każdy węzeł „ n ” w drzewie rozbiór syntaktycznego do
 for każdy atrybut „ a ” symbolu gramatyki znajdującego się w węźle „ n ” do

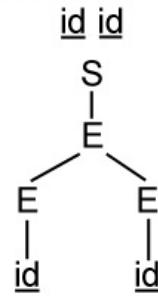
 zbuduj dla „ a ” węzeł w grafie zależności;

for każdy węzeł „ n ” w drzewie rozbiór syntaktycznego do
 for każda reguła semantyczna $b \leftarrow f(c_1, \dots, c_k)$ związana z produkcją stosowaną w węźle „ n ” do
 for $i := 1$ to k do
 zbuduj połączenie z węzła odpowiadającego „ c_i ” do węzła odpowiadającego „ b ”;

Przykład – konstruowanie grafu zależności (1)

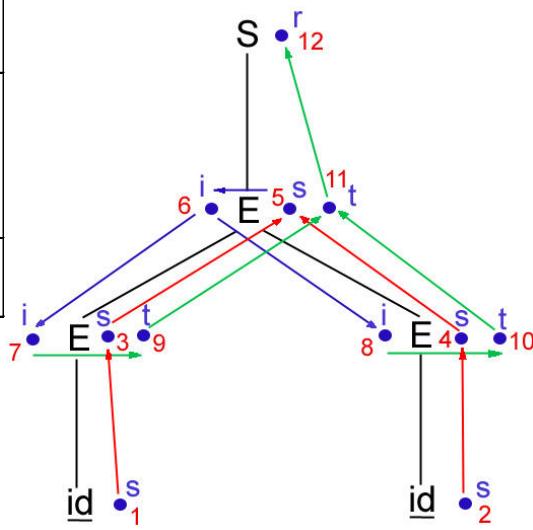
$S \rightarrow E$	$E.i \leftarrow g(E.s)$ $S.r \leftarrow E.t$
$E \rightarrow E_1 E_2$	$E.s \leftarrow f_s(E_1.s, E_2.s)$ $E_1.i \leftarrow f_{il}(E.i)$ $E_2.i \leftarrow f_{i2}(E.i)$ $E.t \leftarrow f_t(E_1.t, E_2.t)$
$E \rightarrow \underline{id}$	$E.s \leftarrow \underline{id}.s$ $E.t \leftarrow h(E.i)$

Analizowane słowo:



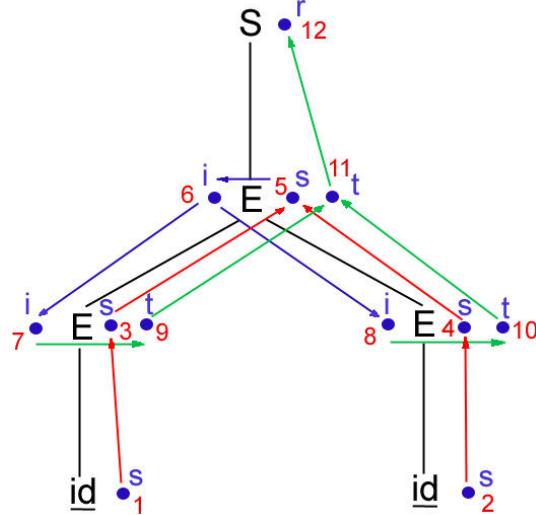
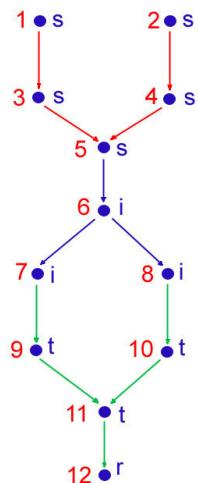
Przykład – konstruowanie grafu zależności (2)

$S \rightarrow E$	$E.i \leftarrow g(E.s)$ $S.r \leftarrow E.t$
$E \rightarrow E_1 E_2$	$E.s \leftarrow f_s(E_1.s, E_2.s)$ $E_1.i \leftarrow f_{il}(E.i)$ $E_2.i \leftarrow f_{i2}(E.i)$ $E.t \leftarrow f_t(E_1.t, E_2.t)$
$E \rightarrow \underline{id}$	$E.s \leftarrow \underline{id}.s$ $E.t \leftarrow h(E.i)$



Przykład – konstruowanie grafu zależności (3)

Obliczanie atrybutów



Uporządkowanie topologiczne skierowanego grafu acyklicznego

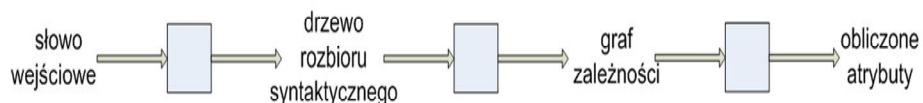
Uporządkowanie topologiczne zorientowanego grafu acyklicznego jest to każda permutacja m_1, m_2, \dots, m_k wierzchołków tego grafu taka, że każda krawędź grafu biegnie od wierzchołka wcześniejszego w tym uporządkowaniu do węzła późniejszego. Np. jeśli $m_i \rightarrow m_j$ jest krawędzią biegącą od wierzchołka m_i do wierzchołka m_j , to m_i pojawia się w uporządkowaniu wcześniej niż m_j .

Porządek topologiczny w grafie zależności daje właściwą kolejność obliczania atrybutów w drzewie rozbiórku syntaktycznego.

Metody obliczania reguł semantycznych i atrybutów

1. Metody oparte o drzewa rozbiórku syntaktycznego

Kolejność obliczania atrybutów jest ustalana w czasie komplikacji na podstawie drzewa rozbiórku syntaktycznego poprzez skonstruowanie grafu zależności dla każdego wejściowego słowa.



Metoda ta jest nieskuteczna tylko w przypadku gdy graf zależności dla rozważanego słowa wejściowego posiada cykle.

Metody obliczania reguł semantycznych i atrybutów

2. Metody bazujące na regułach semantycznych

Kolejność obliczania atrybutów jest rozstrzygana na etapie konstrukcji kompilatora. Analizowane są reguły semantyczne związane z poszczególnymi produkcjami (automatycznie lub ręcznie). Dla każdej produkcji kolejność obliczania atrybutów związanych z tą produkcją jest zdeterminowana już podczas konstrukcji kompilatora.



Metody obliczania reguł semantycznych i atrybutów

3. Metody nie uwzględniające bezpośrednio reguł sematycznych

Kolejność obliczania atrybutów jest ustalana bez bezpośredniego rozważania reguł semantycznych. Często tłumaczenie (analiza semantyczna) ma miejsce równolegle z analizą syntaktyczną; kolejność obliczania atrybutów jest wymuszana przez parser. Taka zasada postępowania ogranicza klasę gramatyk atrybutywnych, które mogą opisywać dokonywane tłumaczenie.



Obliczanie atrybutów w atrybutowanym drzewie rozbiórku syntaktycznego

Problem: dana gramatyka atrybutywna AG i drzewo rozbiórku syntaktycznego pewnego słowa z języka $L(G)$. Obliczyć atrybuty symboli w wierzchołkach drzewa rozbiórku syntaktycznego.
Założenie: gramatyka atrybutywna jest dobrze zdefiniowana.

Przypomnienie:

Relacja \leq określona na zbiorze A spełniająca warunki:

- (I) $\forall a \in A : x \leq x$ (zwrotność)
- (II) $(x \leq y) \wedge (y \leq x) \Rightarrow x = y$
- (III) $(x \leq y) \wedge (y \leq z) \Rightarrow x \leq z$ (przechodniość)

Nazywana jest relacją porządkującą zbiór A.

(Dawniej relację taką nazywano relacją częściowego porządku.)

Obliczanie atrybutów w atrybutowanym drzewie rozbiór syntaktycznego

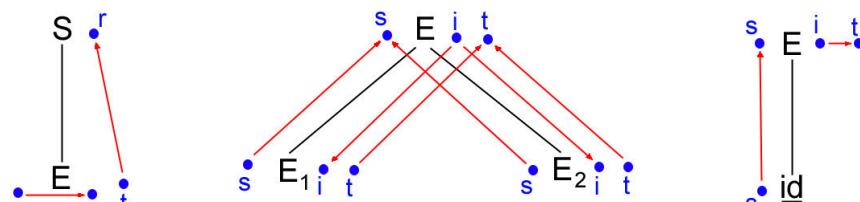
1. Dla każdego $X \in (V \cup \Sigma)$ skonstruować relację \mapsto porządkującą zbiór jego atrybutów $A(X)$ tak, aby $(X.a \mapsto X.b) \Rightarrow$ wartość $X.a$ musi być znana dla obliczenia wartości $X.b$

Ogólny algorytm budowania relacji \mapsto nie jest objęty niniejszym wykładem. Dalsze rozważania mają sprawdzić trafność podjętej decyzji dotyczącej relacji \mapsto .

2. Dla każdej produkcji $p \in P$ zbudować zbiór (relację) zależności atrybutów $DP(p)$

Przykład

(1) $S \rightarrow E$	$E.i \leftarrow g(E.s)$ $S.r \leftarrow E.t$
(2) $E \rightarrow E_1 E_2$	$E.s \leftarrow f_s(E_1.s, E_2.s)$ $E_1.i \leftarrow f_{i1}(E.i)$ $E_2.i \leftarrow f_{i2}(E.i)$ $E.t \leftarrow f_t(E_1.t, E_2.t)$
(3) $E \rightarrow \underline{id}$	$E.s \leftarrow \underline{id}.s$ $E.t \leftarrow h(E.i)$



DDP ($S \rightarrow E$)

DDP ($E \rightarrow EE$)

DDP ($E \rightarrow \underline{id}$)



Obliczanie atrybutów w atrybutowanym drzewie rozbioru syntaktycznego

Zbiory (relacje) bezpośrednich zależności atrybutów $DDP(p)$ dają informacje o własnościach lokalnych gramatyki atrybutywnej. Dla poprawnego obliczenia atrybutów potrzebne są informacje z bezpośredniego „otoczenia” poszczególnych produkcji, czyli informacje o bardziej globalnym charakterze.

Definicja

Dla każdej produkcji $p = (X_0 \rightarrow X_1 \dots X_n) \in P$

$$NDDP(p) = DDP(p)^+ \setminus \{(X_i.a \rightarrow X_j.b) : X_i.a, X_j.b \in AF(p)\}$$

Jest znormalizowanym domknięciem przechodnim grafu $DDP(p)$.



Przykład c. d.

$$NDDP(S \rightarrow E) = \{(E.s \rightarrow E.i), \\ (E.t \rightarrow S.r)\} \quad (= DDP(S \rightarrow E))$$

$$NDDP(E \rightarrow E_1 E_2) = \{(E_1.s \rightarrow E.s), \\ (E_2.s \rightarrow E.s), \\ (E.i \rightarrow E_1.i), \\ (E.i \rightarrow E_2.i), \\ (E_1.t \rightarrow E.t), \\ (E_2.t \rightarrow E.t)\} \quad (= DDP(E \rightarrow E_1 E_2))$$

$$NDDP(E \rightarrow \underline{id}) = \{(\underline{id}.s \rightarrow E.s), \\ (E.i \rightarrow E.t)\} \quad (= DDP(E \rightarrow \underline{id}))$$

(W naszym przykładzie obliczenie $NDDP$ nie zmieniło poprzedniej zawartości DDP .)



Obliczanie atrybutów w atrybutowanym drzewie rozbioru syntaktycznego

Definicja

Indukowane zależności atrybutów gramatyki atrybutywnej AG definiuje się następująco:

$$(1) \forall p \in P : IDP(p) := NDDP(p)$$

(2)

$$\forall X \in (N \cup T) : IDS(X) := \{(X.a \rightarrow X.b) : \exists q : (X.a \rightarrow X.b) \in IDP(q)^+\}$$

$$(3) \forall p = (X_0 \rightarrow X_1 \dots X_n) \in P$$

$$IDP(p) := IDP(p) \cup IDS(X_0) \cup \dots \cup IDS(X_n)$$

(4) Powtarzać kroki (2) i (3) dopóty, dopóki zbiory IDS i IDP ulegają zmianie



Przykład c. d.

$$IDP(S \rightarrow E) = NDDP(S \rightarrow E) \cup \{(E.i \rightarrow E.t)\}$$

$$IDP(E \rightarrow E_1 E_2) = NDDP(S \rightarrow E) \cup$$

$$\{(E.s \rightarrow E.i), (E_1.s \rightarrow E_1.i), (E_2.s \rightarrow E_2.i), (E.i \rightarrow E.t), (E_1.i \rightarrow E_1.t), (E_2.i \rightarrow E_2.t)\}$$

$$IDP(E \rightarrow \underline{id}) = NDDP(E \rightarrow \underline{id}) \cup \{(E.s \rightarrow E.i)\}$$

$$IDS(S) = IDS(\underline{id}) = \emptyset$$

$$IDS(E) = \{(E.s \rightarrow E.i), (E.i \rightarrow E.t)\}$$

Intuicyjnie na podstawie $IDS(E)$ można określić relację porządkującą zbiór $A(E)$:

$$\left\{ (E.s \mapsto E.i), (E.i \mapsto E.t), (E.s \mapsto E.t), \begin{cases} E.s \mapsto E.s \\ E.i \mapsto E.i \\ E.t \mapsto E.t \end{cases} \right\}$$

(czyli $E.s \mapsto E.i \mapsto E.t$)

Definicja

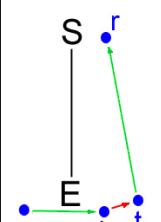
Dla każdej produkcji $p = (X_0 \rightarrow X_1 \dots X_n) \in P$

$DP(p) = IDP(p) \cup \{(X_i.a \rightarrow X_i.b) : X_i.a \mapsto X_i.b\}$
 nazywamy zbiorem (relacją) zależności atrybutów.
 \mapsto - relacja porządkująca zbiór $A(X_i)$

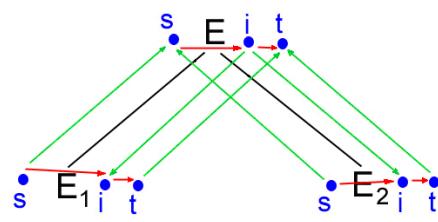
Przykład c. d.

$$\left. \begin{array}{l} DP(S \rightarrow E) = IDP(S \rightarrow E) \\ DP(E \rightarrow EE) = IDP(E \rightarrow EE) \\ DP(E \rightarrow id) = IDP(E \rightarrow id) \end{array} \right\} (*)$$

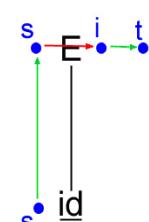
(*) – dla uporządkowania $E.s \mapsto E.i \mapsto E.t$



DP (S → E)



DP (E → EE)



DP (E → id)



Obliczanie atrybutów w atrybutowanym drzewie rozboru syntaktycznego

3. Badamy acykliczność grafów $DP(p)$ dla każdej $p \in P$

Definicja

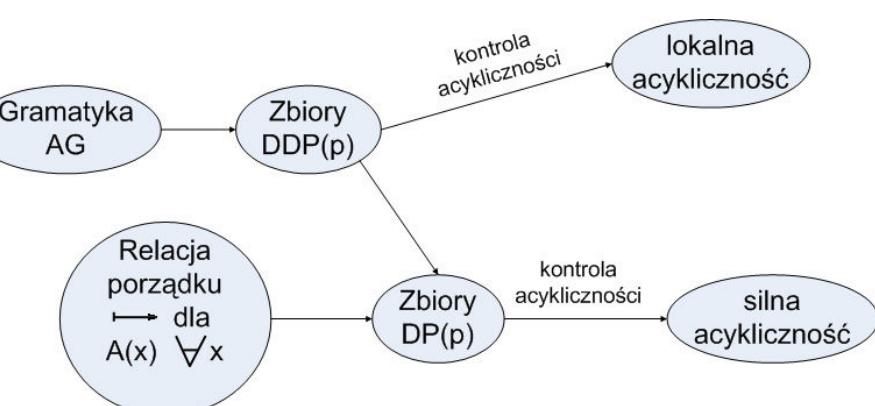
Gramatyka atrybutywna jest silnie acykliczna, gdy graf $DP(p)$ jest acykliczny dla każdej produkcji $p \in P$.

Twierdzenie

Atrybuty w gramatyce atrybutywnej mogą być wyliczane w kolejności wynikającej z relacji porządkującej (\mapsto) zbiory $A(X) (\forall X \in (V \cup \Sigma))$ wtedy i tylko wtedy, gdy gramatyka ta jest silnie acykliczna.



Obliczanie atrybutów w atrybutowanym drzewie rozboru syntaktycznego





Przykład c. d.

Można sprawdzić, że zbiory $DP(p)$ $\forall p \in P$ w naszej gramatyce są acykliczne. Ponieważ zbudowano je dla uporządkowania atrybutów symbolu E takiego, że $E.s \mapsto E.i \mapsto E.t$, więc atrybuty symbolu E powinny być obliczane w tej kolejności.



Obliczanie atrybutów w atrybutowanym drzewie rozboru syntaktycznego

4. Konstruujemy funkcje rekurencyjne do obliczania atrybutów syntetyzowanych.

Funkcja zwraca wartość atrybutu syntetyzowanego symbolu A . Parametrem wejściowym jest wskaźnik wierzchołka odpowiadającego symbolowi A .

Jeśli do obliczenia atrybutu syntetyzowanego potrzebna jest wartość atrybutu dziedziczonego symbolu A , co wynika z przyjętego uporządkowania \mapsto zbioru $A(A)$, to wartość tego atrybutu dziedziczonego jest parametrem wejściowym funkcji.



Przykład c.d.

$$\begin{array}{ll} E \rightarrow E_1 E_2 & \{E.s \leftarrow f_s(E_1.s, E_2.s)\} \\ E \rightarrow \underline{id} & \{E.s \leftarrow \underline{id}.s\} \end{array}$$

```
function Es(n:node):Es_type;
var
    s1,s2:Es_type;
begin
    case produkcja zastosowana w węźle n of
        'E → E1E2' : begin
            s1:=Es(child(n,1));
            s2:=Es(child(n,2));
            Es:=fs(s1,s2);
        end;
        'E → id' : begin
            Es:=get_id_s;
        end;
    else error;
    end;
end;
```



Przykład c.d.

$$\begin{array}{ll} E \rightarrow E_1 E_2 & \{E.i \leftarrow f_{i1}(E.i) \\ & \quad E_2.i \leftarrow f_{i2}(E.i) \\ & \quad E.t \leftarrow f_t(E_1.t, E_2.t) \} \\ E \rightarrow \underline{id} & \{E.t \leftarrow h(E.i)\} \end{array}$$

```
function Et(n:node; i:Ei_type):Et_type;
var
    i1, i2 : Ei_type;
    t1, t2 : Et_type;
begin
    case produkcja zastosowana w węźle n of
        'E → E1E2' : begin
            i1 := f1(i);
            t1 := Et(child(n,1), i1);
            i2 := f2(i);
            t2 := Et(child(n,2), i2);
            Et := ft(t1, t2);
        end;
        'E → id' : begin
            Et:=h(i);
        end;
    else error;
    end;
end;
```



Przykład c.d.

$$S \rightarrow E \quad \{E_1.i \leftarrow g(E.s) \\ S.r \leftarrow E.t\}$$

```
function Sr(n:node):Sr_type;  
var  
    s : Es_type;  
    i : Ei_type;  
    t : Et_type;  
begin  
    s := Es(child(n,1));  
    i := g(s);  
    t := Et(child(n,1), i);  
    Sr := t;  
end;
```



Przykład c.d.

$$S \rightarrow E \quad \{E_1.i \leftarrow g(E.s) \\ S.r \leftarrow E.t\}$$

```
function Sr(n:node):Sr_type;  
var  
    s : Es_type;  
    i : Ei_type;  
    t : Et_type;  
begin  
    s := Es(child(n,1));  
    i := g(s);  
    t := Et(child(n,1));  
    Sr := t;  
end;
```



Przykład c.d.

Wywołanie dla rozpoczęcia obliczania atrybutów:

$value := Sr(root);$

gdzie:

$root$ - korzeń drzewa rozbioru syntaktycznego (wskaźnik)



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Kontrola typów

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki



Analiza semantyczna

Kompilator musi sprawdzić, czy w programie zachowane są zarówno zasady syntaktyczne, jak i semantyczne. To ostatnie sprawdzanie czasem nosi nazwę kontroli statycznej (static checking). Analiza semantyczna obejmuje następujące zagadnienia:

(1) kontrolę typów (type checking) – kompilator musi przykładowo sprawdzić, czy operatory nie są zastosowane do nieodpowiednich operandów,

(1)

```
var
  ss : array [char] of char;
  rr : real;
  ...
  ss:=ss+rr;      → niezgodność typów operandów
                  dla operatora '+'
  ...
```



Analiza semantyczna

- (2) kontrolę przebiegu sterowania (flow-of-control-checking) – kompilator musi sprawdzić, czy sterowanie w programie jest przekazywane w odpowiednie miejsce; (przykład – czy nie jest wykonywany skok spoza pętli do jej wnętrza),

```
(2)    int fun ( i, j, k )
          int i, j, k;
          {
              if(j==k) return i;
              break;      → gdzie przekazać sterowanie?
              if(j<k) return j;
              break;      → gdzie przekazać sterowanie?
              if(j>k) return k;
          }
```



Analiza semantyczna

- (3) kontrolę unikalności (uniqueness checking) – kompilator powinien sprawdzić, czy pewne obiekty są definiowane w programie tylko jeden raz, przykładowo etykiety w komplikowanym module (bloku) muszą być unikalne, kategorie w typach enumeracyjnych muszą być unikalne,

```
(3)    type
          colors = (violet, indigo, magenta,
                     cyan)
          kolory = (yellow, red, gray, violet,
                     orange)

violet → elementy typu wyliczeniowego nie mogą się
                     powtarzać
```

Analiza semantyczna

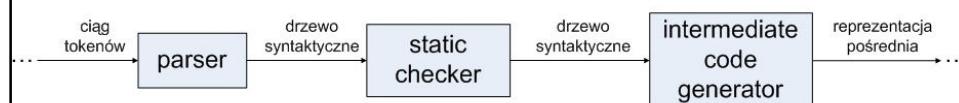
- (4) kontrolę powtarzalności nazw (name-related checking) – kompilator powinien sprawdzić, czy w pewnych konstrukcjach określona nazwa pojawia się więcej niż jeden raz.

```
(4)      dane SEGMENT WORD PUBLIC
          zmienna DW 13
          komunikat DB 'Gdzie jest błąd,
          kochanie?'
          adres DD komunikat
          extra ENDS
```

dane, extra → nazwa musi się powtórzyć

Analiza semantyczna

Kontrola statyczna, a w szczególności także kontrola typów może być przeprowadzana (przynajmniej częściowo) równolegle z parsingiem i/lub generacją kodu pośredniego albo niezależnie, jak na schemacie poniżej. Wykorzystywany jest przy tym mechanizm gramatyk atrybutywnych (definicji kierowanych składnią).





System typów (przykładowy)

Typ konstrukcji językowej określony jest przez „wyrażenie typu”. Wyrażenie typu jest albo typem podstawowym, albo jest tworzone z wyrażenia typu poprzez zastosowanie „konstruktora typu”.

Wyrażenie typu może być zdefiniowane w następujący sposób:

- (1) Typ podstawowy jest wyrażeniem typu. Poza typami podstawowymi jak *boolean*, *char*, *integer*, *real* wprowadza się specjalne typy podstawowe: *void* dla konstrukcji nie wymagających sprecyzowania typu oraz *type_error* dla konstrukcji błędnych z punktu widzenia kontrolera typów.
- (2) Jeżeli typy mogą mieć własne nazwy – nazwa typu jest (a właściwie może być) wyrażeniem typu.



System typów (przykładowy)

- (3) Konstruktor typu zastosowany do wyrażenia typu jest wyrażeniem typu.
 - (a) Jeśli T jest wyrażeniem typu, to $\text{array}(I, T)$ jest wyrażeniem typu określającym typ tablicy o elementach typu T i typie indeksowym I
- Przykład: deklaracja
- ```
var A:array [char] of integer;
```
- kojarzy wyrażenie typu  $\text{array}(\text{char}, \text{integer})$  ze zmienną „ $A$ ”.
- (b) Jeżeli  $T_1$  i  $T_2$  są wyrażeniami typu, to ich „produkt kartezjański”  $T_1 \times T_2$  jest wyrażeniem typu (operacja „ $\times$ ” jest lewostronnie łączna).



## System typów (przykładowy)

- (c) Konstruktor *record*(...) zastosowany do produktu wyrażeń typu pól rekordu jest wyrażeniem typu. Wyrażenie typu dla pola rekordu ma postać:  
(nazwa\_pola  $\times$  typ\_pola)

Przykład:

```
type row = record
 address : integer;
 lexeme : array[byte] of char
 end;
var table : array[1..100] of row;
```

Ze zmienną "table" skojarzone jest wyrażenie typu:  
*array*(1..100, *record*((*address*  $\times$  *integer*)  $\times$  (*lexeme*  $\times$  *array*(*byte*, *char*))))



## System typów (przykładowy)

- (d) Jeśli *T* jest wyrażeniem typu, to *pointer(T)* jest wyrażeniem typu określającym wskaźnik do obiektu typu *T*.

Przykład: deklaracja

```
var p : ↑integer;
```

powoduje skojarzenie wyrażenia typu: *pointer(integer)* ze zmienną „*p*”.



## System typów (przykładowy)

- (e) Jeśli  $D$  jest wyrażeniem typu odpowiadającym liście argumentów funkcji, a  $R$  wyrażeniem typu określającym typ wyniku funkcji, to zapis  $D \rightarrow R$  jest wyrażeniem typu skojarzonym z tą funkcją (operacja „ $\rightarrow$ ” jest prawostronnie łączna i ma niższy priorytet od operacji „ $\times$ ”)

Przykład: deklaracja

`function f(a, b : char) : ^integer;`  
kojarzy wyrażenie typu:  $char \times char \rightarrow pointer(integer)$  z funkcją  $f$ .

Niektóre języki, np. LISP dopuszczają konstrukcje o typie:  
 $(integer \rightarrow integer) \rightarrow (integer \rightarrow integer)$



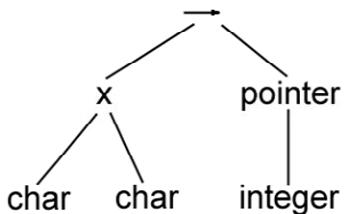
## System typów (przykładowy)

- (4) Wyrażenie typu może zawierać zmienne, wartościami których są wyrażenia typu.

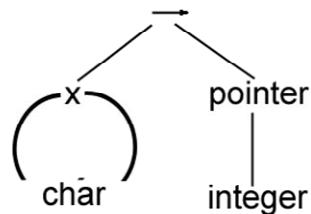
## System typów (przykładowy)

Wyrażenia typu mogą być reprezentowane np. w postaci grafów lub drzew.

Przykład: Wyrażenie typu:  $char \times char \rightarrow pointer(integer)$



(drzewo)



(dag)

## System typów

Systemem typów nazywamy zbiór reguł przypisujących wyrażenia typu do różnych konstrukcji programu kompilowanego. Kompilator powinien więc implementować system typów, aby mieć możliwość kontroli poprawności użycia typów przez programistę i poprawnie dokonać tłumaczenia.

Kontrolę typów wykonywaną przez kompilator nazywamy statyczną, natomiast sprawdzanie podczas działania programu wynikowego – sprawdzaniem dynamicznym. Właściwie każde sprawdzanie może być zrobione dynamicznie, jeżeli kod wynikowy zawiera typ elementu wraz z jego wartością.



## Statyczna i dynamiczna kontrola typów

Z reguły nie wszystkie kontrole związane z typami mogą być przeprowadzone statycznie (w trakcie kompilacji).

### Przykład

```
var table : array[0..255] of char;
 i : integer;
```

Poprawna w Pascalu konstrukcja:

```
table[i]
```

nie pozwala kompilatorowi zagwarantować, że w trakcie wykonywania programu wartość zmiennej indeksowej „i” będzie mieścić się w przedziale 0..255.



## Statyczna i dynamiczna kontrola typów

Język programowania nazywamy „ścisłe typowanym” (strongly typed) jeżeli jego kompilator może zagwarantować, że zaakceptowany przez niego program wykona się bez „ błędów typów”.



## Specyfikacja przykładowego kontrolera typów

|                                           |                                                                                         |
|-------------------------------------------|-----------------------------------------------------------------------------------------|
| $P \rightarrow D ; S$                     |                                                                                         |
| $D \rightarrow D ; D$                     |                                                                                         |
| $D \rightarrow id : T$                    | $\{addtype(id.entry, T.type)\}$                                                         |
| $T \rightarrow char$                      | $\{T.type \leftarrow char\}$                                                            |
| $T \rightarrow integer$                   | $\{T.type \leftarrow integer\}$                                                         |
| $T \rightarrow boolean$                   | $\{T.type \leftarrow boolean\}$                                                         |
| $T \rightarrow \uparrow T_1$              | $\{T.type \leftarrow pointer(T_1.type)\}$                                               |
| $T \rightarrow array [ num ] of T_1$      | $\{T.type \leftarrow array(1..num.val, T_1.type)\}$                                     |
| $T \rightarrow function id ( T_1 ) : T_2$ | $\{T.type \leftarrow (T_1.type \rightarrow T_2.type);$<br>$addtype(id.entry, T.type)\}$ |



## Specyfikacja przykładowego kontrolera typów

|                              |                                                                                                                     |
|------------------------------|---------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow literal$      | $\{E.type \leftarrow char\}$                                                                                        |
| $E \rightarrow num$          | $\{E.type \leftarrow integer\}$                                                                                     |
| $E \rightarrow true$         | $\{E.type \leftarrow boolean\}$                                                                                     |
| $E \rightarrow false$        | $\{E.type \leftarrow boolean\}$                                                                                     |
| $E \rightarrow id$           | $\{E.type \leftarrow lookup(id.entry)\}$                                                                            |
| $E \rightarrow E_1 + E_2$    | $\{E.type \leftarrow if E_1.type = integer \text{ and } E_2.type = integer \text{ then integer else type\_error}\}$ |
| $E \rightarrow E_1 mod E_2$  | $\{E.type \leftarrow if E_1.type = integer \text{ and } E_2.type = integer \text{ then integer else type\_error}\}$ |
| $E \rightarrow E_1 [ E_2 ]$  | $\{E.type \leftarrow if E_2.type = integer \text{ and } E_1.type = array(s,t) \text{ then t else type\_error}\}$    |
| $E \rightarrow E_1 \uparrow$ | $\{E.type \leftarrow if E_1.type = pointer(t) \text{ then t else type\_error}\}$                                    |
| $E \rightarrow E_1 ( E_2 )$  | $\{E.type \leftarrow if E_2.type = s \text{ and } E_1.type = (s \rightarrow t) \text{ then t else type\_error}\}$   |



## Specyfikacja przykładowego kontrolera typów

|                                                |                                                                                                                       |
|------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| $S \rightarrow id := E$                        | $\{S.type \leftarrow \text{if } \text{lookup}(id.entry) = E.type \text{ then}$<br>$\text{void else type\_error}\}$    |
| $S \rightarrow \text{if } E \text{ then } S_1$ | $\{S.type \leftarrow \text{if } E.type = \text{boolean} \text{ then } S_1.type$<br>$\text{else type\_error}\}$        |
| $S \rightarrow S_1 ; S_2$                      | $\{S.type \leftarrow \text{if } S_1.type = \text{void and } S_2.type =$<br>$\text{void then void else type\_error}\}$ |



## Równoważność wyrażeń typu

Reguły semantyczne w poprzednim przykładzie miały postać:  
„if dwa wyrażenia typu są równe (równoważne)  
    then zwróć określony typ  
    else zwróć ‘type\_error’”

Pytanie: „kiedy dwa wyrażenia typu są równoważne?”



## Równoważność strukturalna

### (1) Równoważność strukturalna (structural equivalence)

W przypadku, gdy wyrażenia typu są zbudowane wyłącznie z typów podstawowych i konstruktorów (tzn. nie zawierają nazw typów) naturalnym sposobem określenia równoważności typów jest równoważność strukturalna. Dwa wyrażenia typu są równoważne strukturalnie, gdy są albo tymi samymi typami podstawowymi, albo są utworzone przez zastosowanie tego samego konstruktora do równoważnych strukturalnie wyrażeń typu (jest to definicja rekurencyjna). Innymi słowy dwa wyrażenia typu są strukturalnie równoważne, wtedy i tylko wtedy, gdy są one identyczne.



## Równoważność strukturalna

Przykład:

„*pointer(integer)*” jest równoważny tylko wyrażeniu  
„*pointer(integer)*”  $\Rightarrow$  zastosowano ten sam konstruktor *pointer(...)*  
do równoważnych wyrażeń typu (tutaj do identycznych typów  
podstawowych - *integer*)



## Równoważność strukturalna

Przykład:

```
type
 cell = array[char] of integer;
 link = \uparrow cell;
var
 next : link;
 last : link;
 p : \uparrow cell;
 q, r : \uparrow cell;
```

Strukturalna równoważność typów nie dopuszcza nazw typów. W związku z tym:

*link* oznacza *pointer(array(char, integer))*

$\uparrow$  *cell* oznacza *pointer(array(char, integer))*

czyli w sensie równoważności strukturalnej zmienne: *next*, *last*, *p*, *q*, *r* mają ten sam typ.



## Równoważność strukturalna

Procedura (funkcja rekurencyjna) do testowania równoważności strukturalnej:

```
function SEQUIV(s, t) : boolean;
begin
 if s oraz t są identycznymi typami podstawowymi then
 SEQUIV := true
 else if s=array(s1,s2) and t=array(t1,t2) then
 SEQUIV := SEQUIV(s1,t1) and SEQUIV(s2,t2)
 else if s= s1 × s2 and t= t1 × t2 then
 SEQUIV := SEQUIV(s1,t1) and SEQUIV(s2,t2)
 else if s=pointer(s1) and t=pointer(t1) then
 SEQUIV := SEQUIV(s1,t1)
 else if s= s1 → s2 and t= t1 → t2 then
 SEQUIV := SEQUIV(s1,t1) and SEQUIV(s2,t2)
 else SEQUIV := false
end;
```

Uwaga: dla uproszczenia pominięto konstruktor *record(...)*.



## Równoważność strukturalna

(...)

else if s=array(  $s_1, s_2$  ) and t=array(  $t_1, t_2$  ) then

SEQUIV := SEQUIV(  $s_1, t_1$  ) and SEQUIV(  $s_2, t_2$  ) //(\*)

(...)

(\*) – Uwaga: W praktyce niekiedy równoważność strukturalna bywa rozumiana mniej rygorystycznie. Jeśli np. tablice są przekazywane jako parametry procedur lub funkcji, może nas nie interesować zakres zmienności indeksu tablicy, a tylko typ jej elementu. Wówczas odpowiedni zapis w procedurze SEQUIV może mieć postać:

(...)

else if s = array(  $s_1, s_2$  ) and t = array(  $t_1, t_2$  ) then

SEQUIV := SEQUIV(  $s_2, t_2$  )

(...)



## Równoważność strukturalna

### Przykład:

W kompilatorze języka C napisanym przez Ritchie'ego stosowano m.in. następujące konstruktory:

*pointer(t)* – wskaźnik na obiekt typu *t*

*freturns(t)* – funkcja pewnych argumentów zwracająca obiekt typu *t*

*array(t)* – tablica (długość nieistotna) elementów typu *t*

Wówczas:

*array(pointer(freturns(char)))* – oznacza tablicę wskaźników na funkcje zwracające obiekty typu znakowego.



## Równoważność strukturalna

Przy zastosowaniu kodowania:

| konstruktor | kod |
|-------------|-----|
| pointer     | 01  |
| array       | 10  |
| freturns    | 11  |
| ...         | ... |

| typ podstawowy | Kod   |
|----------------|-------|
| boolean        | 0000  |
| char           | 0001  |
| integer        | 0010  |
| real           | 0011  |
| .....          | ..... |



## Równoważność strukturalna

...można wyrażenia typu oznaczać ciągiem bitów:

| wyrażenie typu                 | kod        |
|--------------------------------|------------|
| char                           | 0000000001 |
| freturns(char)                 | 0000110001 |
| pointer(freturns(char))        | 0001110001 |
| array(pointer(freturns(char))) | 1001110001 |

Wówczas dwie różne sekwencje bitów nie mogą reprezentować tego samego typu. Oczywiście dwie jednakowe sekwencje bitów mogą reprezentować różne typy, gdyż wielkości tablic ani argumenty funkcji w takim zapisie nie są reprezentowane.



## Równoważność przy dopuszczeniu nazw typów

### (2) Równoważność przy dopuszczeniu nazw typów (name equivalence)

Jeśli w pewnych językach dopuszcza się nadawanie typom nazw, to można zgodzić się aby nazwy typów pojawiały się w wyrażeniach typu. Traktuje się każdą nazwę typu jako odrębny typ. Przy takich założeniach dwa wyrażenia typu są równoważne wtedy i tylko wtedy, gdy są one identyczne.

Działanie takiego systemu typów zależy od implementacji...



## Równoważność przy dopuszczeniu nazw typów

Przykład:

```
type link = ↑cell;
var next : link;
 last : link;
 p : ↑cell;
 q, r : ↑cell;
```

| Zmienna | Wyrażenie typu |
|---------|----------------|
| next    | link           |
| last    | link           |
| p       | pointer(cell)  |
| q       | pointer(cell)  |
| r       | pointer(cell)  |

Zgodnie z tak rozumianą równoważnością (przy dopuszczeniu nazw):

- równoważne są typy zmiennych: „next” i „last”
- równoważne są typy zmiennych: „p”, „q”, „r”
- nie sa równoważne typy zmiennych np.: „next” i „p”

## Równoważność przy dopuszczeniu nazw typów

Przykład: Wiele implementacji Pascal'a zakłada niejawną nazwę typu dla deklaracji zmiennych, w których typ nie został nazwany.

Zapis:

```
type
 link = ^cell;
var
 next : link;
 last : link;
 p : ^cell;
 q, r : ^cell;
```

jest interpretowany jako:

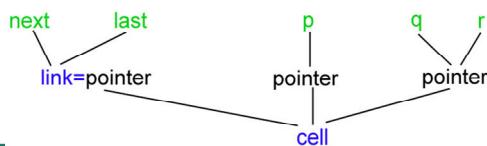
```
type
 link = ^cell;
 ntyp_p = ^cell;
 ntyp_q_r = ^cell;
var
 next : link;
 last : link;
 p : ntyp_p;
 q : ntyp_q_r;
 r : ntyp_q_r;
```

## Równoważność przy dopuszczeniu nazw typów

```
type
 link = ^cell;
 ntyp_p = ^cell;
 ntyp_q_r = ^cell;
var
 next : link;
 last : link;
 p : ntyp_p;
 q : ntyp_q_r;
 r : ntyp_q_r;
```

Wobec tego:

- (a) równoważne typy mają zmienne:
  - (-) "next" oraz "last"
  - (-) "q" oraz "r"
- (b) każda ze zmiennych "next", "p" oraz "q" ma inny typ.





## Cykle w reprezentacji typów



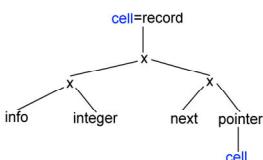
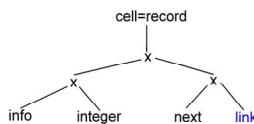
## Cykle w reprezentacji typów

Cykle w wyrażeniach typu mogą pojawiać się w zasadzie tylko i wyłącznie w konstrukcjach: *pointer(record(...))*.

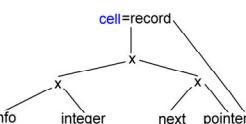
# Cykle w reprezentacji typów

Przykład:

```
type
 link = ^ cell;
 cell = record
 info : integer;
 next : link;
 end;
```



Reprezentacja acykliczna



Reprezentacja cykliczna

# Cykle w reprezentacji typów

Przykład:

Analogiczna definicja w języku C ma postać:

```
struct cell {
 int info;
 struct cell *next;
};
```

W języku C nazwa rekordu (nazywanego tutaj strukturą) stanowi część nazwy typu. Poza tym język C wymaga aby nazwa była zadeklarowana przed jej użyciem.

W związku z tym język C wykorzystuje reprezentację acykliczną konstrukcji rodzaju *pointer(record(...))*. Wszystkie potencjalne cykle sprowadzają się do konstrukcji *pointer(record(...))*. Ponieważ nazwa rekordu jest częścią jego typu, sprawdzanie równoważności jest kończone gdy po konstruktorze *pointer(...)* spotykany jest konstruktor *record(nazwa rekordu, ...)*. Wówczas typy są równoważne, gdy rekordy porównywane mają takie same nazwy, w przeciwnym przypadku sygnalizowana jest nierównoważność typów.



## Niejawne konwersje typów

Przykład:

```
var x : real;
 i : integer;
.....
x := x+i;
.....
```

$x+i$  – tłumaczenie to do np. odwrotnej notacji polskiej powinno być następujące:

$x \quad i \quad \underline{\text{inttoreal}} \quad \underline{\text{real+}}$



## Niejawne konwersje typów

Przykład: ustalanie typu wyrażenia

|                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow \underline{\text{num}}$                          | $\{E.\text{type} \leftarrow \text{integer}\}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| $E \rightarrow \underline{\text{num}} . \underline{\text{num}}$ | $\{E.\text{type} \leftarrow \text{real}\}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| $E \rightarrow \underline{id}$                                  | $\{E.\text{type} \leftarrow \text{lookup}(\underline{id}.\text{entry})\}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| $E \rightarrow E_1 \underline{\text{op}} E_2$                   | $\{E.\text{type} \leftarrow \underline{\text{if }} E_1.\text{type} = \text{integer} \underline{\text{and}} \underline{\text{if }} E_2.\text{type} = \text{integer} \underline{\text{then}} \text{integer}$<br>$\underline{\text{else if }} E_1.\text{type} = \text{integer} \underline{\text{and}} E_2.\text{type} = \text{real}$<br>$\underline{\text{then}} \text{real}$<br>$\underline{\text{else if }} E_1.\text{type} = \text{real} \underline{\text{and}} E_2.\text{type} = \text{integer}$<br>$\underline{\text{then}} \text{real}$<br>$\underline{\text{else if }} E_1.\text{type} = \text{real} \underline{\text{and}} E_2.\text{type} = \text{real}$<br>$\underline{\text{then}} \text{real}$<br>$\underline{\text{else type_error}}$ |

O ile jest to możliwe konwersje (niewiązane) stałych powinny być dokonywane na etapie kompilacji, a nie wykonania.



## Przeciążanie funkcji lub operatorów

Operatory lub funkcje mogą mieć „różne znaczenia” w zależności od kontekstu. Nazywamy je wtedy przeciążonymi.

Przykład: (Turbo Pascal)

var

```
x, y : real;
i, j : integer;
s, t : string;
```

$x + y \rightarrow$  dodawanie zmennopozycyjne

$i + j \rightarrow$  dodawanie stałopozycyjne

$s + t \rightarrow$  konkatenacjałańcuchów znaków

Operator ‘+’ nazywamy operatorem przeciążonym.



## Przeciążanie funkcji lub operatorów

Przy założeniu, że podwyrażenia mają unikalny typ, kontrolę typu funkcji przeprowadzano w następujący sposób:

|                          |                                                                                                                                      |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1(E_2)$ | $\{E.type \leftarrow \text{if } E_2.type = s \text{ and } E_1.type = (s \rightarrow t) \\ \text{then } t \text{ else type\_error}\}$ |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|

Jeżeli dopuścimy aby funkcje mogły być przeciążone, wówczas określenie typu podwyrażenia będącego wywołaniem funkcji będzie np. następujące:

|                          |                                                                                            |
|--------------------------|--------------------------------------------------------------------------------------------|
| $E \rightarrow E_1(E_2)$ | $\{E.types \leftarrow \{t : \exists s \in E_2.types : (s \rightarrow t) \in E_1.types\}\}$ |
|--------------------------|--------------------------------------------------------------------------------------------|

W wielu językach dopuszczających przeciążalność wymaga się, aby wyrażenie miało unikalny (pojedynczy) typ, podczas gdy podwyrażenia składowe tego warunku spełniać nie muszą.



## Przeciążanie funkcji lub operatorów

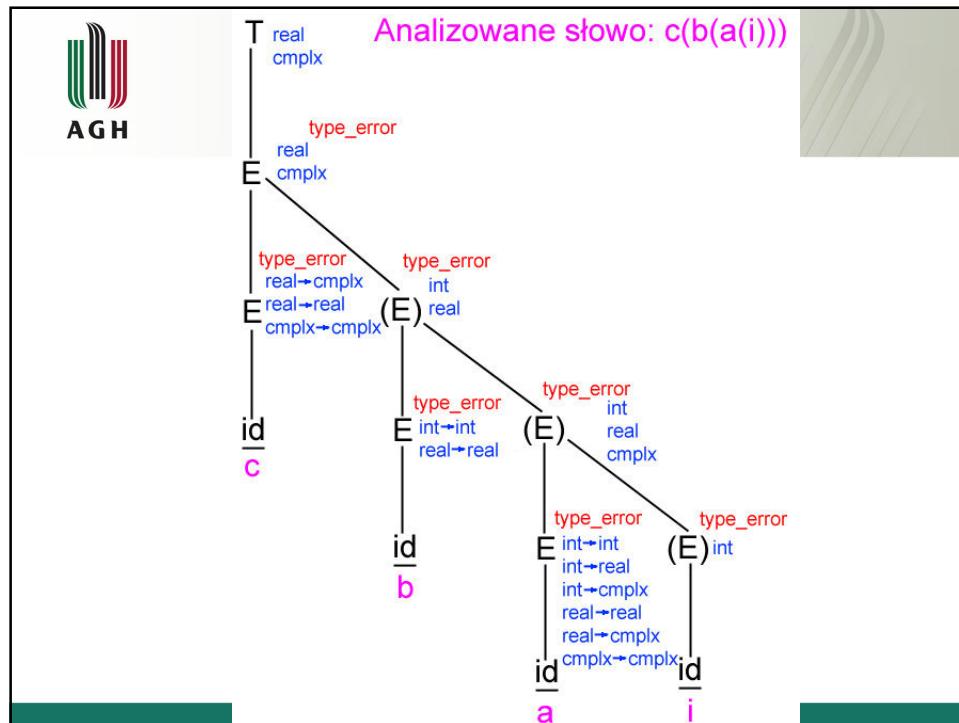
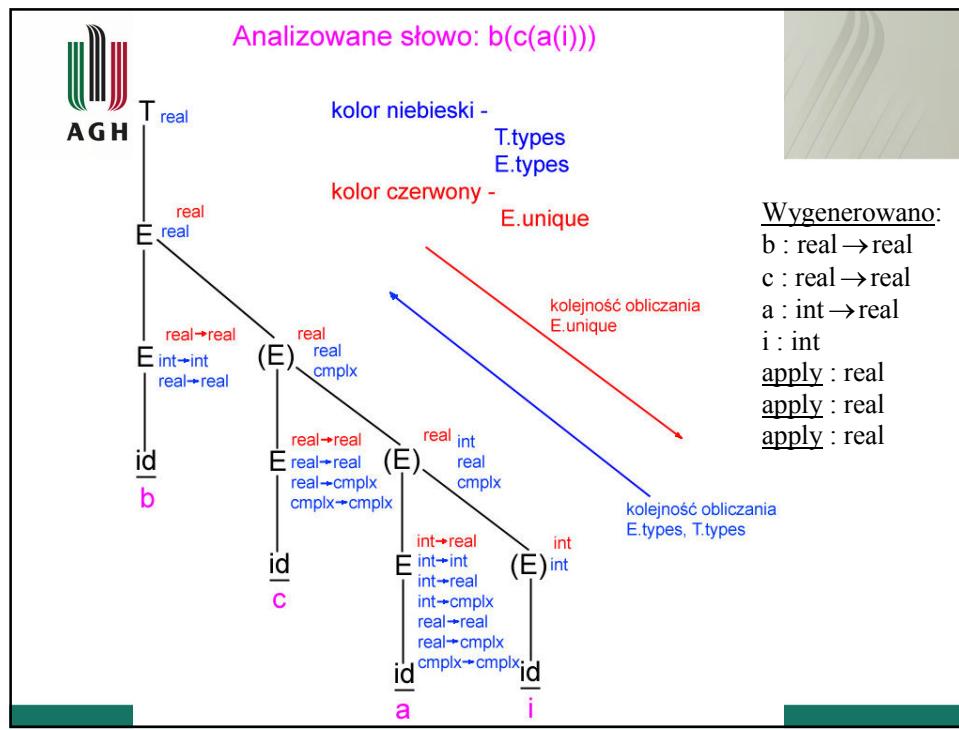
Przykład kontroli unikalności typu wyrażenia i generacji ONP

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $T \rightarrow E$        | $T.types \leftarrow E.types$<br>$E.unique \leftarrow \underline{if} \ T.types = \{t\} \underline{then} \ t \ \underline{else} \ type\_error$<br>$T.code \leftarrow E.code$                                                                                                                                                                                                                                                                                                                                         |
| $E \rightarrow id$       | $E.types \leftarrow lookup(id.entry)$<br>$E.code \leftarrow gen(id.lexeme ': E.unique)$                                                                                                                                                                                                                                                                                                                                                                                                                            |
| $E \rightarrow E_1(E_2)$ | $E.types \leftarrow \{s' : \exists s \in E_2.types : (s \rightarrow s') \in E_1.types\}$<br>$t := E.unique$<br>$S := \{s : s \in E_2.types \ and \ (s \rightarrow t) \in E_1.types\}$<br>$E_2.unique \leftarrow \underline{if} \ S = \{s\} \underline{then} \ s \ \underline{else} \ type\_error$<br>$E_1.unique \leftarrow \underline{if} \ S = \{s\} \underline{then} \ (s \rightarrow t) \ \underline{else} \ type\_error$<br>$E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('apply ': E.unique)$ |



## Przeciążanie funkcji lub operatorów

| Identyfikator | Typ                                                                                                                                                               |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $i$           | $int$                                                                                                                                                             |
| $a$           | $int \rightarrow int \quad real \rightarrow real \quad cmplx \rightarrow cmplx$<br>$int \rightarrow real \quad real \rightarrow cmplx$<br>$int \rightarrow cmplx$ |
| $b$           | $int \rightarrow int \quad real \rightarrow real$                                                                                                                 |
| $c$           | $real \rightarrow cmplx \quad cmplx \rightarrow cmplx$<br>$real \rightarrow real$                                                                                 |





**AGH**

AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

# **Przydział pamięci**

## **Teoria kompilacji**

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

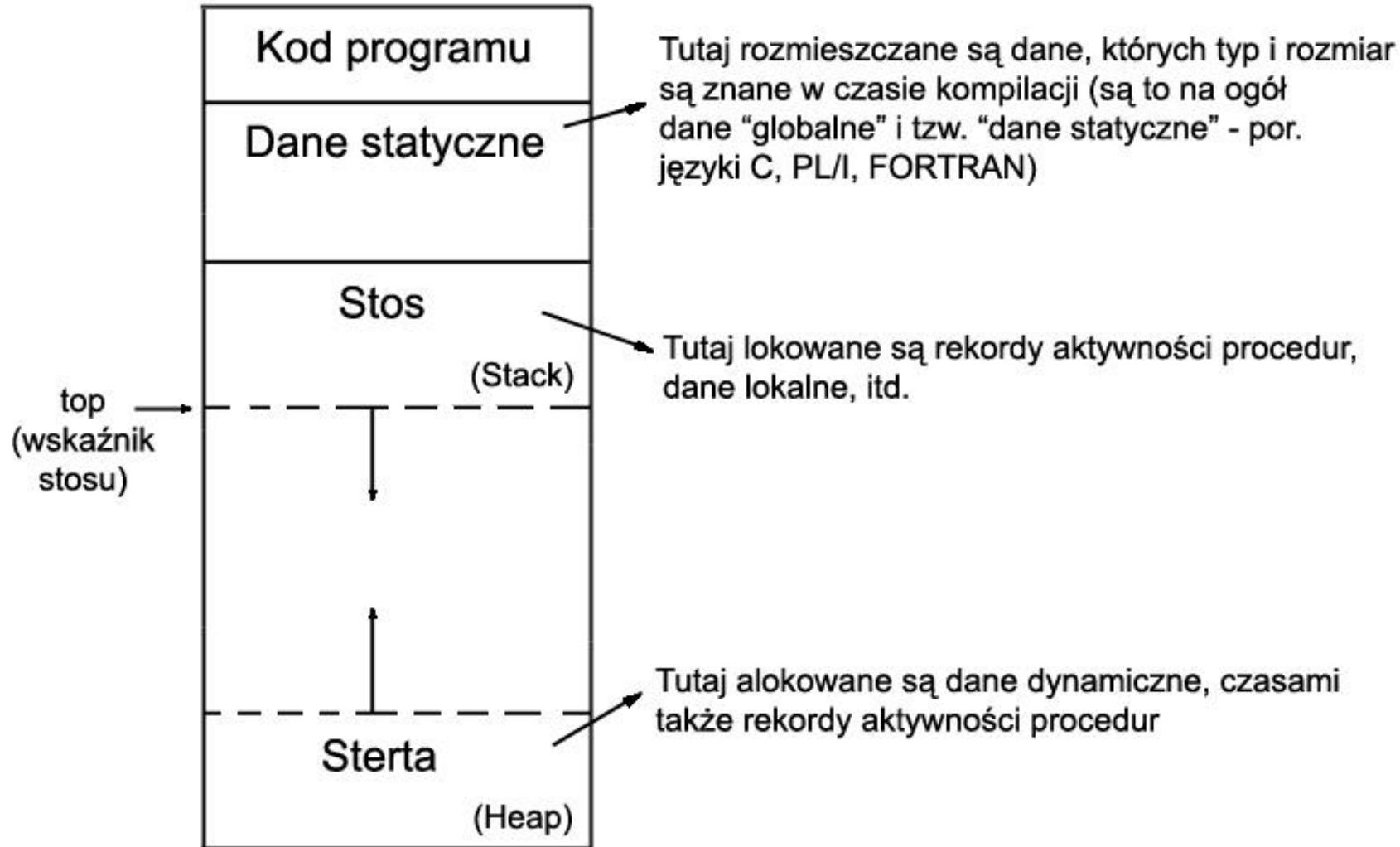
# Terminologia

```
Program s(input,output) {SORT} program główny można traktować
 także jako procedurę
var a: array[0..10] of integer; deklaracja zmiennych globalnych
procedure r; {READ ARRAY}
 var i: integer;
 Begin for i := 1 to 9 do read(a[i]); end;
Function p(y,z :integer):integer; {PARTITION} y,z – parametry formalne
Var i,j,x,v : integer; deklaracje zmiennych lokalnych
Begin ... end;
Procedure q(m,n:integer) ; {QUICKSORT}
 Var i integer;
 Begin
 If (n>m) then
 Begin
 i := p(m,n); m,n – parametry aktualne
 q(m,i - 1) wywołanie rekursywne
 q(i + 1,n);
 end;
 end;
Begin
 a[0] := -9999; a[10] := 9999;
 r;
 q(1,9);
end.
```

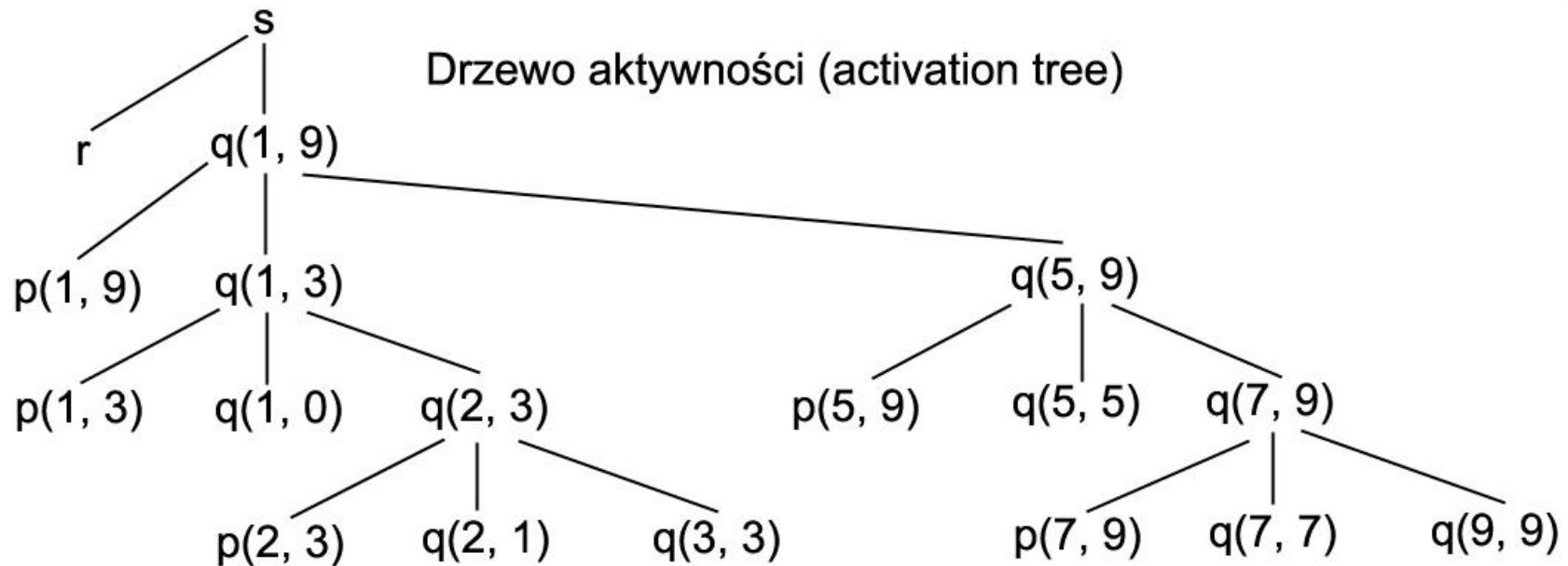
# Pytania projektanta kompilatora

1. Czy procedury mogą być rekursywne?
2. Co się dzieje z wartościami zmiennych lokalnych w momencie, gdy sterowanie opuszcza aktywną w danym momencie procedurę?
3. Czy procedura może odwoływać się do nazw nielokalnych?
4. Jak są przekazywane parametry do procedury?
5. Czy procedury mogą być parametrami wywołań?
6. Czy procedury mogą zwracać wyniki (czy mogą być funkcjami)?
7. Czy wynikiem działania procedury może być procedura?
8. Czy pamięć może być dynamicznie przydzielana pod kontrolą programu?
9. Czy pamięć musi być jawnie zwalniania?

# Typowy podział pamięci dla programu użytkowego



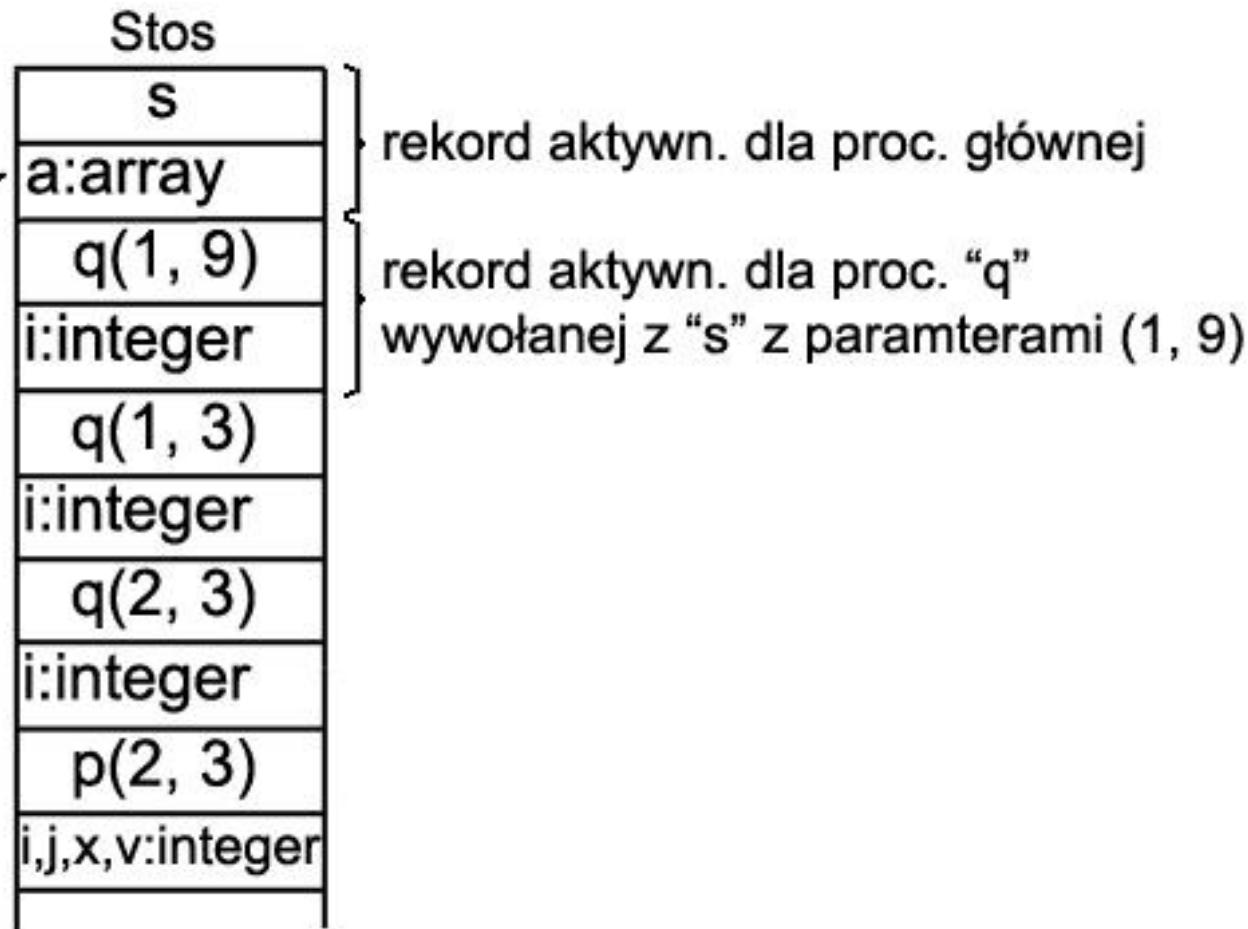
# Drzewo aktywności

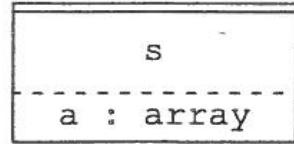
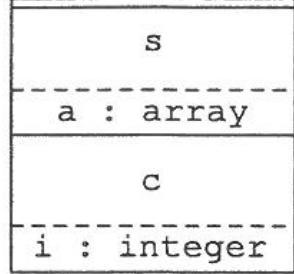
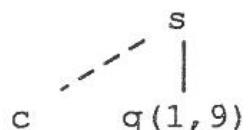
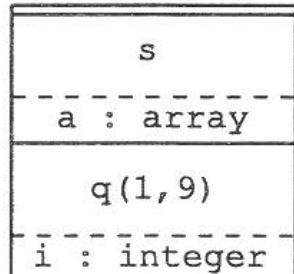
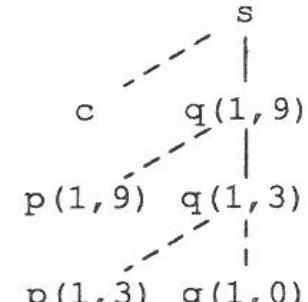
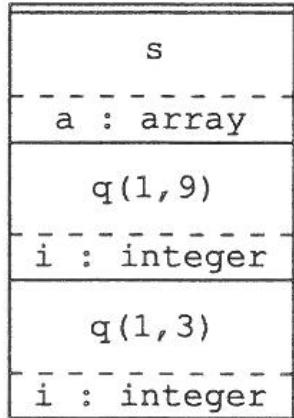


1. Każdy wierzchołek reprezentuje czynność procedury
2. Korzeń reprezentuje czynność głównego programu
3. Wierzchołek  $a$  jest przodkiem wierzchołka  $b \Leftrightarrow$  sterowanie przechodzi z czennej procedury  $a$  do  $b$
4. Wierzchołek  $a$  jest położony z lewej strony wierzchołka  $b$  oraz  $a$  i  $b$  są rodzeństwem  $\Leftrightarrow$  czas aktywności  $a$  jest wcześniejszy od czasu aktywności  $b$

# Stos programu

ta dana może być  
lokowana w pamięci  
stat. zamiast na stosie



| PÓZYCJE<br>W DRZEWIE AKTYWACJI                                                          | REKORDY AKTYWACJI<br>NA STOSIE                                                      | UWAGI                                                                    |
|-----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| s                                                                                       |   | ramka dla s                                                              |
| c<br>  |   | c jest aktywowana                                                        |
| c<br>  |   | ramka dla c<br>została usunięta<br>i q(1,9) została<br>wstawiona na stos |
| c<br> |  | sterowanie właśnie<br>powróciło do q(1,3)                                |

# Rekord aktywności (rama) procedury

|                              |                                               |                                                                                                                                                   |
|------------------------------|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| sztywna<br>długość<br>format | wynik zwracany<br>przez procedurę             | często do tego celu wykorzystywany jest rejestr,<br>a nie pamięć                                                                                  |
|                              | parametry aktualne                            | także czasami są przekazywane w rejestrach                                                                                                        |
|                              | wskaźnik<br>“control link”<br>(opcjonalny)    | wskazuje rekord aktywności procedury wywołującej                                                                                                  |
|                              | wskaźnik<br>“access link”<br>(opcjonalny)     | umożliwia dostęp do danych nielokalnych w językach<br>dopuszczających zagłębianie procedur (Pascal PL I)                                          |
|                              | obszar dla<br>zapamiętania<br>stanu procesora | adres powrotu, status procesora przed wywołaniem<br>procedury i inne informacje konieczne do<br>odtworzenia stanu maszyny po powrocie z procedury |
|                              | obszar na<br>zmienne lokalne                  | zmienne lokalne procedury                                                                                                                         |
|                              | obszar na<br>zmienne robocze                  | zmienne robocze, konieczne np. do wyliczania<br>wartości wyrażeń                                                                                  |

# Rekord aktywności (rama) procedury

W skład rekordu aktywacji (ramki procedury) mogą wchodzić (obligatoryjnie lub opcjonalnie – zależnie od języka programowania oraz jego konkretnej implementacji):

- wartość zwracana,
- parametry aktualne wywołania,
- wiązanie sterowania (*control link*),
- wiązanie dostępu (*access link*),
- zapamiętany stan procesora,
- zmienne lokalne,
- wskaźniki do lokalnych tablic o zmiennej długości,
- lokalne tablice o stałej długości,
- zmienne tymczasowe (robocze).

# Rekord aktywności (rama) procedury

Pamięć dla rekordów aktywacji może być rezerwowana:

- w pamięci statycznej,
- w pamięci stosowej,
- w pamięci sterty,
- częściowo w rejestrach procesora.

# Rekord aktywności (rama) procedury

Przy stosowej rezerwacji pamięci zmienne lokalne dla każdego wywołania są przypisywane do nowej pamięci w każdej aktywacji, ponieważ na stosie jest umieszczany nowy rekord aktywacji; wartości zmiennych lokalnych są usuwane, gdy aktywacja się kończy, ponieważ pamięć dla nich jest usuwana razem z rekordem aktywacji.

Jeżeli ta sama procedura jest n-krotnie wywoływana rekurencyjnie to, przy stosowej rezerwacji pamięci, rekord aktywacji tej procedury jest umieszczany n-krotnie na stosie, po jednym wystąpieniu dla każdej aktywacji.

# Rekord aktywności (rama) procedury

Rezerwacja stertowa rekordów aktywacji jest stosowana, jeżeli wartości zmiennych lokalnych muszą być zachowane, gdy aktywacja się kończy lub gdy aktywacja wywoływana istnieje dłużej niż aktywacja wywołującą.

W rezerwacji statycznej rekordów aktywacji nazwy są przypisywane do pamięci w trakcie komplikacji; ponieważ te przypisania nie zmieniają się w trakcie działania programu, za każdym razem, gdy procedura jest aktywowana, jej nazwy przypisane są do tych samych adresów pamięci, więc wartości nazw lokalnych są zachowywane między kolejnymi aktywacjami tej procedury.

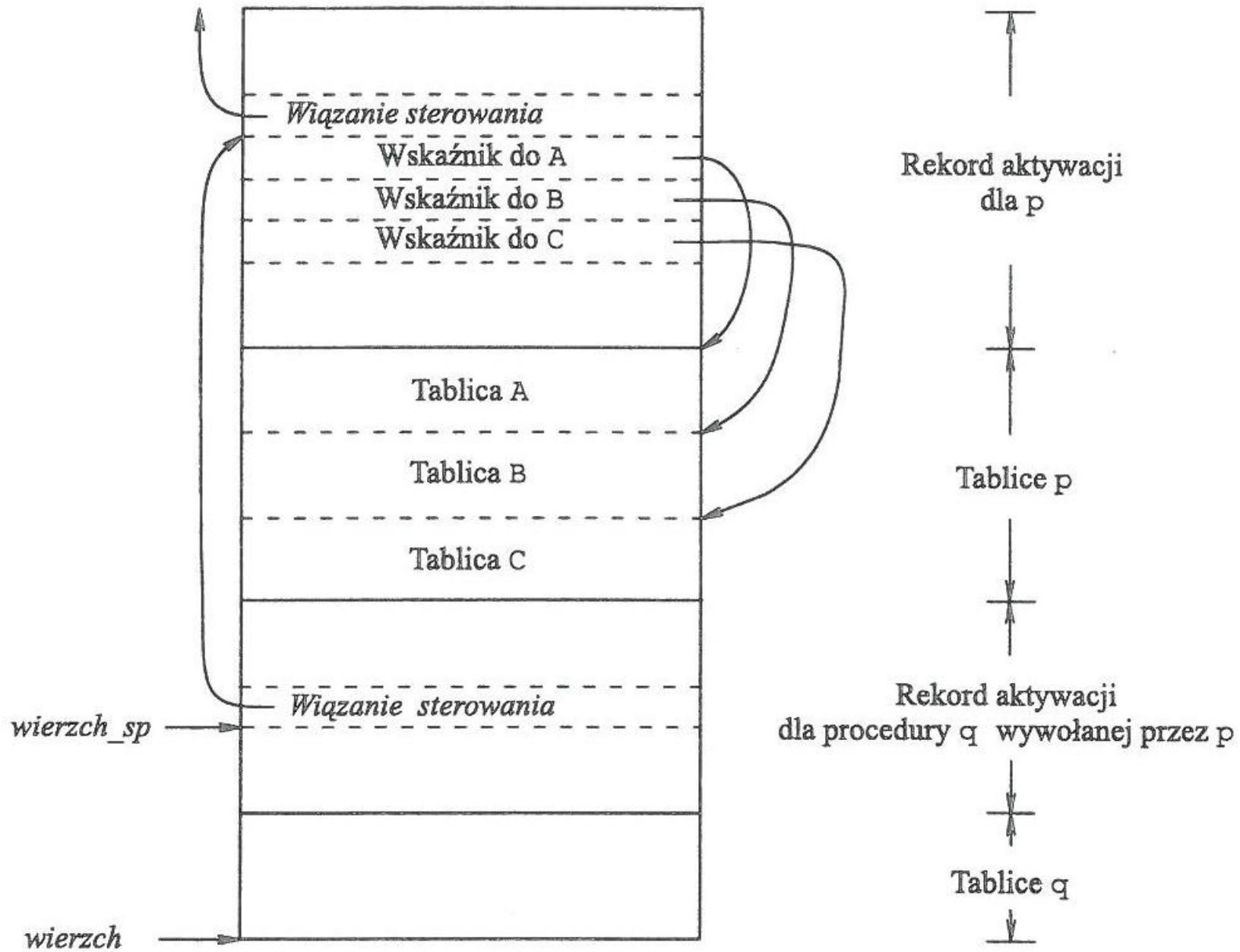
# Rekord aktywności (rama) procedury

Wiązanie sterowania (*control link*) jest wskaźnikiem umieszczonym w rekordzie aktywacji procedury wywoływanej wskazującym na ustalone miejsce w rekordzie aktywacji procedury wywołującej.

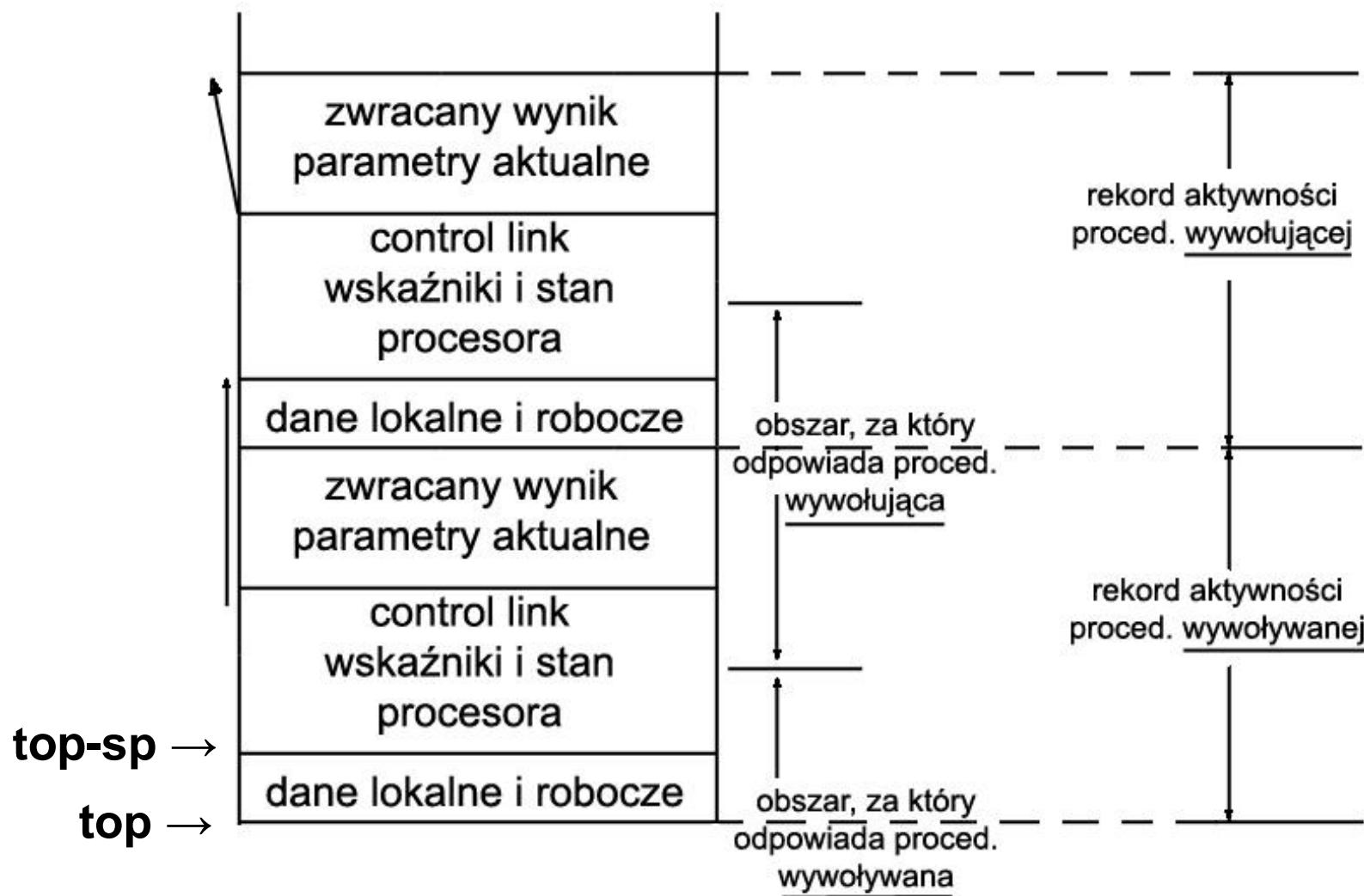
Bezpośredni dostęp do umieszczonych w rekordzie aktywacji parametrów aktualnych, zmiennych lokalnych i zmiennych tymczasowych wymaga znajomości przesunięcia (*offset*) adresu pamięci tych obiektów w stosunku do ustalonego miejsca w rekordzie aktywacji. Adres tego ustalonego miejsca w rekordzie aktywacji przechowywany jest w odpowiednim rejestrze procesora (np. EBP w procesorach Intel'a).



AC



# Rekord aktywności (rama) procedury



# Rekord aktywności (rama) procedury

Sekwencja wywołania :

1. Procedura wywołująca oblicza parametry aktualne i umieszcza je w rekordzie aktywności procedury wywoływanej.
2. Procedura wywołująca zapamiętuje adres powrotu i starą wartość „*top-sp*” (w polu „*control link*”) w rekordzie aktywności procedury wywoywanej. Następnie modyfikuje wartość „*top-sp*” (patrz poprzedni slajd).
3. Sterowanie przekazywane jest procedurze wywoywanej. Procedura wywoływana zapamiętuje pozostałe informacje związane ze stanem procesora.
4. Procedura wywoływana inicjuje dane lokalne i rozpoczyna wykonywanie „właściwych” czynności obliczeniowych.

# Rekord aktywności (rama) procedury

Sekwencja powrotu:

1. Procedura wywoływana umieszcza zwracany wynik w swoim rekordzie aktywności, usuwa ze stosu część rekordu ze zmiennymi lokalnymi i roboczymi.
2. Wykorzystując informacje zawarte we własnym rekordzie aktywności (pole „control link”) odtwarza poprzednią wartość „top-sp”, a następnie odtwarza poprzedni stan procesora i przekazuje sterowanie procedurze wywołującej zgodnie z adresem powrotu.
3. Procedura wywołująca kopiuje wynik procedury wywoływanej do własnego rekordu aktywności, usuwa ze stosu resztę rekordu aktywności procedury wywoywanej i wznowia własne obliczenia.

# Dostęp do zmiennych nielokalnych

Dwa generalne podejścia (dwie zasady widzialności):

- (1) Statyczne (leksykalne): na podstawie znajomości kodu źródłowego można jednoznacznie przyporządkować deklarację zmiennej do wystąpienia nazwy zmiennej. Najczęściej stosowaną zasadą jest:
  - (a) jeśli nazwa jest zadeklarowana w tym samym bloku, w którym występuje, to właściwą deklaracją jest deklaracja z własnego bloku.
  - (b) jeśli nazwa nie jest zadeklarowana w tym bloku, w którym występuje, to właściwą deklaracją jest deklaracja w tym bloku, który jest najbardziej wewnętrznym w sensie zagnieżdżenia blokiem zawierającym deklarację tej nazwy.



AGH

# Dostęp do zmiennych nielokalnych

Dostęp do danych nielokalnych w językach z widzialnością leksykalną (statyczną) oraz z możliwością zagnieżdżania procedur jest realizowany poprzez mechanizm wiązań dostępu (*access link*) lub mechanizm tablic *display*.

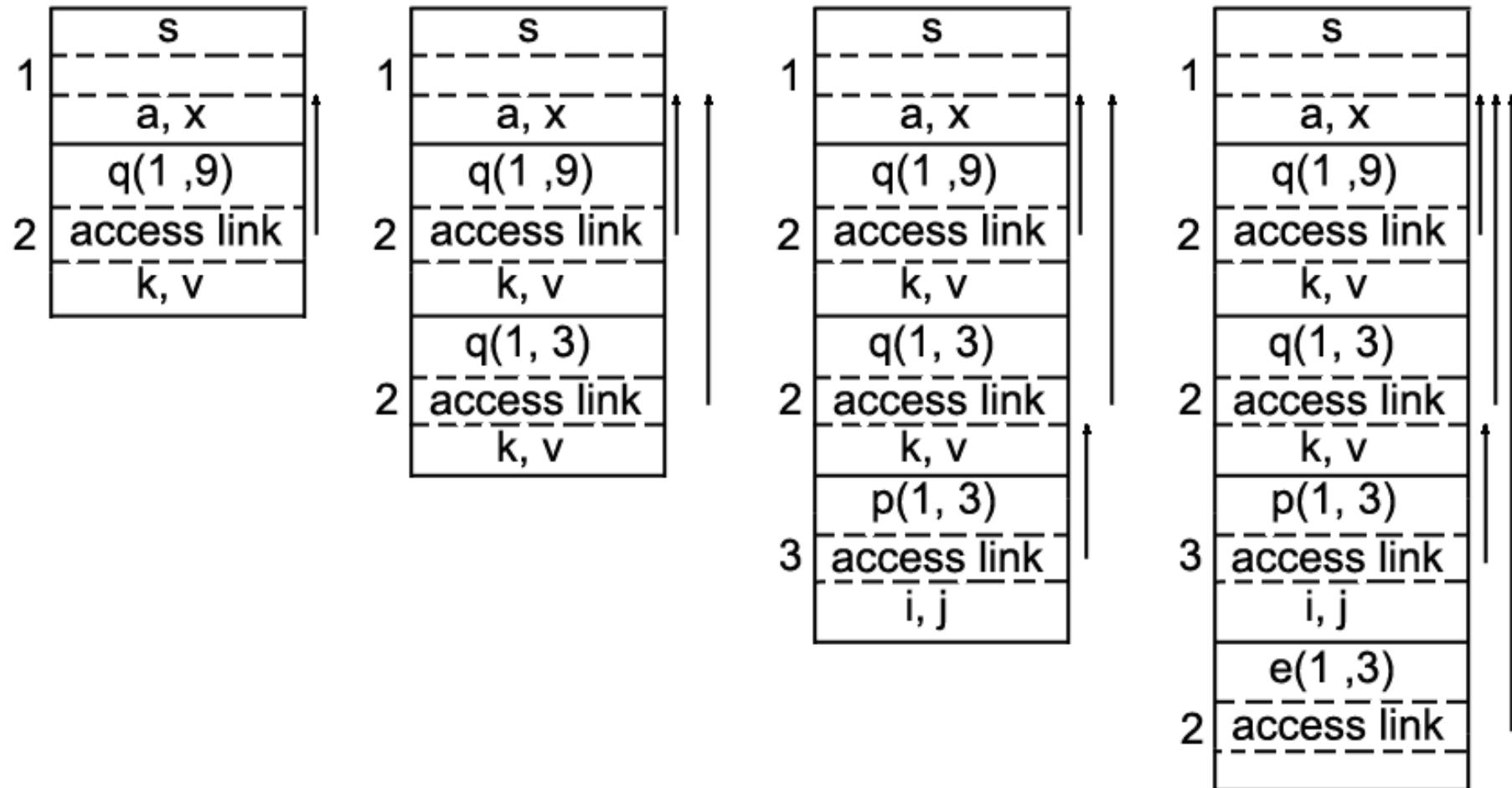
Jeżeli procedura  $p$  jest zagnieżdzona **bezpośrednio** w procedurze  $q$ , wiązanie dostępu (*access link*) w rekordzie aktywacji  $p$  wskazuje wiązanie dostępu w rekordzie dla ostatniej aktywacji  $q$ .

# Zasada statyczna; możliwość zagnieżdżanych procedur

```
program S(input , output);
var a : array [0..10] of integer;
 x : integer;
procedure r;
1
2 var i : integer;
 begin ... a ... end;
procedure e(i,j : integer);
2 begin x := a[i] ; a[i] := a[j]; a[j] := x; er
procedure q(m,n : integer);
 var k ,v :integer;
 function p(y,z : integer) : integer;
2
3 var i,j :integer;
 begin ... a ... v ...
 ... q (i,j); ...
 end;
 begin ... end;
begin ... end;
begin ... end.
```

poziomy zagnieżdżenia

# Zasada statyczna; możliwość zagnieżdżanych procedur



# Zasada statyczna; możliwość zagnieżdżanych procedur

Chcąc przy widzialności leksykalnej w procedurze  $p$  uzyskać dostęp do poszukiwanej zmiennej nielokalnej z procedury  $q$  należy przejść tyle razy przez łańcuch wiązań dostępu (*access link*), ile wynosi różnica poziomów zagnieżdżenia procedur  $p$  i  $q$ , a po dotarciu do rekordu procedury  $q$  należy od ustalonego miejsca w tym rekordzie odliczyć odpowiednie przesunięcie (*offset*) do poszukiwanej zmiennej nielokalnej.

# Zasada statyczna; możliwość zagnieżdżanych procedur

Niech procedura „ $p$ ” o poziomie zagnieżdżenia  $n_p$  zawiera wystąpienie zmiennej „ $a$ ” deklarowanej na poziomie zagnieżdżenia  $n_a$ ,  $n_a \leq n_p$

Pamięć dla „ $a$ ” może być znaleziona w następujący sposób :

1. Gdy sterowanie jest w „ $p$ ”, rekord aktywności „ $p$ ” jest na wierzchołku stosu. Należy  $(n_p - n_a)$  razy „przejść” przez łańcuch wskazań „*access link*”. Wartość  $(n_p - n_a)$  jest znana na etapie komplikacji. W ten sposób osiągnie się rekord aktywności, w którym zlokalizowana jest pamięć dla „ $a$ ”.
2. Właściwa lokalizacja „ $a$ ” jest określona poprzez znany na etapie komplikacji offset wewnątrz rekordu aktywności procedury zawierającej deklarację „ $a$ ”

# Zasada statyczna; możliwość zagnieżdżanych procedur

Przy ustawianiu wiązań dostępu w sekwencji wywołania nowej aktywacji wykorzystujemy informację o poziomie zagnieżdżenia procedury wywołującej i procedury wywoływanej oraz w pewnych przypadkach dotychczasowe wiązania dostępu z rekordów aktywacji znajdujących się na stosie.

# Uzupełnienie sekwencji czynności dokonywanych przy wywoływaniu

Procedura „ $p$ ” o poziomie zagnieżdżenia  $n_p$  wywołuje procedurę „ $x$ ” o poziomie zagnieżdżenia  $n_x$ .

(1) Przypadek  $n_p < n_x$

Wówczas „ $x$ ” jest bardziej zagłębiona niż „ $p$ ”, „ $x$ ” jest zanurzone w „ $p$ ”.

„Access link” w procedurze wywoływanej musi wskazywać rekord aktywności procedury wywołującej.

(2) Przypadek  $n_p \geq n_x$

Wówczas „ $x$ ” nie jest bardziej zagłębione niż „ $p$ ”. „Przechodząc”  $(n_p - n_x + 1)$  razy łańcuchem wskazań „access link” poczynając z rekordu aktywności procedury wywołującej dojdzie się do rekordu aktywności najbardziej wewnętrznej procedury zawierającej zarówno „ $x$ ” jak i „ $p$ ”. „Access link” w procedurze wywoywanej powinien zawierać adres identyczny jak zawartość „access link” osiągniętego po przejściu  $(n_p - n_x)$  razy łańcucha wskazań „access link” poczynając od „access link” procedury wywołującej.

W każdym przypadku „access link” w rekordzie aktywności procedury wywoywanej musi być określany przez procedurę wywołującą przed przekazaniem sterowania do procedury wywoywanej.

# Przykład

```
program param(input , output) ;

procedure b(function h(n:integer) :integer) ;
 begin writeln(h(2)) end; {b}

procedure c;
 var m:integer;
 function f(n:integer) :integer;
 begin f:=m+n end; {f}
 begin m:=0; b(f) end; {c}

begin
 c;

end.
```

↑  
zakres deklaracji „m”  
↓

# Przykład

```

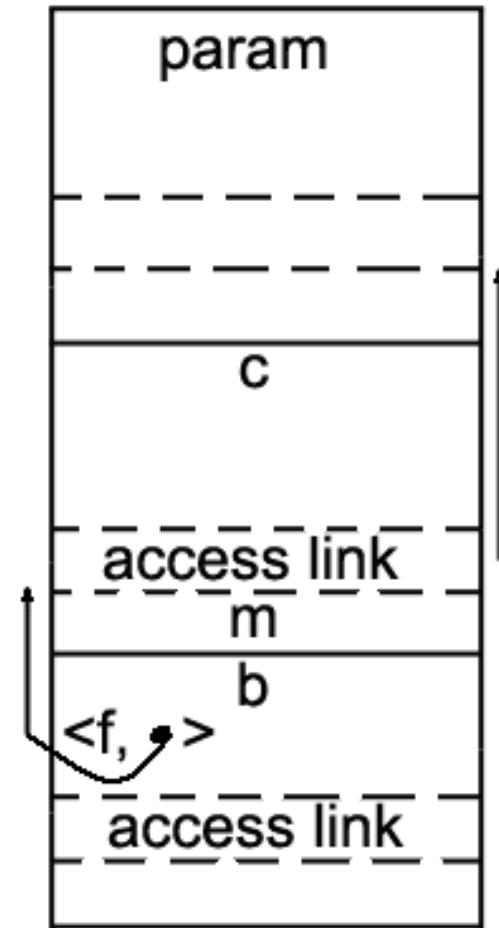
program param(input , output);
 procedure b(function h(n:integer) : integer);
 begin writeln(h(2)) end; {b}

 procedure c;
 var m:integer;
 function f(n:integer) : integer;
 begin f:=m+n end; {f}
 begin m:=0; b(f) end; {c}
 begin
 c;
 end.

```

procedura przekazywana  
 jako parametr  
 musi być “wyposażona”  
 w obliczony algorytmem  
 “access link”

zakres deklaracji „m”



# Tablice display

(procedury nie są przekazywane jako parametry)

Dostęp do zmiennych nielokalnych, szybszy niż za pomocą wiązań *access-link*, można uzyskać, stosując tablicę  $d$  wskaźników do rekordów aktywacji, zwaną *display*. Dzięki tej tablicy pamięć dla zmiennej nielokalnej a na głębokości zagnieżdżenia  $i$  może zostać znaleziona w rekordzie aktywacji wskazywanym przez element tablicy  $d[i]$ .

Gdy nowy rekord aktywacji jest tworzony dla procedury o głębokości zagnieżdżenia  $i$ , należy:

- zapamiętać starą wartość  $d[i]$  w nowym rekordzie aktywacji,
- pozycji  $d[i]$  w tablicy *display* przypisać wskaźnik do nowego rekordu aktywacji.

# Tablice display

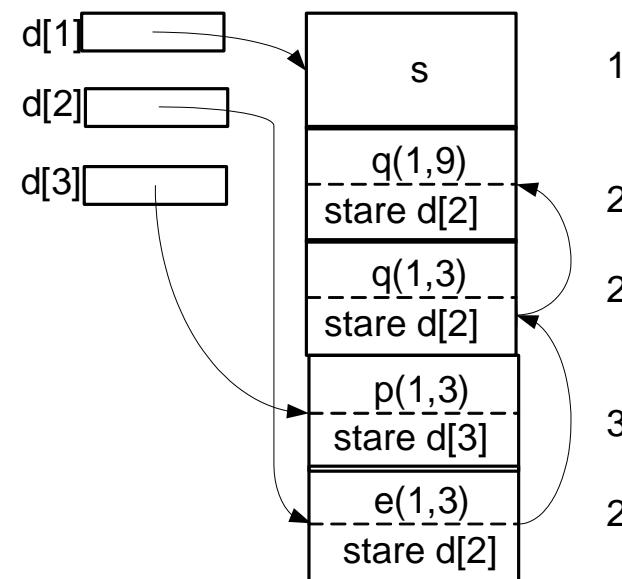
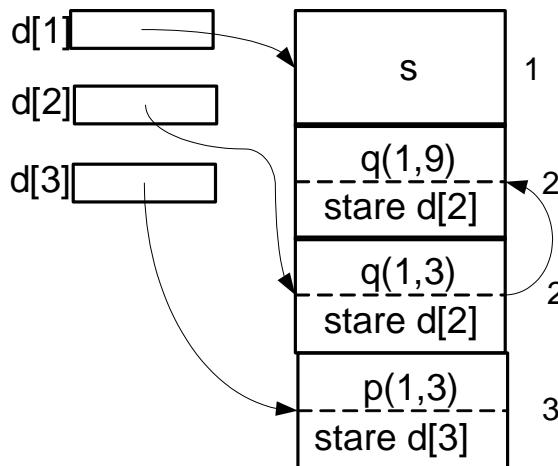
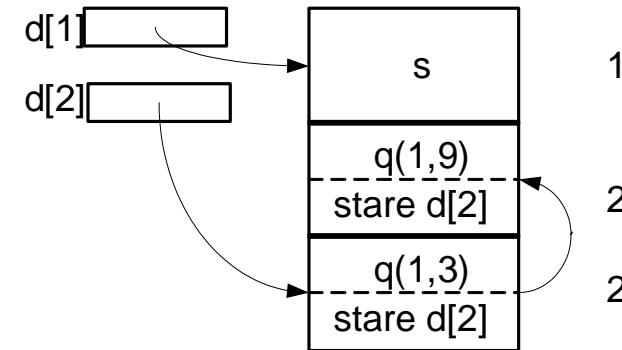
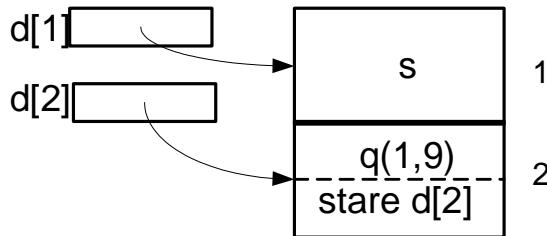
(procedury nie są przekazywane jako parametry)

W mechanizmie tablic *display* przy widzialności leksykalnej pamięć dla zmiennej nielokalnej zlokalizowanej na głębokości zagnieżdżenia *i* może zostać znaleziona w rekordzie aktywacji wskazywanym przez element tablicy *display[i]*.

Elementy tablic *display* muszą się zmieniać, gdy występuje nowa aktywacja, a także wtedy, gdy sterowanie powraca z aktywacji i należy przywrócić poprzednią zawartość elementu tablicy *display*, dlatego też w czasie nowej aktywacji należy w nowym rekordzie aktywacji zapamiętać poprzednią zawartość odpowiedniego elementu tablicy *display*.

# Tablice display

**AGH** (procedury nie są przekazywane jako parametry)



# Zasada widzialności dynamicznej

- Przyporządkowanie deklaracji do wystąpienia zmiennej odbywa się w trakcie wykonania programu. Najczęściej stosowana zasada: przypisanie nielokalnej nazwy do określonej lokalizacji w pamięci nie ulega zmianie w czasie przejścia sterowania do nowej procedury (nazwa nielokalna w wywoływanej procedurze odwołuje się do tej samej pamięci, co w procedurze wywołującej).



AGH

# Zasada dynamiczna - przykład

```
program dynamic(input, output);

var r : real;

procedure show;

begin
 write (r:5:3)

end;

procedure small;

var r : real;

begin
 r := 0.125;
 show;
end;

begin
 r := 0.25;
 show; small; writeln;
 show; small; writeln;
end.
```

*zmienna "r" w procedurze "show" jest  
nielokalna*

Zasada statyczna

Wynik programu

0.250 0.250  
0.250 0.250

Zasada dynamiczna:

Wynik programu  
0.250 0.125  
0.250 0.125

# Zasada dynamiczna

Implementacja wiązań dynamicznych może być dwojaka:

- Dostęp głęboki (pewna analogia do zasady statycznej realizowanej poprzez wskaźniki access-link). Nie wykorzystuje się wiązań typu access-link i używa się wiązań typu control-link do przeszukiwania stosu i poszukiwania pierwszego rekordu aktywacji zawierającego wartość dla tej nazwy nielokalnej. „Głębokości” przeszukiwania stosu nie można wyznaczyć w czasie kompilacji.

# Zasada dynamiczna - przykład

AGH

```
program dynamic(input, output) ;

var r : real;

procedure show;

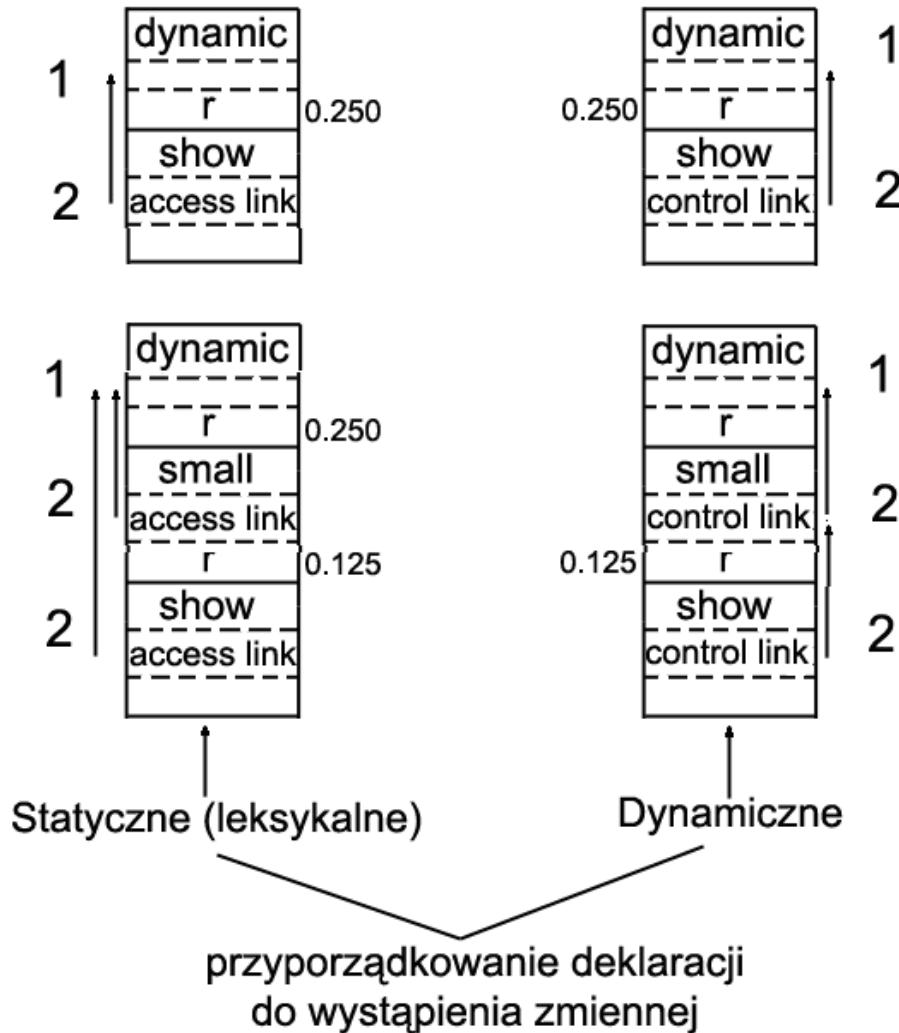
begin
 write (r:5:3)
end;

procedure small;

var r : real;

begin
 r := 0.125;
 show;
end;

begin
 r := 0.25;
 show; small; writeln;
 show; small; writeln;
end.
```



# Zasada dynamiczna

Drugi rodzaj implementacji wiązań dynamicznych:

- Dostęp płytka (pewna analogia do zasady statycznej realizowanej poprzez tablice *display*).  
Przechowywanie obecnej wartości dla każdej nazwy w pamięci zarezerwowanej statycznie. Gdy następuje wywołanie procedury *p* nazwa lokalna *n* zajmuje pamięć zarezerwowaną statycznie dla *n*. Poprzednia wartość *n* jest przechowywana w rekordzie aktywacji dla *p* i należy ją przywrócić do pamięci statycznej podczas sekwencji powrotu procedury *p*.

# Alokacja rekordów aktywności na stercie

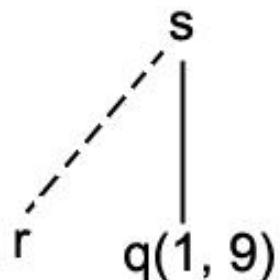
Rekordów aktywności procedur nie można alokować na stosie w dwóch przypadkach :

- Gdy wartości zmiennych lokalnych mają być zachowane po opuszczeniu przez sterowanie procedury.
- Gdy wywoływana procedura może pozostać aktywna, pomimo, że wywołującą ją procedura przestała być aktywna.

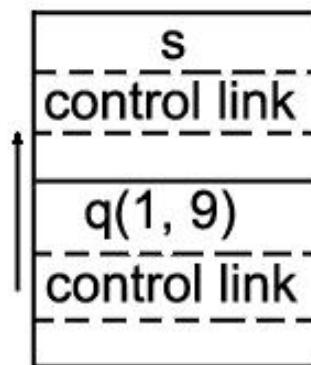
W obu tych przypadkach dealokacja rekordu aktywności procedury, która przestała być aktywna, nie może następować zgodnie z zasadą Last-In First-Out. Miejscem alokacji rekordów aktywności jest wówczas sterta (heap) a nie stos (stack).

# Alokacja rekordów aktywności na stercie

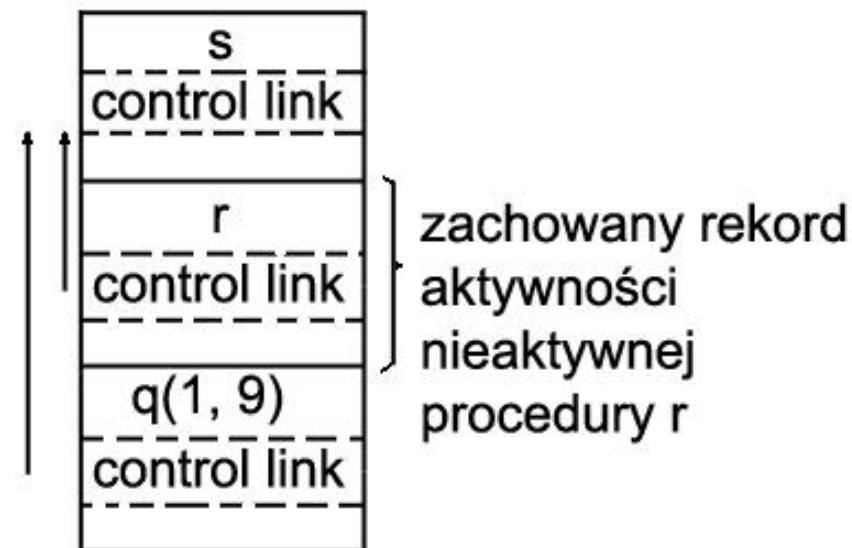
Drzewo aktywności



Rekordy aktywności alokowane na stosie



Rekordy aktywności alokowane na stercie



# Przekazywanie parametrów przez wartość

## Przekazywanie parametrów do procedury przez wartość (call-by-value)

- (a) Parametr formalny jest traktowany jako nazwa lokalna, więc pamięć dla niego jest rezerwowana w rekordzie aktywności procedury wywoływanej
- (b) Procedura wywołującą oblicza parametry aktualne i umieszcza ich wartości (*r-wartości*) w rekordzie aktywności procedury wywoywanej w miejscu przeznaczonym na parametry formalne

# Przekazywanie parametrów przez wartość

Przykład :

```
var a, b : integer;

procedure swap (x, y :integer);
 var temp : integer;

begin
 temp := x;
 x := y;
 y := temp;
end;

begin
 a := 1 b := 2;
 swap (a, b);
 writeln('a =', a, 'b =', b);
 /* a = 1 b = 2 */
end;
```

```
a := 1
b := 2
x := a
y := b
temp := x
x := y
y := temp
write(a,b)
```

# Przekazywanie parametrów przez referencję

## Przekazywanie parametrów do procedury przez adres (referencję, lokację, call-by-reference)

- (a) Jeżeli aktualny parametr jest nazwą lub wyrażeniem posiadającym *l-wartość* (adres), wówczas przekazywana jest *l-wartość* do rekordu aktywności procedury wywoływanej;
- (b) Jeżeli aktualny parametr jest wyrażeniem nie posiadającym *l-wartości* (np.:  $a + b$  czy też  $2 + x$ ) wówczas wyrażenie jest obliczane, jego wartość (*r-wartość*) umieszczana jest w obszarze roboczym i adres tego położenia roboczego jest przekazywany do procedury wywoywanej.

# Przekazywanie parametrów przez referencję

Przykład :

```
var i : integer;
 a : array[1..20] of integer;

procedure swap (var x, y:integer);
 var temp : integer;

begin
 temp := x;
 x := y;
 y := temp;
end;

begin
 ... swap(i , a[i]); ...
end.
```

*x := &i  
y := &a[i]  
temp := \*x  
\*x := \*y  
\*y := temp  
Jeśli  $I_0$  jest początkową  
wartością zmiennej i to  
po wykonaniu  
procedury mamy:  
 $i = a[I_0]$   
 $a[I_0] = I_0$*

# Przekazywanie parametrów przez nazwę

## Przekazywanie parametrów do procedury przez nazwę (call-by-name)

- (a) Procedura wywoływana traktowana jest jakby była makrosem; wywołanie procedury jest w miejscu wywołania „zastępowane” ciałem procedury, przy czym parametry formalne są zastępowane „dosłownie” parametrami aktualnymi (rozwinięcie w miejscu wywołania, *in-line expansion*)
- (b) Lokalne zmienne w procedurze wywoływanie są rozróżniane od zmiennych o tych samych nazwach w procedurze wywołującej, tzn. można wyobrazić sobie, że każda zmienna lokalna jest przemianowana przed rozpoczęciem „rozwijania makrosa”
- (c) Parametry aktualne są ujmowane w nawiasy wszędzie tam, gdzie jest to konieczne dla zachowania ich integralności.

# Przekazywanie parametrów przez nazwę

Przykład:

```
var i : integer;
 a : array [1..10] of integer;

procedure swap(name x, y:integer);
 var temp : integer;
begin
 temp := x;
 x := y;
 y := temp;
end;

begin
 (...)

 swap(i,a[i]); (...)

end.
```

temp := i  
i := a[i]  
a[i] := temp

*Jeśli  $I_0$  jest początkową wartością zmiennej  $i$ , to po wykonaniu procedury mamy:*

$i = a[I_0]$   
 $a[a[I_0]] = I_0$   
zamiast  $a[I_0] = I_0$

# Przekazywanie parametrów metodą „przepisz-odtwórz”

## Przekazywanie parametrów metodą mieszana „przepisz-odtwórz” (*copy-restore linkage*)

- (a) Przed przekazaniem sterowanie do procedury wywoływanej obliczane są wartości parametrów aktualnych. Wartości te (*r-wartości*) są przekazywane do procedury wywoływanej jak w wywołaniu przez wartość. Jednocześnie obliczane i zapamiętywane są adresy (*l-wartości*) tych parametrów aktualnych, które mają *l-wartość*.
- (b) Gdy sterowanie powraca z procedury wywoywanej do wywołującej bieżące wartości parametrów formalnych (*r-wartości*) są kopiowane z powrotem do rekordu aktywności procedury wywołującej z wykorzystaniem zapamiętanych adresów (*l-wartości*) parametrów aktualnych. Oczywiście odtwarzane są tylko parametry aktualne mające *l-wartość*.

# Przekazywanie parametrów metodą „przepisz-odtwórz”

Przykład:

Wywołanie procedury "swap" w postaci  $swap(i, a[i])$   
z poprzednich przykładów da wynik poprawny tzn:

$$\left. \begin{array}{l} i = a[I_0] \\ a[I_0] = I_0 \end{array} \right\} \text{gdzie } I_0 \text{ początkowa wartość zmiennej "i"}$$

Przykład:

```
var a: integer;

procedure unsafe(copy-restore x: integer);
begin
 x := 2;
 a := 0;
end;

begin;
 a := 1;
 unsafe(a);
 writeln(a);
end.
```

call-by-reference  
 $a := 1$   
 $x := \&a$   
 $*x := 2$   
 $a := 0$   
wynik: writeln(a) = 0

copy-restore  
 $a := 1$   
 $l\_val\_act := \&a$   
 $x := a$   
 $x := 2$   
 $a := 0$   
 $*l\_val\_act := x$   
wynik: writeln(a) = 2



AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

## Generacja kodu pośredniego

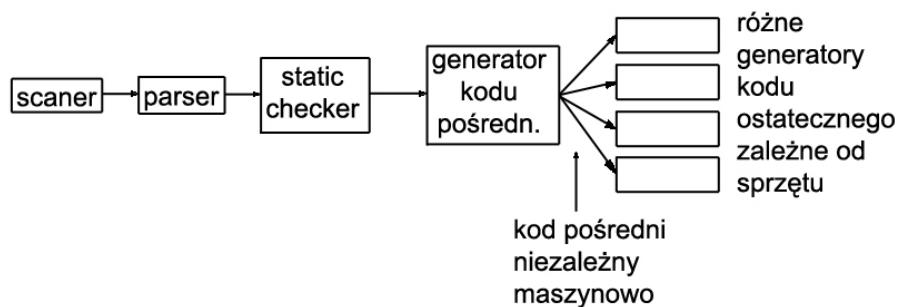
### Teoria kompilacji

Dr inż. Janusz Majewski  
Katedra Informatyki



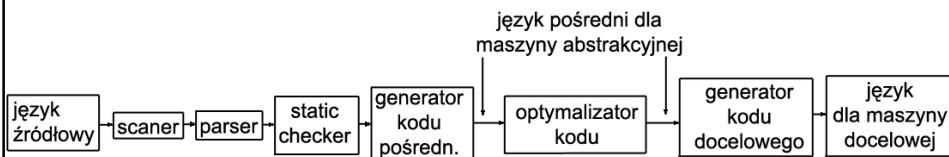
## Przyczyny dwustopniowego tłumaczenia

- Łatwość generowania kompilatorów tego samego języka dla różnych platform systemowo-sprzętowych



## Przyczyny dwustopniowego tłumaczenia

- Łatwość przeprowadzania optymalizacji na bazie kodu pośredniego niezależnie od sprzętu



## Języki kodu pośredniego

Języki kodu pośredniego są językami dla pewnej maszyny abstrakcyjnej :

- Odwrotna notacja polska (notacja postfiksowa) → maszyna stosowa
- Drzewa syntaktyczne lub grafy skierowane acykliczne
- Kod trójadresowy



## Przykład – translacja wyrażeń do odwrotnej notacji polskiej (ONP)

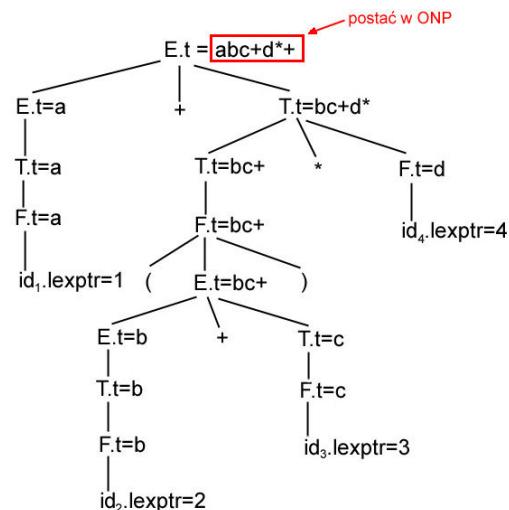
|                         |                                                    |
|-------------------------|----------------------------------------------------|
| $E \rightarrow E_1 + T$ | $E.t \leftarrow E_1.t \parallel T.t \parallel '+'$ |
| $E \rightarrow T$       | $E.t \leftarrow T.t$                               |
| $T \rightarrow T_1 * F$ | $T.t \leftarrow T_1.t \parallel F.t \parallel '*'$ |
| $T \rightarrow F$       | $T.t \leftarrow F.t$                               |
| $F \rightarrow (E)$     | $F.t \leftarrow E.t$                               |
| $F \rightarrow id$      | $F.t \leftarrow name(id.lexptr)$                   |

gdzie:  $\parallel$  - operator konkatenacji tekstów

Rozważane słowo źródłowe: **a+(b+c)\*d**

Po analizie leksykalnej: **id<sub>1</sub>+id<sub>2</sub>+id<sub>3</sub>)\*id<sub>4</sub>**

| <u>id.lexptr</u> | <u>name(id.lexptr)</u> |
|------------------|------------------------|
| 1                | a                      |
| 2                | b                      |
| 3                | c                      |
| 4                | d                      |



## Maszyna wirtualna działająca w oparciu o ONP

Przykład:

- źródło:  $day := (1461 * y) \text{ div } 4 + (153 * m + 2) \text{ div } 5 + d$
- ONP:  $day\ 1461\ y\ * 4\ \text{div}\ 153\ m\ * 2\ + 5\ \text{div}\ + d\ + :=$

Instrukcje maszyny stosowej:

- **push v** - złożenie stałej na stos
- **rvalue l** - złożenie zawartości l na stos
- **lvalue l** - złożenie adresu l na stos
- **(operacja, np:+)** - wykonanie operacji na dwóch argumentach na wierzchołku stosu i bezpośrednio pod wierzchołkiem, zdjęcie obu argumentów ze stosu złożenie tam wyniku.
- **:=** - r-wartość z wierzchołka stosu przesyłana jest do pamięci pod adres ( l-wartość) znajdujący się bezpośrednio pod wierzchołkiem. Obie wartości są zdejmowane ze stosu

## Program dla maszyny stosowej

- źródło:  $\text{day} := (1461 * \text{y}) \text{div } 4 + (153 * \text{m} + 2) \text{div } 5 + \text{d}$
- ONP:  $\text{day} 1461 \text{y} * 4 \text{div} 153 \text{m} * 2 + 5 \text{div} + \text{d} + :=$

- Tłumaczenie dla maszyny stosowej:

```

lvalue day
push 1461
rvalue y
*
push 4
div
push 153
rvalue m
*
push 2
+
push 5
div
+
rvalue d
+
:=

```

## Przykład – translacja instrukcji przypisania do kodu trójadresowego

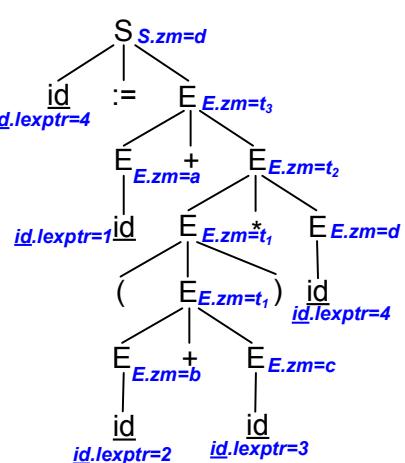
|                                     |                                                                                           |
|-------------------------------------|-------------------------------------------------------------------------------------------|
| $S \rightarrow \underline{id} := E$ | $S.zm \leftarrow \text{name}(\underline{id}.lexptr)$<br>$\text{gen}(S.zm    ":"    E.zm)$ |
| $E \rightarrow E_1 + E_2$           | $E.zm \leftarrow \text{new\_temp}()$<br>$\text{gen}(E.zm    "+"    E_1.zm    E_2.zm)$     |
| $E \rightarrow E_1 * E_2$           | $E.zm \leftarrow \text{new\_temp}()$<br>$\text{gen}(E.zm    "*"    E_1.zm    E_2.zm)$     |
| $E \rightarrow (E_1)$               | $E.zm \leftarrow E_1.zm$                                                                  |
| $E \rightarrow id$                  | $E.zm \leftarrow \text{name}(\underline{id}.lexptr)$                                      |

gdzie:  $\parallel$  - operator konkatenacji tekstów

Rozważane słowo źródłowe:  $d := a + (b + c) * d$

Po analizie leksykalnej:  $\underline{id}_4 := \underline{id}_1 + (\underline{id}_2 + \underline{id}_3) * \underline{id}_4$

| $\underline{id}.lexptr$ | $\text{name}(\underline{id}.lexptr)$ |
|-------------------------|--------------------------------------|
| 1                       | a                                    |
| 2                       | b                                    |
| 3                       | c                                    |
| 4                       | d                                    |



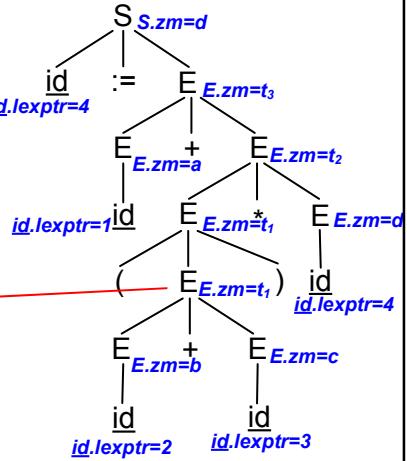
## Przykład – translacja instrukcji przypisania do kodu trójadresowego

|                           |                                                                                   |
|---------------------------|-----------------------------------------------------------------------------------|
| $S \rightarrow id := E$   | $S.zm \leftarrow name(id.lexptr)$<br>$gen(S.zm    " := "    E.zm)$                |
| $E \rightarrow E_1 + E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "+"    E_2.zm)$ |
| $E \rightarrow E_1 * E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "*"    E_2.zm)$ |
| $E \rightarrow (E_1)$     | $E.zm \leftarrow E_1.zm$                                                          |
| $E \rightarrow id$        | $E.zm \leftarrow name(id.lexptr)$                                                 |

słowo źródłowe: **d:=a+(b+c)\*d**

Tłumaczenie:

**$t_1 := b + c$**



## Przykład – translacja instrukcji przypisania do kodu trójadresowego

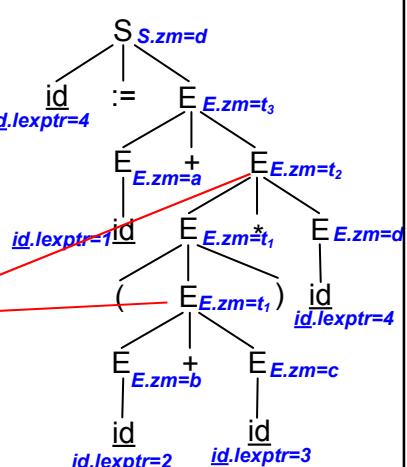
|                           |                                                                                   |
|---------------------------|-----------------------------------------------------------------------------------|
| $S \rightarrow id := E$   | $S.zm \leftarrow name(id.lexptr)$<br>$gen(S.zm    " := "    E.zm)$                |
| $E \rightarrow E_1 + E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "+"    E_2.zm)$ |
| $E \rightarrow E_1 * E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "*"    E_2.zm)$ |
| $E \rightarrow (E_1)$     | $E.zm \leftarrow E_1.zm$                                                          |
| $E \rightarrow id$        | $E.zm \leftarrow name(id.lexptr)$                                                 |

słowo źródłowe: **d:=a+(b+c)\*d**

Tłumaczenie:

**$t_1 := b + c$**

**$t_2 := t_1 * d$**



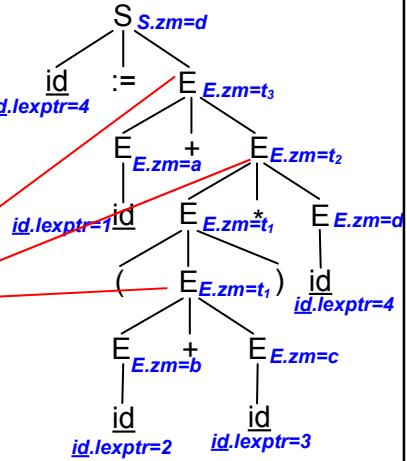
## Przykład – translacja instrukcji przypisania do kodu trójadresowego

|                           |                                                                                   |
|---------------------------|-----------------------------------------------------------------------------------|
| $S \rightarrow id := E$   | $S.zm \leftarrow name(id.lexptr)$<br>$gen(S.zm    " := "    E.zm)$                |
| $E \rightarrow E_1 + E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "+"    E_2.zm)$ |
| $E \rightarrow E_1 * E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "*"    E_2.zm)$ |
| $E \rightarrow (E_1)$     | $E.zm \leftarrow E_1.zm$                                                          |
| $E \rightarrow id$        | $E.zm \leftarrow name(id.lexptr)$                                                 |

słowo źródłowe: **d:=a+(b+c)\*d**

Tłumaczenie:

$t_1 := b + c$   
 $t_2 := t_1 * d$   
 $t_3 := a + t_2$



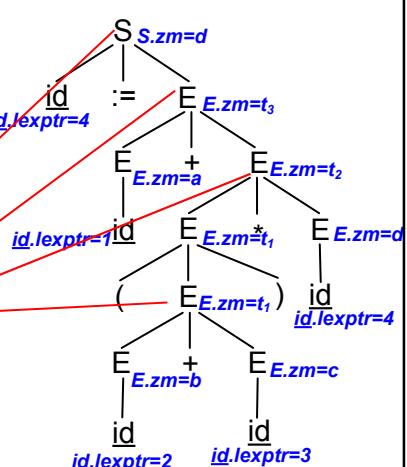
## Przykład – translacja instrukcji przypisania do kodu trójadresowego

|                           |                                                                                   |
|---------------------------|-----------------------------------------------------------------------------------|
| $S \rightarrow id := E$   | $S.zm \leftarrow name(id.lexptr)$<br>$gen(S.zm    " := "    E.zm)$                |
| $E \rightarrow E_1 + E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "+"    E_2.zm)$ |
| $E \rightarrow E_1 * E_2$ | $E.zm \leftarrow new\_temp()$<br>$gen(E.zm    " := "    E_1.zm    "*"    E_2.zm)$ |
| $E \rightarrow (E_1)$     | $E.zm \leftarrow E_1.zm$                                                          |
| $E \rightarrow id$        | $E.zm \leftarrow name(id.lexptr)$                                                 |

słowo źródłowe: **d:=a+(b+c)\*d**

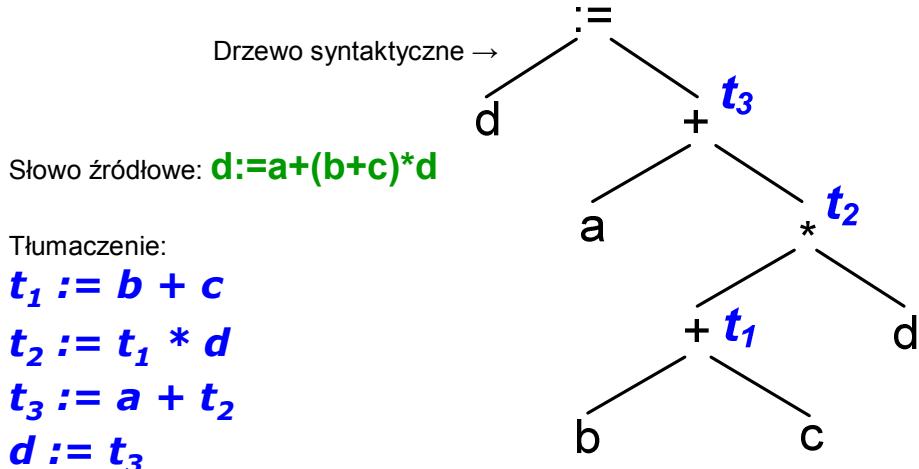
Tłumaczenie:

$t_1 := b + c$   
 $t_2 := t_1 * d$   
 $t_3 := a + t_2$   
 $d := t_3$



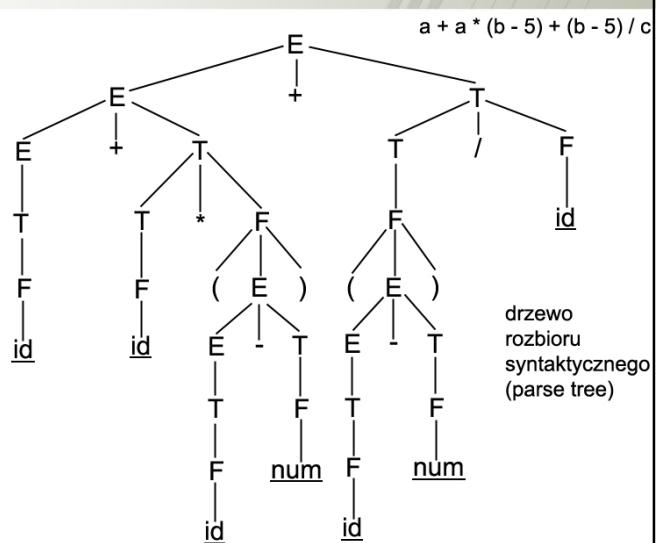


## Przykład – translacja instrukcji przypisania do kodu trójadresowego



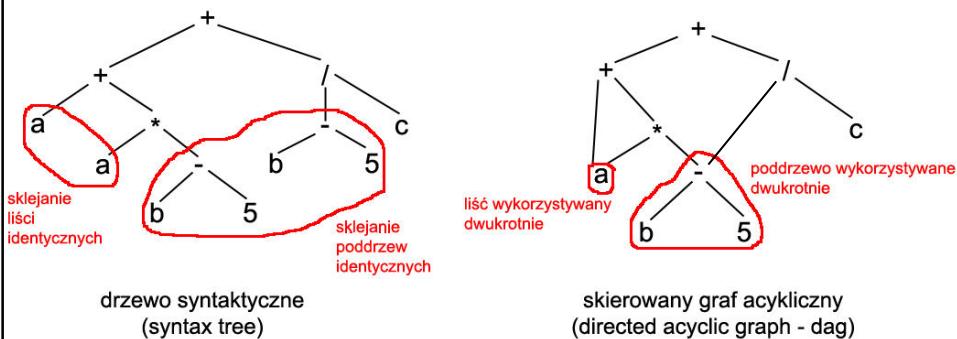
## Przypomnienie: drzewa rozbiórku

$E \rightarrow E + T \mid E - T \mid T$   
 $T \rightarrow T * F \mid T / F \mid F$   
 $F \rightarrow (E) \mid id \mid num$



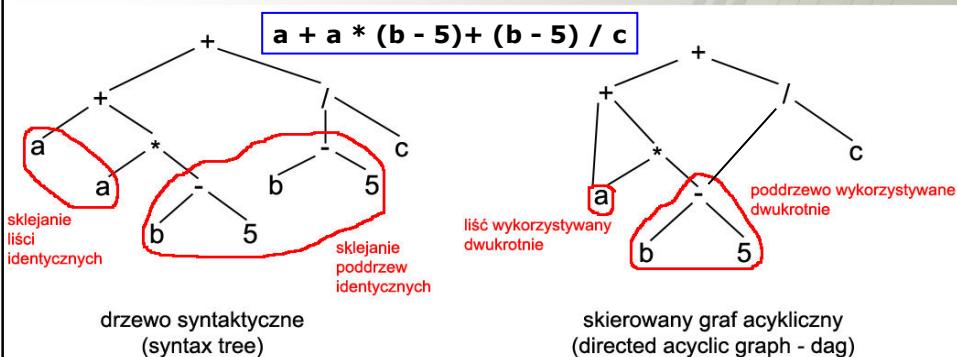
## Drzewa syntaktyczne i dagi

$a + a * (b - 5) + (b - 5) / c$



## Drzewa syntaktyczne i dagi a kod trójadresowy

$a + a * (b - 5) + (b - 5) / c$



$t_1 := b - 5$   
 $t_2 := a * t_1$   
 $t_3 := a + t_2$   
 $t_4 := b - 5$   
 $t_5 := t_4 / c$   
 $t_6 := t_3 + t_5$

$t_1 := b - 5$   
 $t_2 := a * t_1$   
 $t_3 := a + t_2$   
 $t_4 := t_1 / c$   
 $t_5 := t_3 + t_4$



## Drzewa syntaktyczne i dagi a kod trójadresowy

Kod trójadresowy jest zlinearyzowaną reprezentacją drzew syntaktycznych lub skierowanych grafów cyklicznych.



## Kod trójadresowy implementacja

Czwórki:

|     | oper.         | arg 1          | arg 2          | wyn            |
|-----|---------------|----------------|----------------|----------------|
| (0) | <u>uminus</u> | c              |                | t <sub>1</sub> |
| (1) | *             | b              | t <sub>1</sub> | t <sub>2</sub> |
| (2) | <u>uminus</u> | c              |                | t <sub>3</sub> |
| (3) | *             | b              | t <sub>3</sub> | t <sub>4</sub> |
| (4) | +             | t <sub>2</sub> | t <sub>4</sub> | t <sub>5</sub> |
| (5) | :=            | t <sub>5</sub> |                | a              |

Zmienne tymczasowe muszą  
być w tablicy symboli

pojntery do  
tabl. symboli

Reprezentacja pośrednia  
umożliwiająca przedstawianie  
instrukcji

Trójki:

|     | instr |      | oper          | arg 1 | arg 2 |
|-----|-------|------|---------------|-------|-------|
| (0) | (14)  | (14) | <u>uminus</u> | c     |       |
| (1) | (15)  | (15) | *             | b     | (14)  |
| (2) | (16)  | (16) | <u>uminus</u> | c     |       |
| (3) | (17)  | (17) | *             | b     | (16)  |
| (4) | (18)  | (18) | +             | (15)  | (17)  |
| (5) | (19)  | (19) | :=            | a     | (18)  |



## Zestaw instrukcji trójadresowych

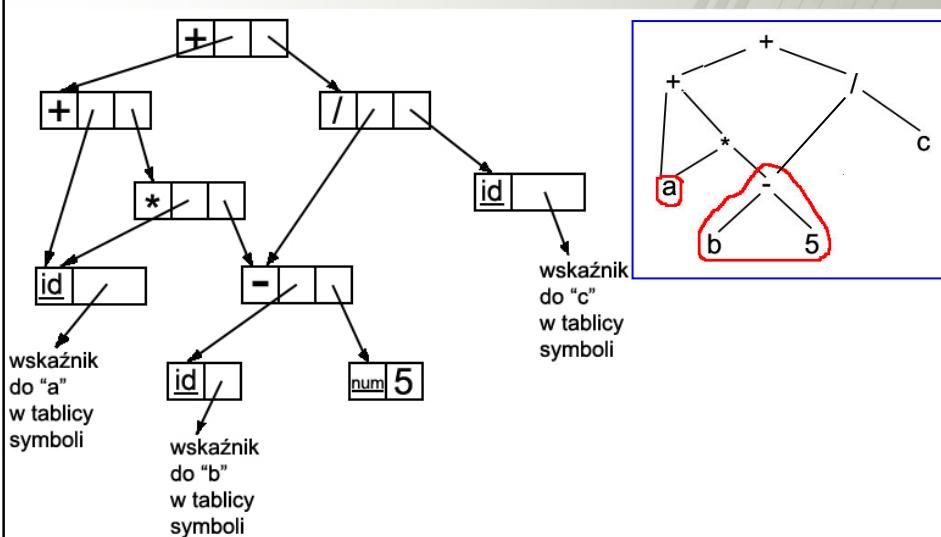
- (a)  $x := y \ op z$  dla operacji dwuargumentowych
- (b)  $x := op y$  dla operacji jednoargumentowych
- (c)  $x := y$  kopiowanie
- (d) goto L skok bezwarunkowy
- (e) if x relop y goto L skok warunkowy
- (f) param x  
call p, n do obsługi procedur  
return y
- (g)  $x := y[i]$  do obsługi tablic  
 $x[i] := y$  *i – odległość elementu od początku tablicy liczona w jednostkach pamięci, np. w bajtach*
- (h)  $x := &y$   
 $x := *y$  do obsługi wskaźników  
 $*x := y$



## Konstruowanie drzew syntaktycznych lub skierowanych grafów acyklicznych dla wyrażeń

|                            |                                                                                     |
|----------------------------|-------------------------------------------------------------------------------------|
| $E \rightarrow E_1 + T$    | $\{E.\text{nptr} \leftarrow \text{mknode}( '+ ', E_1.\text{nptr}, T.\text{nptr})\}$ |
| $E \rightarrow E_1 - T$    | $\{E.\text{nptr} \leftarrow \text{mknode}( '- ', E_1.\text{nptr}, T.\text{nptr})\}$ |
| $E \rightarrow T$          | $\{E.\text{nptr} \leftarrow T.\text{nptr}\}$                                        |
| $T \rightarrow T_1 * F$    | $\{T.\text{nptr} \leftarrow \text{mknode}( '*', T_1.\text{nptr}, F.\text{nptr})\}$  |
| $T \rightarrow T_1 / F$    | $\{T.\text{nptr} \leftarrow \text{mknode}( '/', T_1.\text{nptr}, F.\text{nptr})\}$  |
| $T \rightarrow F$          | $\{T.\text{nptr} \leftarrow F.\text{nptr}\}$                                        |
| $F \rightarrow (E)$        | $\{F.\text{nptr} \leftarrow E.\text{nptr}\}$                                        |
| $F \rightarrow \text{id}$  | $\{F.\text{nptr} \leftarrow \text{mkleaf}(\text{id}, \text{id}.\text{lexptr})\}$    |
| $F \rightarrow \text{num}$ | $\{F.\text{nptr} \leftarrow \text{mkleaf}(\text{num}, \text{num}.\text{val})\}$     |

- (i) `mknode(op, left, right)`
  - (a) sprawdza, czy istnieje wierzchołek (`op`, `left`, `right`), jeśli tak zwraca wskaźnik do tego wierzchołka
  - (b) konstruuje wierzchołek (`op`, `left`, `right`) i zwraca wskaźnik do tego wierzchołka
- (ii) `mkleaf(id ,id.lexptr)`
  - (a) sprawdza, czy istnieje liść (`id`, `id.lexptr`), jeśli tak zwraca wskaźnik do tego liścia
  - (b) konstruuje liść (`id`, `id.lexptr`) i zwraca wskaźnik do tego liścia
- (iii) `mkleaf(num,num.val)`
  - (a) sprawdza, czy istnieje liść (`num`, `num.val`), jeśli tak zwraca wskaźnik do tego liścia
  - (b) konstruuje liść (`num`, `num.val`) i zwraca wskaźnik do tego liścia

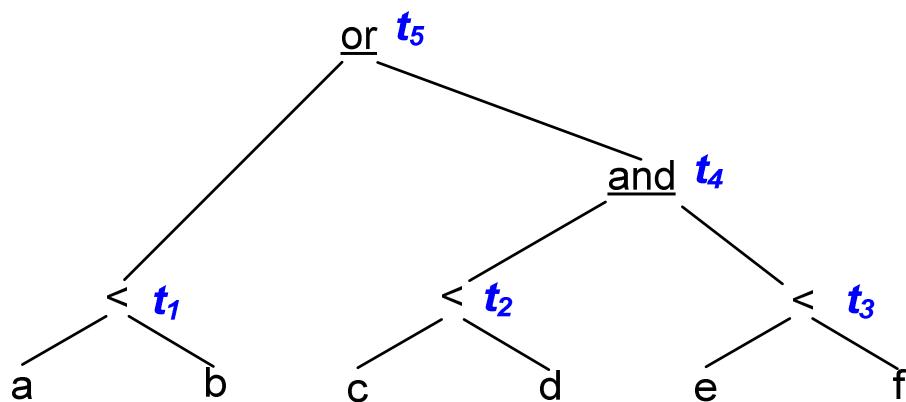


## Translacja wyrażeń logicznych „do końca”

|                                                                  |                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 \text{ or } E_2$                              | $E.place \leftarrow \text{newtemp}$<br>$\text{gen}(E.place := E_1.place \text{ 'or' } E_2.place)$                                                                                                                                                                |
| $E \rightarrow E_1 \text{ and } E_2$                             | $E.place \leftarrow \text{newtemp}$<br>$\text{gen}(E.place := E_1.place \text{ 'and' } E_2.place)$                                                                                                                                                               |
| $E \rightarrow \text{not } E_1$                                  | $E.place \leftarrow \text{newtemp}$<br>$\text{gen}(E.place := \text{'not'} E_1.place)$                                                                                                                                                                           |
| $E \rightarrow \underline{id}_1 \text{ relop } \underline{id}_2$ | $E.place \leftarrow \text{newtemp}$<br>$\text{gen}(\text{'if' } id_1.place \text{ relop.op } id_2.place \text{ 'goto' } \text{nexstat + 3})$<br>$\text{gen}(E.place := '0')$<br>$\text{gen}(\text{'goto' } \text{nextstat + 2})$<br>$\text{gen}(E.place := '1')$ |
| $E \rightarrow \text{true}$                                      | $E.place \leftarrow \text{newtemp};$<br>$\text{gen}(E.place := '1')$                                                                                                                                                                                             |
| $E \rightarrow \text{false}$                                     | $E.place \leftarrow \text{newtemp};$<br>$\text{gen}(E.place := '0')$                                                                                                                                                                                             |
| $E \rightarrow (E_1)$                                            | $E.place \leftarrow E_1.place$                                                                                                                                                                                                                                   |

## Translacja wyrażeń logicznych „do końca”

Przykład: **a < b or c < d and e < f**



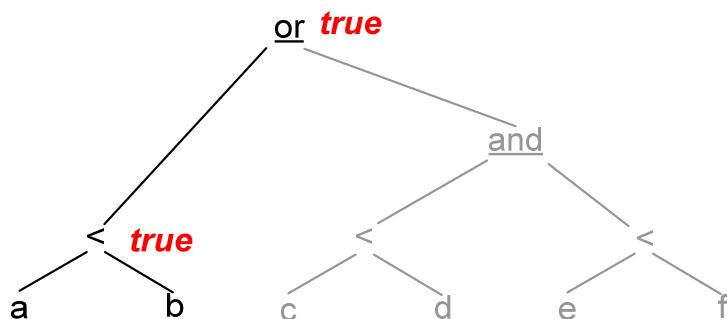
## Translacja wyrażeń logicznych „do końca”

Przykład: **a < b or c < d and e < f**

|                                   |                                                                      |
|-----------------------------------|----------------------------------------------------------------------|
| <b>100 : if a &lt; b goto 103</b> | <b>107 : t<sub>2</sub> := 1</b>                                      |
| <b>101 : t<sub>1</sub> := 0</b>   | <b>108 : if e &lt; f goto 111</b>                                    |
| <b>102 : goto 104</b>             | <b>109 : t<sub>3</sub> := 0</b>                                      |
| <b>103 : t<sub>1</sub> := 1</b>   | <b>110 : goto 112</b>                                                |
| <b>104 : if c &lt; d goto 107</b> | <b>111 : t<sub>3</sub> := 1</b>                                      |
| <b>105 : t<sub>2</sub> := 0</b>   | <b>112 : t<sub>4</sub> := t<sub>2</sub> <u>and</u> t<sub>3</sub></b> |
| <b>106 : goto 108</b>             | <b>113 : t<sub>5</sub> := t<sub>1</sub> <u>or</u> t<sub>4</sub></b>  |

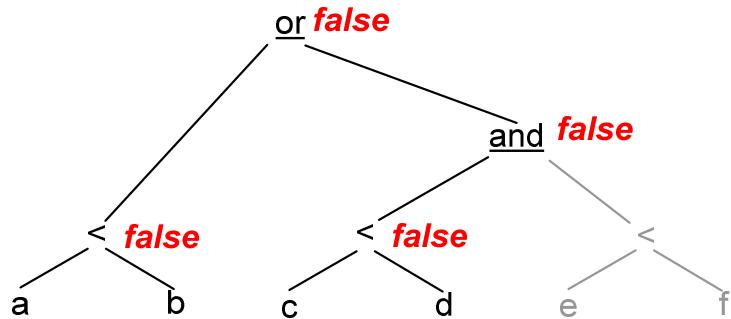
## Translacja wyrażeń logicznych („short – circuit” code)

Przykład: **a < b or c < d and e < f**



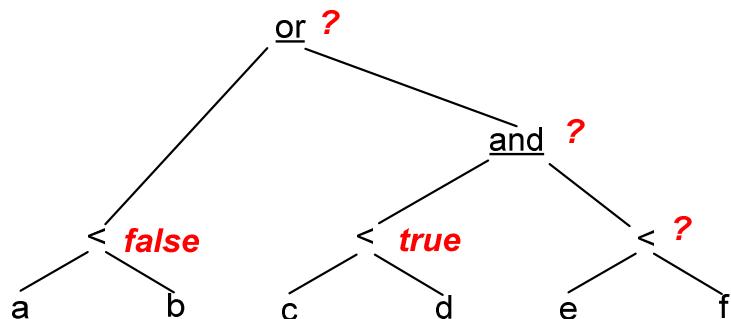
## Translacja wyrażeń logicznych „short – circuit” code

Przykład: **a < b or c < d and e < f**

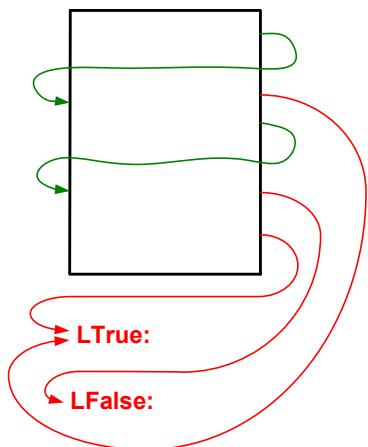


## Translacja wyrażeń logicznych „short – circuit” code

Przykład: **a < b or c < d and e < f**



## Translacja wyrażeń logicznych „short – circuit” code)



## Translacja wyrażeń logicznych „short – circuit” code)

Jeśli  $E$  ma postać  $a < b$ , to można wygenerować

if  $a < b$  goto  $E.true$   
goto  $E.false$

←  
trybuz dziedziczony symbolu  $E$   
= etykieta, do której przechodzi  
sterowanie, gdy  $a < b$  jest „true”

Jeśli  $E$  jest postaci  $E_1 \text{ or } E_2$ , to jeśli  $E_1$  jest „true” wówczas już wiemy  
że  $E$  jest „true” czyli

$E_1.true \leftarrow E.true$

Jeśli nie, to musimy obliczyć  $E_2$ , więc

$E_1.false \leftarrow newlabel$

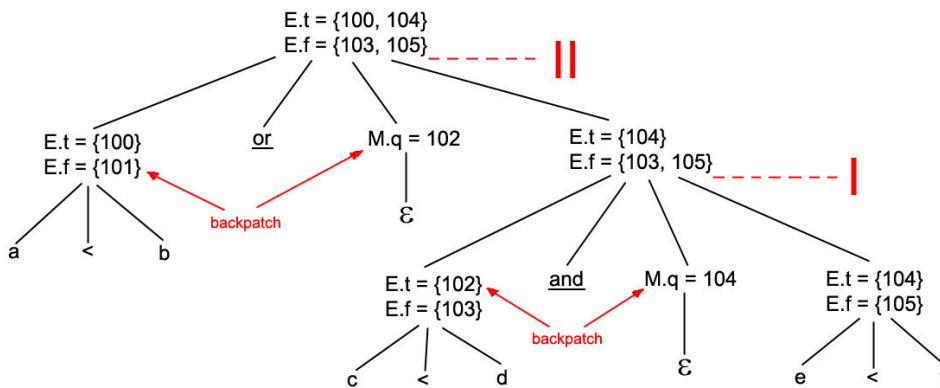
Jeżeli obliczamy  $E_2$  i  $E_2$  jest „true” to  $E$  także jest „true”, w  
przeciwnym przypadku  $E$  jest „false”

$E_2.true \leftarrow E.true$   
 $E_2.false \leftarrow E.false$

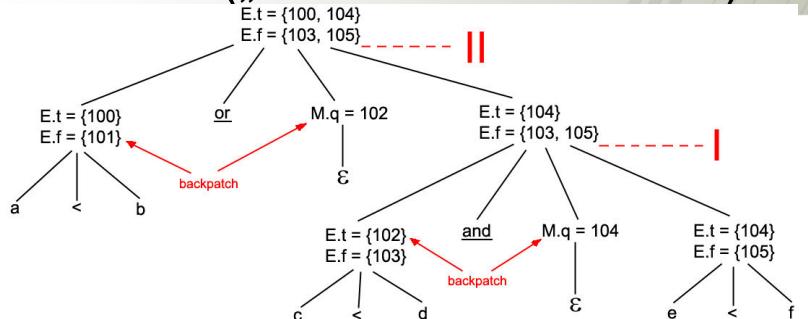
## Translacja wyrażeń logicznych „short – circuit” code)

|                                                                            |                                                                                                                                                                                                                                 |
|----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 \underline{\text{or}} M E_2$                            | backpatch(E <sub>1</sub> .falselist, M.quad);<br>E.truelist $\leftarrow$ merge(E <sub>1</sub> .truelist, E <sub>2</sub> .truelist);<br>E.falselist $\leftarrow$ E <sub>2</sub> .falselist;                                      |
| $E \rightarrow E_1 \underline{\text{and}} M E_2$                           | backpatch(E <sub>1</sub> .truelist, M.quad);<br>E.truelist $\leftarrow$ E <sub>2</sub> .truelist;<br>E.falselist $\leftarrow$ merge(E <sub>1</sub> .falselist, E <sub>2</sub> .falselist);                                      |
| $E \rightarrow \underline{\text{not}} E_1$                                 | E.truelist $\leftarrow$ E <sub>1</sub> .falselist;<br>E.falselist $\leftarrow$ E <sub>1</sub> .truelist;                                                                                                                        |
| $E \rightarrow (E_1)$                                                      | E.truelist $\leftarrow$ E <sub>1</sub> .truelist;<br>E.falselist $\leftarrow$ E <sub>1</sub> .falselist;                                                                                                                        |
| $E \rightarrow \underline{id}_1 \underline{\text{relOp}} \underline{id}_2$ | E.truelist $\leftarrow$ makelist(nextquad());<br>E.falselist $\leftarrow$ makelist(nextquad() + 1);<br>gen('if' $\underline{id}_1$ .place $\underline{\text{relOp}}$ .op $\underline{id}_2$ .place 'goto _');<br>gen('goto _'); |
| $E \rightarrow \underline{\text{true}}$                                    | E.truelist $\leftarrow$ makelist(nextquad());<br>gen('goto _');                                                                                                                                                                 |
| $E \rightarrow \underline{\text{false}}$                                   | E.falselist $\leftarrow$ makelist(nextquad());<br>gen('goto _');                                                                                                                                                                |
| $M \rightarrow \varepsilon$                                                | M.quad $\leftarrow$ nextquad()                                                                                                                                                                                                  |

## Translacja wyrażeń logicznych „short – circuit” code)



## Translacja wyrażeń logicznych „short – circuit” code)



Przed redukcją I

```

100: if a < b goto _
101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _

```

Po redukcji I

```

100: if a < b goto _
101: goto _
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

```

Przed redukcją II

```

100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

```

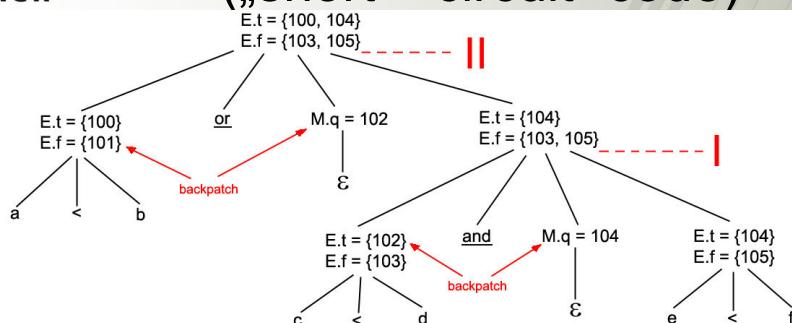
Po redukcji II

```

100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

```

## Translacja wyrażeń logicznych „short – circuit” code)



```

100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

```

Nie zostały wypełnione jedynie miejsca etykiet L.true i L.false z poprzedniego przykładu. Adresy docelowe skoków z linii 100 i 104 (E jest „true”) oraz z linii 103 i 105 (E jest „false”) będą znane po przeanalizowaniu zapisu, w skład którego wchodzi badane w przykładzie wyrażenie boolowskie.



## Translacja wyrażeń logicznych „short – circuit” code

Przykład: **a < b or c < d and e < f**

**if a < b goto L.true**

**goto L1**

**L1: if c < d goto L2**

**goto: L.false**

**L2: if e < f goto L.true**

**goto L.false**

**Po optymalizacji...**

**if a < b goto L.true**

**if c >= d goto L.false**

**if e < f goto L.true**

**goto L.false**



AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

## Optymalizacja kodu pośredniego

### Teoria komplikacji

Dr inż. Janusz Majewski  
Katedra Informatyki



## Analiza przepływu, grafy obliczeń

- Graf obliczeń jest to skierowany graf tworzony na podstawie kodu trójadresowego. Krawędzie (ścieżki) w grafie wskazują możliwą kolejność wykonywania obliczeń. Wierzchołkami grafu są bloki podstawowe. Blok podstawowy jest ciągiem instrukcji trójadresowych, takich że jeśli sterowanie zostanie przekazane do pierwszej instrukcji tego bloku, to opuści blok po wykonaniu ostatniej instrukcji (nie będzie wewnątrz bloku skoków ani rozkazu stopu).



## Analiza przepływu, grafy obliczeń

- Dzielenie kodu trójadresowego na bloki podstawowe :
  - (1) wyodrębniamy pierwsze instrukcje (liderów):
    - (a) pierwsza instrukcja kodu trójadresowego jest liderem
    - (b) każda instrukcja, do której prowadzi skok warunkowy lub bezwarunkowy jest liderem
    - (c) każda instrukcja, która następuje bezpośrednio po skoku warunkowym lub bezwarunkowym jest liderem
  - (2) każdy blok rozpoczyna się swoim liderem i zawiera wszystkie instrukcje, aż do napotkania kolejnego лидera lub końca kodu.



## Analiza przepływu, grafy obliczeń

- Mając podział na bloki podstawowe tworzymy graf obliczeń uwzględniający możliwą kolejność wykonywania obliczeń. Pierwszy blok jest wyróżnionym wierzchołkiem początkowym w grafie. W grafie biegnie krawędź od wierzchołka  $B_i$  do  $B_j$ , jeśli:
  - (a) istnieje skok warunkowy lub bezwarunkowy z ostatniej instrukcji  $B_i$  do pierwszej instrukcji  $B_j$
  - (b)  $B_j$  bezpośrednio następuje po  $B_i$  w kodzie trójadresowym, przy czym  $B_i$  nie kończy się skokiem bezwarunkowym.

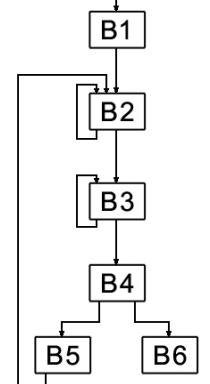
## Przykład: program źródłowy

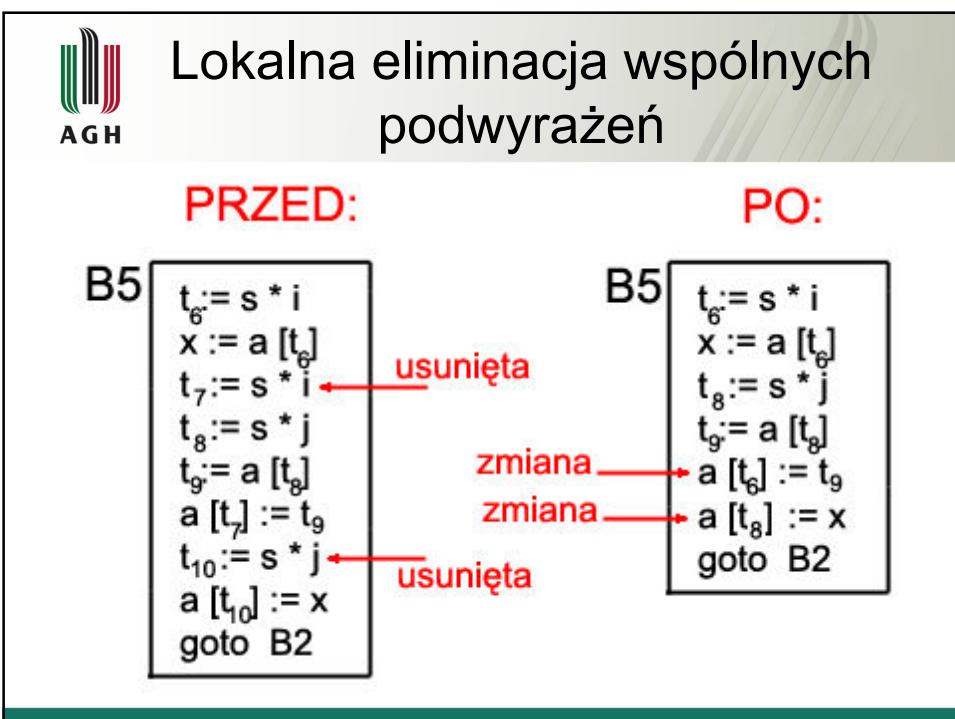
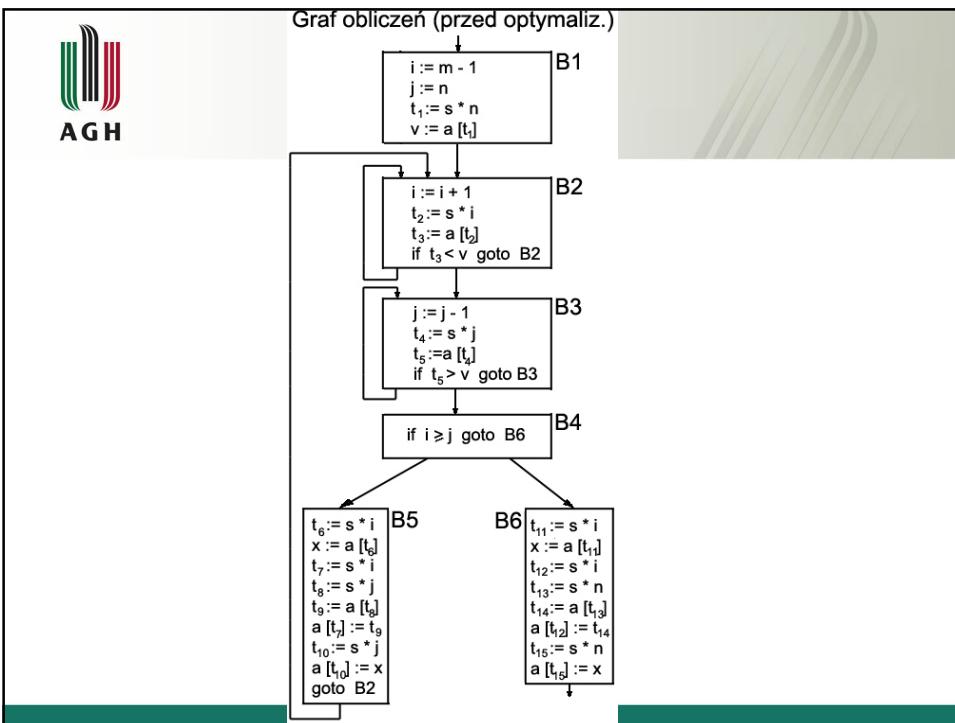
Założenie: stała „s” zawiera `sizeof(int)`

```
void quicksort(m,n)
 int m, n;
 {
 int i, j;
 int v, x;
 if (n <= m) return;
Początek
 i = m - 1; j = n; v = a[n];
 while(1) {
 do i = i + 1; while (a[i] < v);
 do j = j - 1; while (a[j] > v);
 if (i >= j) break;
 x = a[i]; a[i] = a[j]; a[j] = x;
 }
 x = a[i]; a[i] = a[n]; a[n] = x;
Koniec
 quicksort(m, j); quicksort(i + 1, n);
 }
```

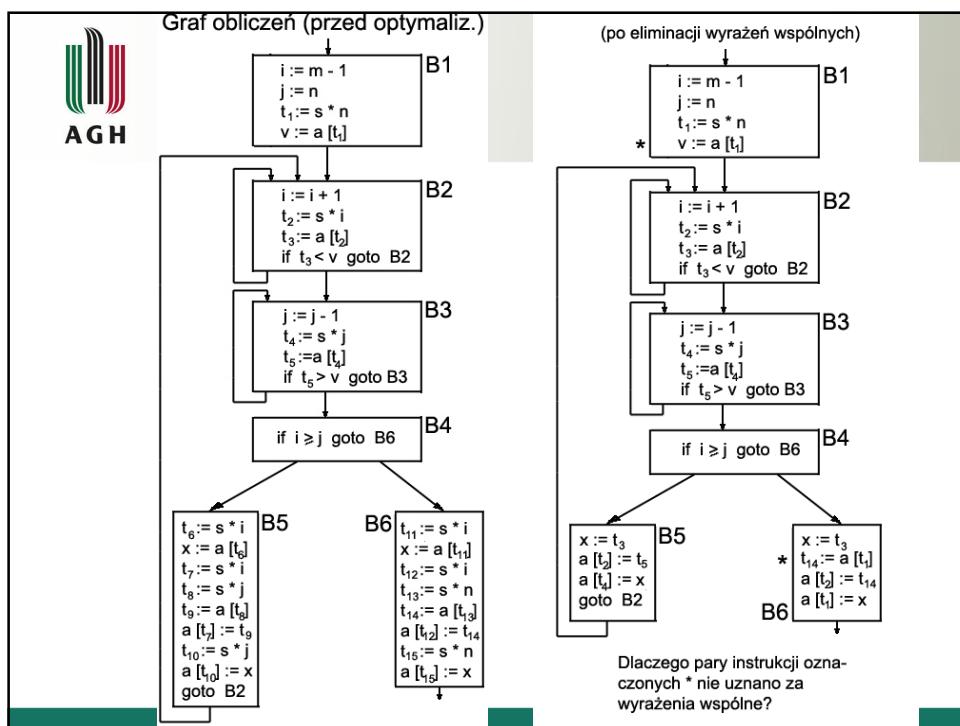
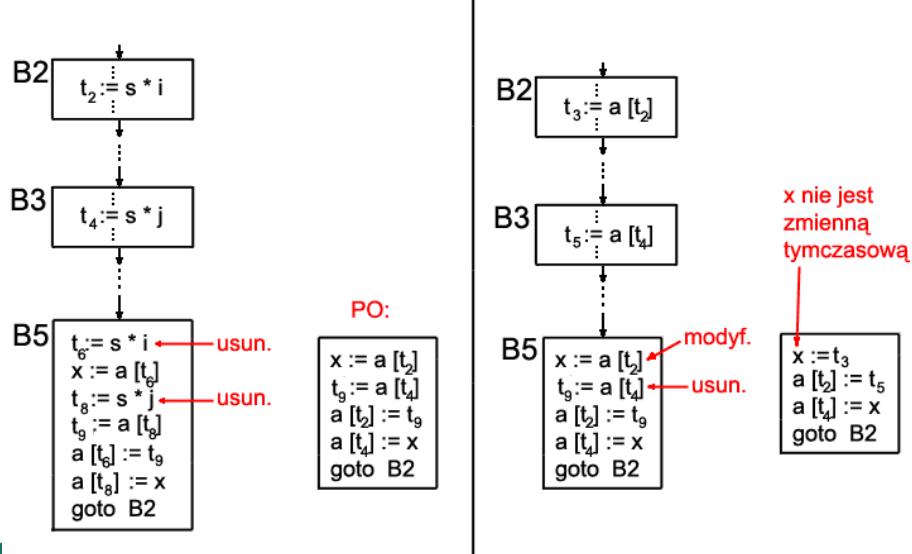
## Przykład: tłumaczenie

|                             |                            |                     |    |
|-----------------------------|----------------------------|---------------------|----|
| (1) $i := m - 1$            | B1                         | (14) $t_6 := s * i$ | B5 |
| (2) $j := n$                |                            | (15) $x := a[t_6]$  |    |
| (3) $t_1 := s * n$          |                            | (16) $t_7 := s * i$ |    |
| (4) $v := a[t_1]$           |                            | (17) $t_8 := s * j$ |    |
| (5) $i := i + 1$            | (18) $t_9 := a[t_8]$       |                     |    |
| (6) $t_2 := s * i$          | (19) $a[t_7] := t_9$       |                     |    |
| (7) $t_3 := a[t_2]$         | (20) $t_{10} := s * j$     |                     |    |
| (8) if $t_3 < v$ goto(5)    | (21) $a[t_{10}] := x$      |                     |    |
| (9) $j := j - 1$            | (22) goto(5)               |                     |    |
| (10) $t_4 := s * j$         | B6                         |                     |    |
| (11) $t_5 := a[t_4]$        | (23) $t_{11} := s * i$     |                     |    |
| (12) if $t_5 > v$ goto(9)   | (24) $x := a[t_{11}]$      |                     |    |
| (13) if $i \geq j$ goto(23) | (25) $t_{12} := s * i$     |                     |    |
|                             | (26) $t_{13} := s * n$     |                     |    |
|                             | (27) $t_{14} := a[t_{13}]$ |                     |    |
|                             | (28) $a[t_{12}] := t_{14}$ |                     |    |
|                             | (29) $t_{15} := s * n$     |                     |    |
|                             | (30) $a[t_{15}] := x$      |                     |    |





## Globalna eliminacja wspólnych podwyrażeń





## Propagacja kopiowania

Można zastąpić:

$b := a; c := b$

Przez:

$b := a; c := a$

W ten sposób „przerywamy” łańcuch propagacji kopiowania

PRZED:

B5:

$x := t_3$   
 $a[t_2] := t_5$   
 $a[t_4] := x$   
goto B2

PO:

B5:

$x := t_3$   
 $a[t_2] := t_5$   
 $a[t_4] := \textcolor{red}{t}_3$   
goto B2

Pozornie nie przynosi to efektów, ale...



## Eliminacja martwego kodu

Eliminuje się te instrukcje trójadresowe, które nadają wartość takim zmiennym, które nie będą potem używane.

PRZED:

B5:

$x := t_3$  ← wyeliminowane  
 $a[t_2] := t_5$   
 $a[t_4] := t_3$   
goto B2

PO:

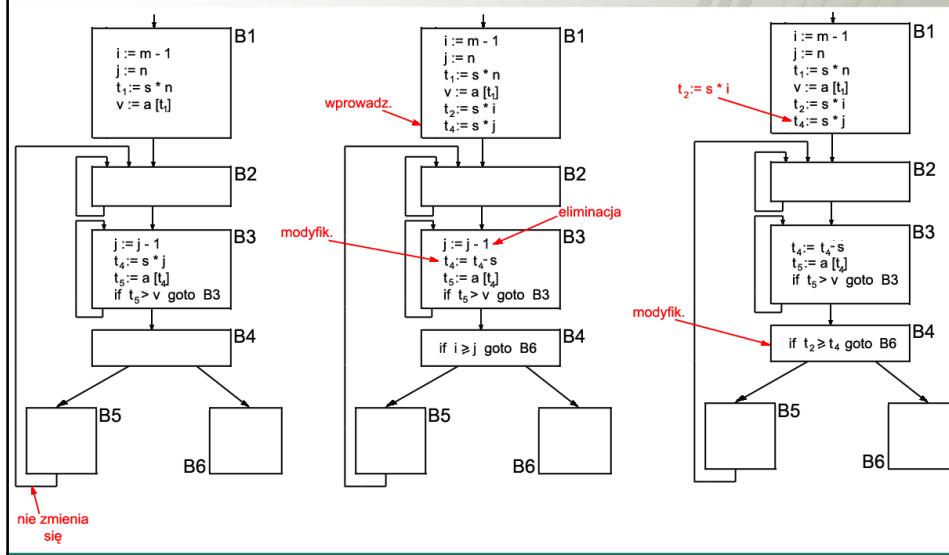
B5:

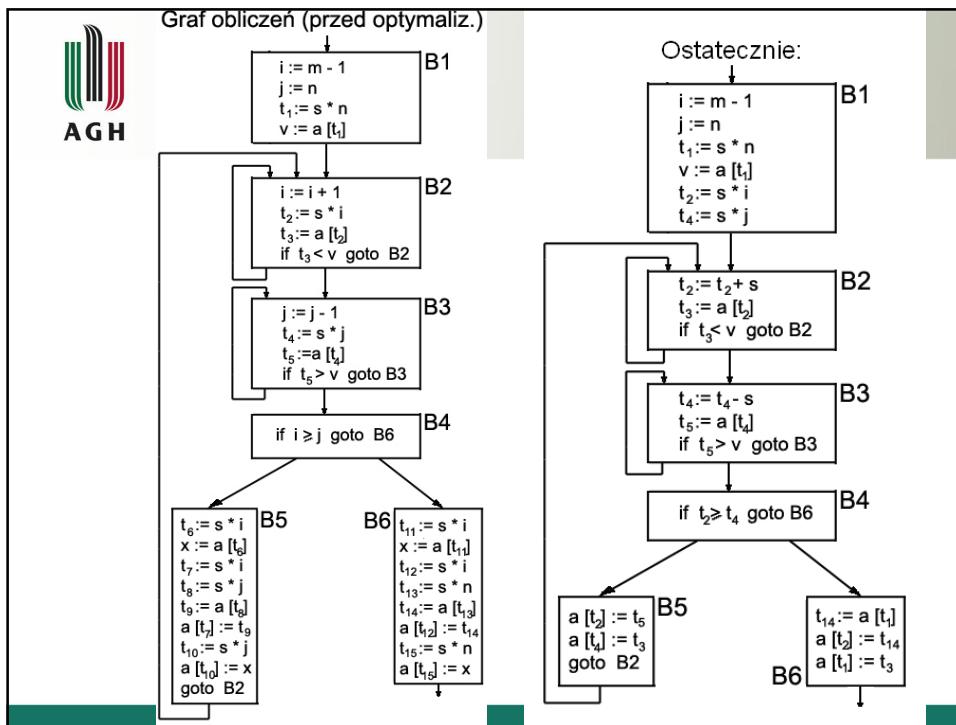
$a[t_2] := t_5$   
 $a[t_4] := t_3$   
goto B2

## Zmienne indukcyjne w pętlach

- Zmienne indukcyjne: zmienne ściśle ze sobą związane w pętli; zmiana jednej z nich powoduje synchroniczną zmianę drugiej.
- Redukcja mocy: zastąpienie operacji dłużej wykonującej się operacją szybszą; przykład: zamiana mnożenia na dodawanie, zamiana mnożenia całkowitoliczbowego bez znaku na przesunięcie bitowe, zamiana potęgowania na mnożenie.
- Wykrycie zmiennych indukcyjnych pozwala na redukcję mocy kodu.
- Istnieje także możliwość eliminacji niektórych zmiennych indukcyjnych.

## Zmienne indukcyjne - przykład





## Przemieszczanie kodu niezmennicznego

### Przykład :

```

while (i <= limit - 2) { ...
 /* instrukcje nie zmieniające
 wartości zmiennej "limit" */
}

```

może być przekształcone do postaci równoważnej:

```

t = limit - 2;
while (i <= t) { ...
 /* instrukcje nie zmieniające
 wartości zmiennych "limit" i "t" */
}

```



## Optymalizacja „przez szparkę”

Peephole optimization – dosłownie: "optymalizacja przez judasza"

- Eliminacja zbędnych skoków

PRZED:

```
if a < b goto L3
L1: if c < d goto L2
 goto L4
L2: if e < f goto L3
 goto L4
```

można wyeliminować

PO:

```
if a < b goto L3
L1: if c < d goto L2
 goto L4
L2: if e < f goto L3
 goto L4
```



## Eliminacja zbędnych skoków

PRZED:

```
if a < b goto L3
L1: if c < d goto L2
 goto L4
L2: if e < f goto L3
 goto L4
```

można zmodyfikować  
„skok przez skok”  
zmieniając warunek na przeciwny

PO:

```
if a < b goto L3
L1: if c >= d goto L4
 goto L2
L2: if e < f goto L3
 goto L4
```

można wyeliminować

```
if a < b goto L3
L1: if c >= d goto L4
L2: if e < f goto L3
 goto L4
```

optymalizacja tego fragmentu  
uzależniona jest od położenia  
etykiet: L3 i L4



## Optymalizacja przebiegu sterowania

```
 goto L1 } goto L2
 ...
L1: goto L2 } → {

if a < b goto L1 } if a < b goto L2
...
L1: goto L2 } → {
L1: goto L2
```

Optymalizacja tego typu nie zmniejsza liczby instrukcji ale zmniejsza liczbę realizowanych skoków w czasie wykonania programu.



## Reorganizacja kodu

```
 goto L1 } if a < b goto L2
 ...
L1: if a < b goto L2 } → { goto L3
L3 : ... } {
L3: ...
```

Zakładamy, że do L1 jest tylko jeden skok (liczba skoków do każdej etykiety może być pamiętana w tablicy symboli).

Optymalizacja tego typu nie zmniejsza liczby instrukcji ale zmniejsza liczbę realizowanych skoków w czasie wykonania programu.



## Eliminacja nieosiągalnego kodu

Przykład:

```
debug := 0
if debug = 1 goto L1
goto L2
L1: /* print debug information */
L2:
```

eliminacja "skoku przez skok"

```
if debug ≠ 1 goto L2
..... /* print debug information */
L2:
```

uwzględnianie tego, że debug = 0

```
if 0 ≠ 1 goto L2
...../* print debug information */
L2:
```

0 ≠ 1 → zawsze prawdziwe  
kod nieosiągalny można usunąć

```
goto L2
L2:
```

**Eliminacja zbędnego skoku powoduje  
całkowite wyeliminowanie rozważanego  
fragmentu kodu**



## Eliminacja tożsamości algebraicznych

Przykład:

$$\left. \begin{array}{l} x := x + 0 \\ x := x * 1 \end{array} \right\}$$

**mogą być wyeliminowane  
gdyż faktycznie nie zmieniają  
wartości zmiennej x**



## Redukcja mocy kodu

Przykład:

$y := 2 * x$

**może być zastąpione przez**

$y := x + x$

**gdyż mnożenie trwa dłużej niż dodawanie**

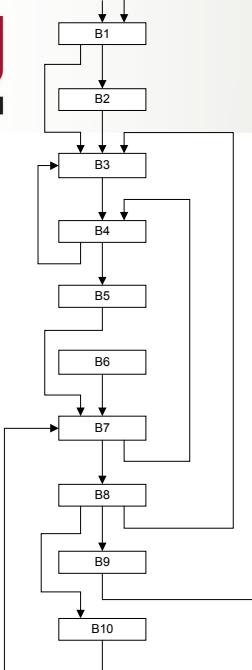


AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

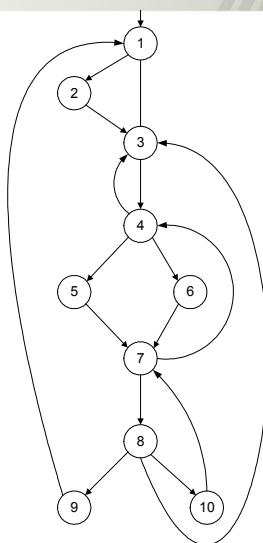
## Grafy przepływu

### Teoria kompilacji

Dr inż. Janusz Majewski  
Katedra Informatyki



### Ilustrowanie grafów przepływu





## Pętle w grafach przepływu

**Pętlą** nazywamy taki zbiór w grafie przepływu, który charakteryzuje się tym, że:

- 1) wszystkie węzły w zbiorze są silnie powiązane, to znaczy, że z każdego węzła w pętli do każdego innego prowadzi droga o długości jeden lub więcej, całkowicie zawarta w pętli
- 2) zbiór węzłów posiada dokładnie jeden węzeł wejściowy, czyli taki węzeł, że wszystkie drogi z zewnątrz pętli prowadzące do węzłów pętli muszą najpierw przejść przez węzeł wejściowy

Pętla nie zawierająca innych pętli nazywana jest pętlą wewnętrzną.



## Relacja dominacji

Mówimy, że węzeł  $d$  w grafie przepływu **dominuje** węzeł  $n$ , co zapisujemy  $d \text{ dom } n$ , jeśli każda droga prowadząca z węzła początkowego do  $n$  przechodzi przez  $d$ .

Zgodnie z tą definicją:

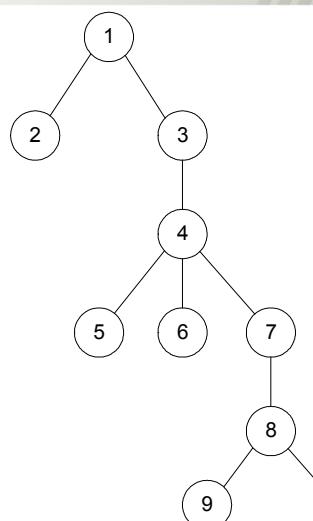
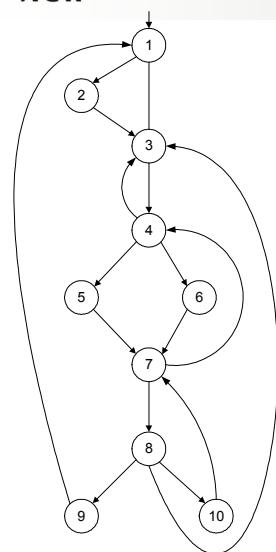
- Każdy węzeł dominuje siebie samego
- Węzeł wejściowy pętli dominuje wszystkie węzły w pętli.

## Relacja dominacji

Relację dominacji ilustrujemy często w postaci tzw. **drzewa dominacji**, w którym korzeniem jest węzeł początkowy grafu przepływu, zaś każdy węzeł dominuje tylko swoje potomstwo w drzewie dominacji.

Istnienie drzewa dominacji wynika z następującej własności relacji dominacji: każdy węzeł  $n$  (z wyjątkiem węzła początkowego grafu) posiada dokładnie jeden węzeł bezpośrednio go dominujący, jest to ostatni węzeł dominujący  $n$  na każdej z dróg prowadzących od węzła początkowego do  $n$ .

## Drzewo dominacji



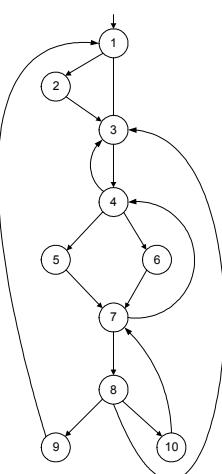
## Relacja dominacji a pętle

Wykorzystanie relacji dominacji do wyznaczania pętli:

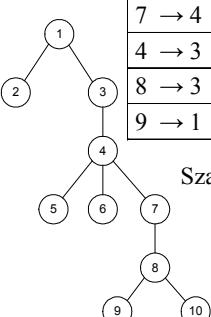
- 1) Pętla musi mieć jedyny punkt wejścia (węzeł wejściowy) zwany nagłówkiem. Ten punkt wejścia dominuje wszystkie węzły w pętli.
- 2) Musi istnieć co najmniej jeden sposób iteracji pętli, tzn. co najmniej jedna droga prowadząca z węzła należącego do pętli z powrotem do nagłówka.

**Krawędzią wsteczną** nazywamy taką krawędź grafu przepływu prowadzącą z węzła  $a$  do  $b$  ( $a \rightarrow b$ ), dla której  $b$  dominuje  $a$  ( $b \text{ dom } a$ ).

## Przykład – pętle

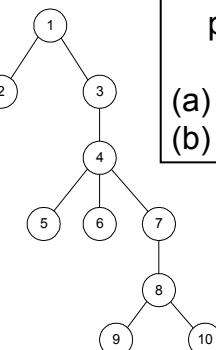
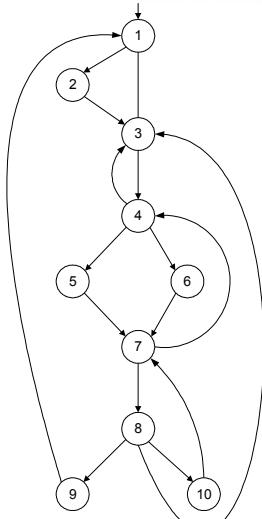


| Krawędź wsteczna:  | Pętla:                              | Komentarz:              |
|--------------------|-------------------------------------|-------------------------|
| $10 \rightarrow 7$ | $\{7, 8, 10\}$                      | Jedyna pętla wewnętrzna |
| $7 \rightarrow 4$  | $\{4, 5, 6, 7, 8, 10\}$             |                         |
| $4 \rightarrow 3$  | $\{3, 4, 5, 6, 7, 8, 10\}$          | } Ta sama pętla         |
| $8 \rightarrow 3$  | $\{3, 4, 5, 6, 7, 8, 10\}$          |                         |
| $9 \rightarrow 1$  | $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ | Cały program            |



Szarym kolorem zostały zaznaczone nagłówki pętli.

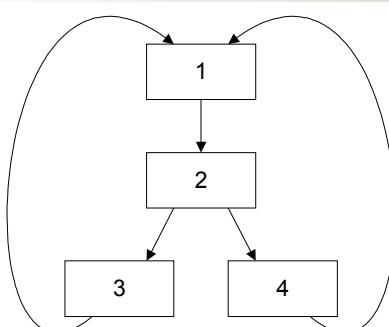
## Przykład – pętle



Dlaczego (wbrew naszej intuicji) pętlami NIE są:

- (a)  $\{3, 4\}$
- (b)  $\{3, 4, 5, 6, 7\}$

## Przykład – pętle



Ile tutaj jest pętli? Ile jest pętli wewnętrznych?

*/\* Dwie pętle o tym samym nagłówku mogą być albo rozłączne, albo jedna z nich jest całkowicie zawarta w drugiej \*/*



## Relacja dominacji a pętle

Mając krawędź wsteczną  $n \rightarrow d$  ( $d \text{ dom } n$ ) definiujemy pętlę jako zbiór zawierający węzeł  $d$  (nagłówek), węzeł  $n$  (jeśli  $n \neq d$ ) oraz te wszystkie węzły, z których (bezpośrednio lub pośrednio) można dojść do  $n$  nie przechodząc przez  $d$ .



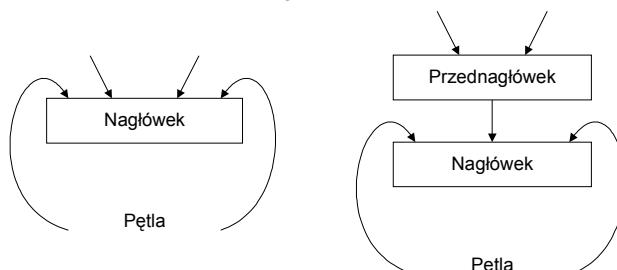
## Algorytm wyznaczania pętli opartej na krawędzi wstecznej $n \rightarrow d$

Wejście: Graf przepływu  $G$  i krawędź wsteczna  $n \rightarrow d$  ( $d \text{ dom } n$ )  
Wyjście: Zbiór loop zawierający wszystkie węzły pętli opartej na krawędzi wstecznej  $n \rightarrow d$

```
procedure insert(m);
if m is not in loop then
 begin
 loop:= loop \cup {m}
 push m onto stack;
 end;
end;
/* main program follows */
stack:=empty;
loop:={d};
insert(n);
while stack is not empty do
 begin
 pop m, the first element of stack, off stack;
 for each predecessor p of m do
 insert(p);
 end;
```

## Przednagłówek pętli

Tworzenie przednagłówka pętli jest praktykowane wówczas gdy transformacje wymagają przeniesienia instrukcji trójadresowych „przed nagłówkiem”. Można utworzyć wówczas nowy blok zwany przednagłówkiem, początkowo pusty; późniejsze transformacje będą mogły tam wstawiać instrukcje przesuwane z wnętrza pętli. Jedynym następcą przednagłówka jest nagłówek; wszystkie krawędzie dochodzące poprzednio do nagłówka z zewnątrz pętli będą teraz dochodziły do przednagłówka, krawędzie wsteczne budujące pętle będą bez zmian kończone w nagłówku.



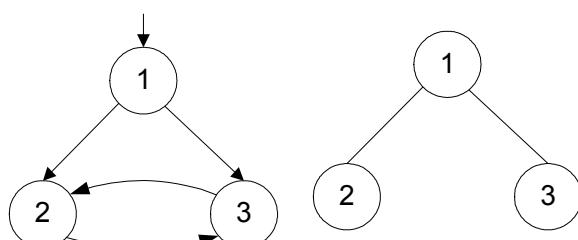
## Redukowalne grafy przepływu

Redukowalne grafy przepływu uzyskujemy w wyniku tłumaczenia programów, do pisania których wykorzystywano wyłącznie instrukcje strukturalne jak: instrukcja złożona, *if – then – else*, *while – do*, *do – while*, *for*, *switch – case*, *continue*, *break*, itd. Zastosowanie *goto* może spowodować uzyskanie nieredukowalnego grafu przepływu.

Graf przepływu nazywamy **redukowalnym** jeżeli można podzielić jego krawędzie na dwie rozłączne grupy: krawędzie skierowane ku przodowi (forward edges) i krawędzie wsteczne (back edges). Krawędzie wsteczne zdefiniowano poprzednio ( $n \rightarrow d$ ,  $d \text{ dom } n$ ), pozostałe są krawędziami skierowanymi ku przodowi. Aby graf przepływu był redukowalny krawędzie skierowane ku przodowi muszą tworzyć graf acykliczny, w którym każdy węzeł jest osiągalny z węzła początkowego.

Przykład grafu nieredukowalnego (obok znajduje się jego drzewo dominacji):

```
/* brak krawędzi
 wstecznych */
/* brak pętli */
/* cały graf nie jest
 acykliczny */
```





AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

## Analiza przepływu

### Teoria kompilacji

Dr inż. Janusz Majewski  
Katedra Informatyki



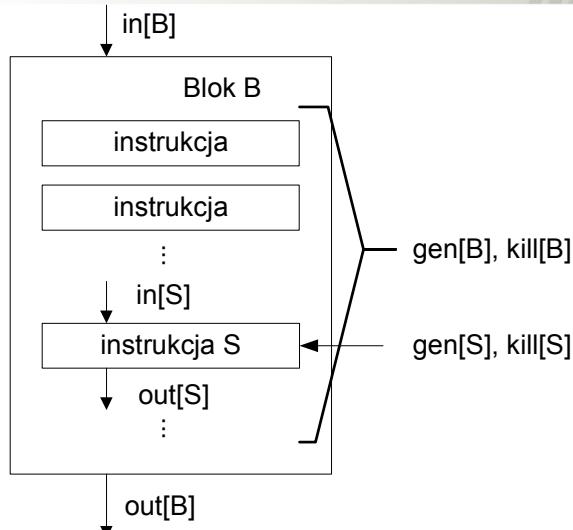
## Równania przepływu

Globalna analiza przepływu danych zajmuje się zbieraniem informacji o pewnych aspektach działania programu i kojarzenia ich ze stosownymi miejscami (punktami) w grafie przepływu. Informacje są uzyskiwane w wyniku konstruowania i rozwiązywania tzw. równań przepływu danych. Typowe równanie ma postać:

$$\text{out}[S] = \text{get}[S] \cup (\text{in}[S] — \text{kill}[S])$$

co może być przeczytane jako: „informacja na końcu instrukcji  $S$  jest albo generowana wewnątrz tej instrukcji, albo jest informacją „przychodzącą” do instrukcji  $S$  i nie zabita (zniszczona) w trakcie wykonywania instrukcji  $S$ ”. Informacja dotyczy instrukcji lub bloku bazowego.

## *in, out, gen i kill*



## Problem zasięgu definicji

Definicją zmiennej  $x$  będziemy nazywać taką instrukcją, która nadaje (lub może to zrobić) wartość zmiennej  $x$ .

Mamy definicje jednoznaczne:

- instrukcje przypisania (podstawienia)
- instrukcja czytania „read”

oraz definicje niejednoznaczne:

- wywołanie procedury ze zmienną  $x$  jako parametrem (nie przekazywanym przez wartość)
- podstawienie przez wskazanie, które może odwoływać się do zmiennej  $x$ , np.:  $*q=y$ , jeśli jest możliwość, żeby  $q$  wskazywało na  $x$ .

## Problem zasięgu definicji

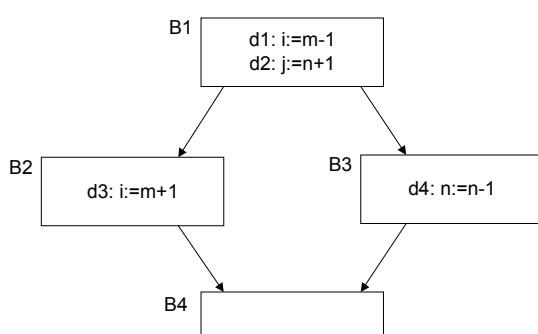
Definicja  $d$  zmiennej  $a$  osiąga punkt  $p$ , jeżeli istnieje ścieżka od punktu następującego po  $d$  do  $p$ , na której nie ma redefinicji  $a$ .

Mówimy, że definicja  $d_2$  zmiennej  $a$  zabija definicję  $d_1$  zmiennej  $a$ , jeżeli  $d_1$  osiąga punkt poprzedzający  $d_2$ .

## Problem zasięgu definicji

Definicja  $d$  zmiennej  $a$  osiąga punkt  $p$ , jeżeli istnieje ścieżka od punktu następującego po  $d$  do  $p$ , na której nie ma redefinicji  $a$ . Mówimy, że definicja  $d_2$  zmiennej  $a$  zabija definicję  $d_1$  zmiennej  $a$ , jeżeli  $d_1$  osiąga punkt poprzedzający  $d_2$ .

Przykład:



Obie definicje  $d_1$  i  $d_2$  z bloku  $B_1$  osiągają blok  $B_4$  chociaż na jednej ze ścieżek definicja  $d_1$  zmiennej  $i$  zostaje zabita poprzez definicję  $d_3$  z bloku  $B_2$ . Istnieje jednak ścieżka z  $B_1$  do  $B_4$  przez  $B_3$  bez redefinicji zmiennej  $i$ .



## Iteracyjny algorytm dla zasięgu definicji

Wejście: Graf przepływu  $G$ , dla każdego bloku  $B$  obliczono zbiory  $gen[B]$  i  $kill[B]$

Wyjście: Zbiory  $in[B]$  i  $out[B]$  obliczone dla każdego bloku  $B$

```

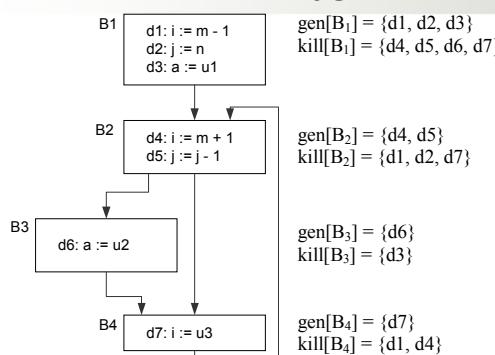
for each block B do out[B] := gen[B];
change := true;
while change do
 begin
 change := false;
 for each block B do
 begin
 in[B] := $\bigcup_{P \text{ a predecessor of } B} out[P];$
 old_out := out[B];
 out[B] := gen[B] \cup (in[B] - kill[B]);
 if out[B] \neq old_out then change := true;
 end
 end

```

Algorytm pracuje dopóki występują zmiany. Górną granicą iteracji jest liczba węzłów



## Iteracyjny algorytm dla zasięgu definicji – przykład



| Blok | Początkowo |          | Po 1-szym przejściu |          | Po 2-gim przejściu |  |
|------|------------|----------|---------------------|----------|--------------------|--|
|      | out[B]     | in[B]    | out[B]              | in[B]    | out[B]             |  |
| B1   | 111 0000   | 000 0000 | 111 0000            | 000 0000 | 111 0000           |  |
| B2   | 000 1100   | 111 0001 | 001 1100            | 111 0111 | 001 1110           |  |
| B3   | 000 0010   | 001 1100 | 000 1110            | 001 1110 | 000 1110           |  |
| B4   | 000 0001   | 001 1110 | 001 0111            | 001 1110 | 001 0111           |  |



## Problem dostępności wyrażeń

Wyrażenie  $s: x \underline{op} y$  jest dostępne w punkcie  $p$ , gdy na każdej ścieżce z punktu początkowego do punktu  $p$  jest ono wyliczane i od ostatniego obliczenia do osiągnięcia punktu  $p$  nie ma definicji ani  $x$ , ani  $y$ .

Mówimy że blok zabija wyrażenie  $x \underline{op} y$  jeżeli redefiniuje  $x$  lub  $y$  i nie oblicza dalej  $x \underline{op} y$ .

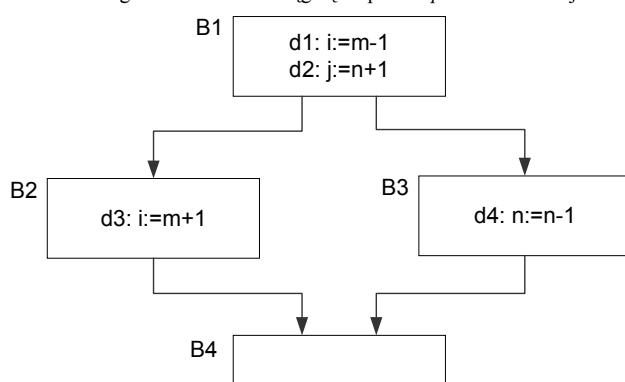
Blok generuje wyrażenie  $x \underline{op} y$  jeżeli oblicza jego wartość i później nie redefiniuje ani  $x$  ani  $y$ .

**Mimo że w problemie dostępności wyrażenia inaczej określamy generowanie i zabijanie to ogólne prawa analizy przepływu danych oczywiście są tutaj takie same.**



## Problem dostępności wyrażeń – przykład

Wyrażenie  $s: x \underline{op} y$  jest dostępne w punkcie  $p$ , gdy na każdej ścieżce z punktu początkowego do punktu  $p$  jest ono wyliczane i od ostatniego obliczenia do osiągnięcia punktu  $p$  nie ma definicji ani  $x$ , ani  $y$ .



W bloku  $B_4$  dostępne mamy wyrażenia  $m-1$  ( $d_1$ ) oraz  $m+1$  ( $d_3$ ). Dostępność wyrażeń  $n+1$  ( $d_2$ ) i  $n-1$  ( $d_4$ ) zostaje zabita przez definicję  $d_4$  zmiennej  $n$ .

## Iteracyjny algorytm dla dostępności wyrażeń

Wejście: Graf przepływu  $G$ , dla każdego bloku  $B$  wyznaczono zbiory  $e\_gen[B]$  i  $e\_kill[B]$

Wyjście: Zbiory  $e\_in[B]$  i  $e\_out[B]$  dla każdego bloku  $B$

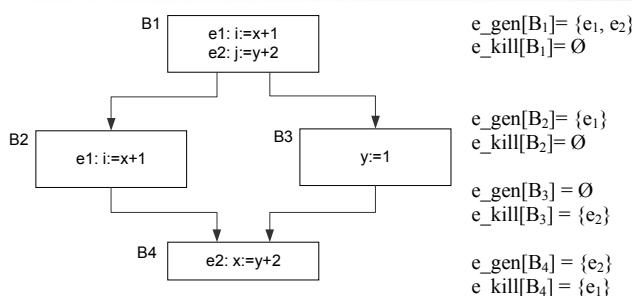
Uwaga:  $U$  – zbiór wszystkich wyrażeń typu  $x \oplus y$  we wszystkich instrukcjach w programie;  
 $B_1$  – blok początkowy

```

e_in[B1] := ∅
e_out[B1] := e_gen[B1]; /* in and out never change for B1 */
for B ≠ B1 do
 e_out[B] := U - e_kill[B];
change := true;
while change do
 begin
 change:=false;
 for B ≠ B1 do
 begin /* por. def.: na każdej ścieżce z B1 do
 danego B jest wyliczane i ... */
 e_in[B] := ⋂
 P a predecessor of B
 oldout:=e_out[B]
 e_out[B]:=e_gen[B] ∪ (e_in[B]- e_kill[B])
 if e_out[B] ≠ oldout then change:=true;
 end;
 end;

```

## Iteracyjny algorytm dla dostępności wyrażeń - przykład



$$U=\{e_1, e_2\}$$

| Blok  | Początkowo |             | Po pierwszym przejściu |             |
|-------|------------|-------------|------------------------|-------------|
|       | $e\_in[B]$ | $e\_out[B]$ | $e\_in[B]$             | $e\_out[B]$ |
| $B_1$ | 00         | 11          | 00                     | 11          |
| $B_2$ | 00         | 11          | 11                     | 11          |
| $B_3$ | 00         | 10          | 11                     | 10          |
| $B_4$ | 00         | 01          | 10                     | 01          |



## Problem zasięgu instrukcji kopирования

Instrukcja kopowania  $s: x:=y$  osiąga punkt  $p$ , gdy na każdej ścieżce z punktu początkowego do punktu  $p$  instrukcja ta się pojawia i po ostatnim pojawieniu się  $s$  nie ma późniejszej redefinicji  $y$ .

Mówimy, że instrukcja kopowania  $s: x:=y$  jest generowana w bloku  $B$  jeżeli  $s$  pojawia się w  $B$  i później wewnątrz bloku  $B$  nie ma redefinicji zmiennej  $y$ . Mówimy, że  $s: x:=y$  jest zabijana w bloku  $B$ , jeśli w bloku  $B$  jest definicja zmiennej  $x$  lub  $y$  oraz  $s$  nie występuje w  $B$ .



## Iteracyjny algorytm dla zasięgu instrukcji kopowania

Wejście: Graf przepływu  $G$ , w którym wyznaczono zbiory  $c\_gen[B]$  i  $c\_kill[B]$

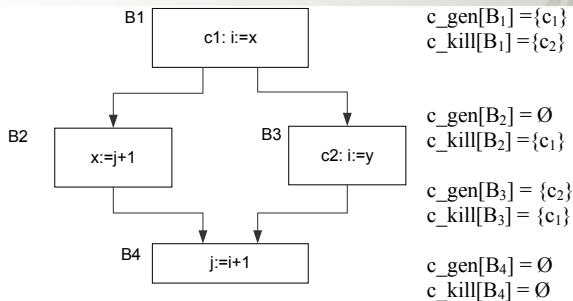
Wyjście: Zbiory  $c\_in[B]$  i  $c\_out[B]$  dla każdego bloku  $B$

Uwaga:  $U$  – zbiór wszystkich instrukcji kopowania  $x:=y$  w całym programie.

$B_1$  – blok początkowy

```
c_in[B1] := ∅
c_out[B1] := c_gen[B1];
for B ≠ B1 do
 c_out[B] := U - c_kill[B];
change := true;
while change do
 begin
 change := false;
 for B ≠ B1 do
 begin
 c_in[B] := ⋂
 P a predecessor of B
 c_out[B];
 oldout := c_out[B]
 c_out[B] := c_gen[B] ∪ (c_in[B] - c_kill[B])
 if c_out[B] ≠ oldout then change := true;
 end;
 end;
```

## Iteracyjny algorytm dla zasięgu instrukcji kopowania – przykład



| U={c <sub>1</sub> , c <sub>2</sub> } |  | Początkowo          |                      | Po pierwszym przejściu |                      |
|--------------------------------------|--|---------------------|----------------------|------------------------|----------------------|
| Blok                                 |  | c <sub>in</sub> [B] | c <sub>out</sub> [B] | c <sub>in</sub> [B]    | c <sub>out</sub> [B] |
| B <sub>1</sub>                       |  | 00                  | 10                   | 00                     | 10                   |
| B <sub>2</sub>                       |  | 00                  | 01                   | 10                     | 00                   |
| B <sub>3</sub>                       |  | 00                  | 01                   | 10                     | 01                   |
| B <sub>4</sub>                       |  | 00                  | 11                   | 00                     | 00                   |

## Problem życia zmiennych

Mówimy, że zmienna  $x$  w punkcie  $p$  jest żywa, jeżeli jej wartość jest używana na jakiejś ścieżce rozpoczynającej się w punkcie  $p$ . W przeciwnym przypadku mówimy, że zmienna jest martwa.

Będziemy określać  $l\_in[B]$  jako zbiór zmiennych żywych w punkcie rozpoczętym blokiem  $B$  (usytuowanym przed pierwszą instrukcją bloku  $B$ ). Podobnie określamy  $l\_out[B]$  jako zbiór zmiennych żywych w punkcie kończącym blok  $B$  (za ostatnią instrukcją bloku  $B$ ).

Niech  $l\_def[B]$  będzie zbiorem zmiennych, którym w bloku  $B$  przypisywane są wartości wcześniej, niż zmienne te są ewentualnie używane w  $B$ . Niech dalej  $l\_use[B]$  będzie zbiorem takich zmiennych za blokiem  $B$ , które są używane w  $B$  wcześniej niż następuje ewentualne przypisanie wartości tym zmiennym w bloku  $B$ . Zbiór  $def[]$  jest odpowiednikiem  $kill[]$ , zaś  $use[]$  jest odpowiednikiem  $gen[]$ .

## Iteracyjny algorytm dla życia zmiennych

Wejście: Graf przepływu  $G$ , dla każdego bloku wyznaczono zbiory  $l\_use[B]$  i  $l\_def[B]$

Wyjście: Zbiory  $l\_in[B]$  i  $l\_out[B]$  dla każdego bloku  $B$

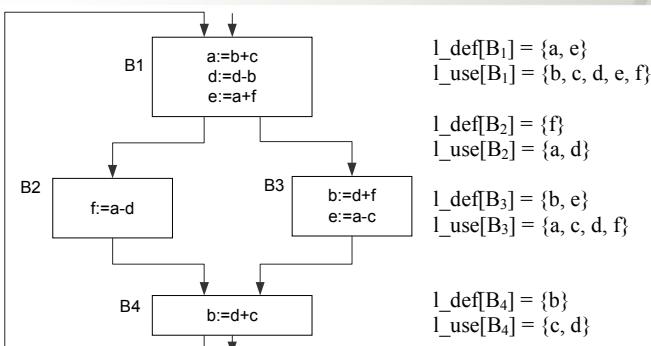
Uwaga: W odróżnieniu od poprzednich algorytm oblicza  $in[]$  w zależności od  $out[]$   
oraz analizuje graf przepływu "wstecz"

```

for each block B do l_in[B] := ∅
change := true;
while change do
 begin
 change := false;
 for each block B do
 begin
 l_out[B] := $\bigcup_{P \text{ a predecessor of } B} l_{in}[S];$
 oldin := l_in[B]
 l_in[B] := c_use[B] $\cup (l_{out}[B] - l_{def}[B])$
 if l_in[B] <> oldin then change := true;
 end;
 end;

```

## Iteracyjny algorytm dla życia zmiennych - przykład



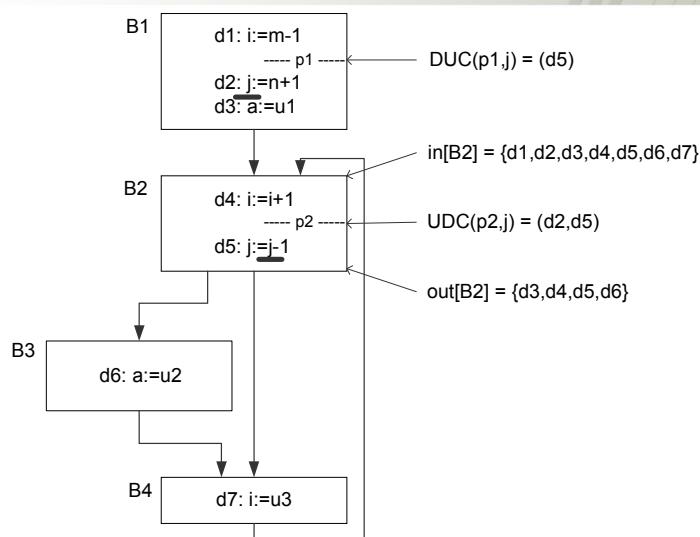
| Blok  | $l_{in^1}[B]$ | $l_{out^2}[B]$ | $l_{in^2}[B]$ | $l_{out^3}[B]$ | $l_{in^3}[B]$ |
|-------|---------------|----------------|---------------|----------------|---------------|
| $B_1$ | b, c, d, f    | a, c, d, f     | b, c, d, f    | a, c, d, f     | b, c, d, f    |
| $B_2$ | a, d          | c, d           | a, c, d       | c, d, f        | a, c, d       |
| $B_3$ | a, c, d, f    | c, d           | a, c, d, f    | c, d, f        | a, c, d, f    |
| $B_4$ | c, d          | b, c, d, f     | c, d, f       | b, c, d, f     | c, d, f       |

## Listy Definition-Use Chains i Use-Definition Chains

Lista  $DUC(p,x)$  – łańcuch użyc dla definicji (Definition-Use Chains) to obliczana dla punktu  $p$ , poprzedzającego definicję  $d$  zmiennej  $x$ , lista tych wszystkich użyc tej zmiennej, które są osiągane przez definicję  $d$ . Zatem dla każdej definicji mamy zebraną listę instrukcji, które mogą ją potencjalnie wykorzystać

Lista  $UDC(p,x)$  – łańcuch definicji dla użycia (Use-Definition Chains) to obliczana dla punktu  $i$ , poprzedzającego użycie  $s$  zmiennej  $x$ , lista tych wszystkich definicji  $i$ , które osiągają punkt  $p$ . Lista  $UDC$  jest więc „odwrotnością” listy  $DUC$  – dla każdego użycia mamy zebrane informacje o definicjach, które mogą osiągać to użycie.

## Listy Definition-Use Chains i Use-Definition Chains - przykład





AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

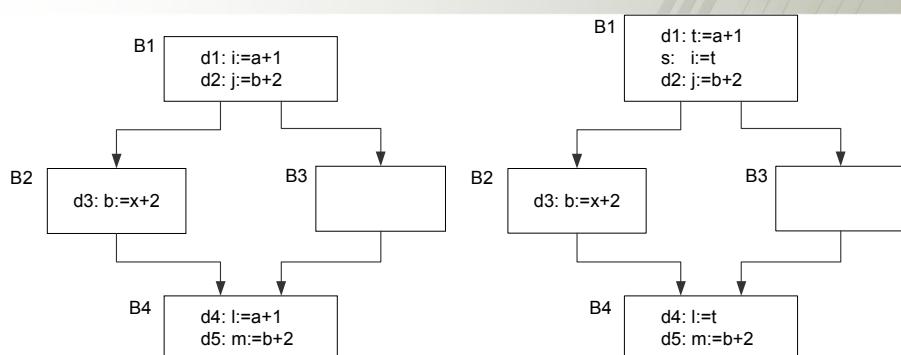
# Transformacje optymalizujące

## Teoria kompilacji

Dr inż. Janusz Majewski  
Katedra Informatyki



# Eliminacja globalnych podwyrażeń



Wyrażenie  $a+1$  w definicji  $d_1$  osiąga blok  $B_4$ , zatem możemy wyeliminować powtórne obliczanie tego wyrażenia w instrukcji  $d_4$  poprzez podstawienie wartości obliczonej w instrukcji  $d_1$  do zmiennej tymczasowej  $t$  i jej wykorzystanie w bloku  $B_4$ . Wyrażenie  $b+2$  nie osiąga bloku  $B_4$ , ponieważ istnieje ścieżka z  $B_1$  poprzez  $B_2$  do  $B_4$ , na której mamy redefinicję  $d_3$  argumentu  $b$  wyrażenia  $b+2$ . Zatem to wyrażenie nie jest dostępne w bloku  $B_4$  i nie możemy przeprowadzić operacji analogicznej do poprzedniej.



## Algorytm usuwania globalnych wspólnych podwyrażeń

Wejście: Graf przepływu i informacje o dostępnych wyrażeniach

Wyjście: Przeredagowany graf przepływu

Metoda: Dla każdej instrukcji postaci  $s: x := y \ op z$ , gdzie  $y \ op z$  jest dostępne na początku bloku zawierającego  $s$  lub generowane w tym bloku przed  $s$ , oraz blok ten nie zawiera redefinicji  $y$  ani  $z$  przed  $s$  wykonaj:

- 1) znajdź takie wyliczenia  $w := y \ op z$ , które osiągają  $s$ , przeglądając graf wstecz,
- 2) utwórz nową zmienną  $t$ ,
- 3) zamień każdą instrukcję znalezioną w punkcie 1) przez parę  
$$t := y \ op z$$
$$w := t$$
- 4) instrukcję  $s: x := y \ op z$  zamień na  $x := t$

Optymalizacja ta może prowadzić do nadmiernego powiększenia ilości zmiennych tymczasowych, dlatego powinna współpracować z algorytmem eliminacji przenoszenia kopiowania.

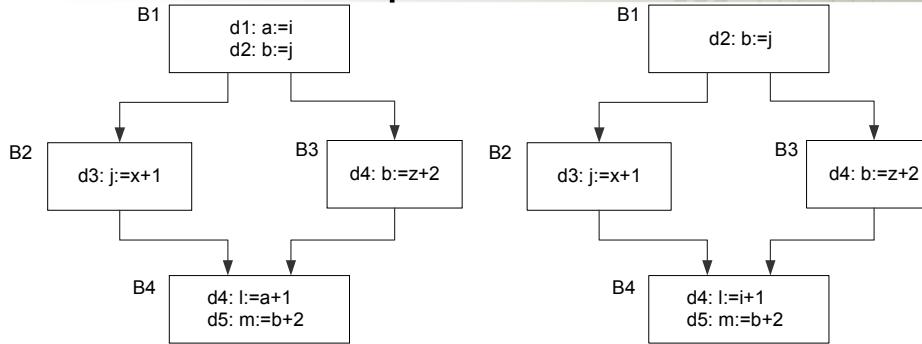


## Eliminacja przenoszenia kopiowania

Algorytm eliminuje kopiowania  $s: x := y$  przez podstawienie  $y$  za  $x$ , w tych użyciach  $u$  zmiennej  $x$ , dla których spełnione są następujące warunki:

- 1)  $s$  musi być jedyną definicją  $x$  osiągającą  $u$  (tzn. lista  $UDC$  w punkcie poprzedzającym  $u$  zawiera tylko  $s$ )
- 2) na każdej ścieżce z  $s: x := y$  do  $u$  nie ma redefinicji  $y$  (sprawdzenie tego warunku uzyskamy przez rozwiązanie problemu zasięgu instrukcji kopiowania)

## Eliminacja przenoszenia kopiowania



Z dwóch instrukcji kopiowania w bloku  $B_1$  grafu lewego można wyeliminować pierwszą ( $d_1$ ) i podstawić za  $a$  wartość zmiennej  $i$  w bloku  $B_4$  ( $d_5$ ). Z drugą ( $d_2$ ) nie można tak postąpić z dwóch powodów:

- 1) na ścieżce z  $B_1$  przez  $B_2$  do  $B_4$  mamy redefinicję zmiennej  $j$  ( $d_3$ )
- 2) definicja  $d_2$  zmiennej  $b$  (której chcielibyśmy wyeliminować) nie jest jedyną definicją tej zmiennej osiągającą blok  $B_4$  (osiąga go również definicja  $d_4$  z bloku  $B_3$ )

## Algorytm eliminacji przenoszenia kopiowania

Wejście: Graf przepływu z informacjami o zasięgu instrukcji kopiowania (zbiory  $c\_in[B]$ , ...) oraz dostępnymi listami  $UDC$  i  $DUC$ .

Wyjście: Przeredagowany graf przepływu

Metoda: Dla każdej instrukcji  $s: x:=y$  wykonaj:

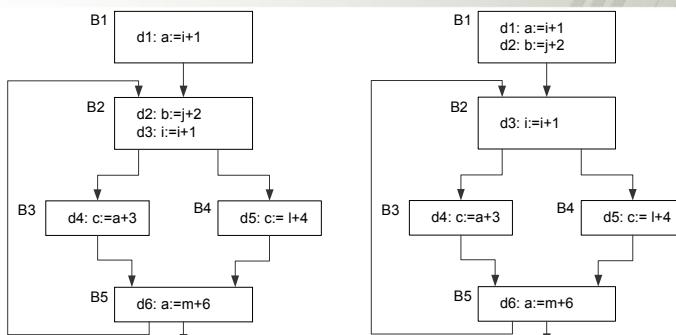
- 1) określ, które użycia zmiennej  $x$  (nazwijmy je  $u$ ) są osiągalne przez definicję  $s$  (lista  $DUC$ ),
- 2) określ, czy dla każdego użycia  $u$  zmiennej  $x$  znalezionego w punkcie 1),  $a$  znajdującego się w bloku  $B$ ,  $s$  znajduje się w  $c\_in[B]$  (\*) lub  $s$  znajduje się w bloku  $B$  przed  $u$  (\*\*),
- 3) określ, czy w przypadku (\*) nie ma wewnątrz bloku  $B$  redefinicji zmiennej  $y$  przed  $u$  (zauważ, że jeśli  $s$  jest elementem  $c\_in[B]$ , to  $s$  jest jedyną definicją zmiennej  $x$ , która osiąga blok  $B$ )
- 4) określ, czy w przypadku (\*\*) nie ma pomiędzy  $s$  a  $u$  redefinicji zmiennej  $y$  (zauważ, że jeżeli  $s$  i  $u$  są wewnątrz jednego bloku oraz  $u$  jest w zasięgu  $s$ , to  $s$  jest jedyną definicją zmiennej  $x$ , która osiąga  $u$ )
- 5) jeżeli warunki 3) lub 4) (odpowiednio dla rozważanego przypadku) są spełnione, to usuń  $s$  i zamień wszystkie użycia zmiennej  $x$  z punktu 1) na  $y$ .

## Eliminacja obliczeń niezmienneńczych z pętli

Można przenieść przed pętlę takie instrukcje  $s$ :  $x := y \ op z$ , które obliczają ciągle tę samą wartość dopóki sterowanie pozostaje w pętli i spełniają poza tym następujące warunki:

- 1) blok zawierający  $s$  dominuje nad wszystkimi wyjściami z pętli (czyli nie istnieje ścieżka przez pętlę, która nie przechodziłaby przez ten blok)
- 2)  $s$  jest jedyną definicją  $x$  w tej pętli
- 3) wszystkie użycia  $x$  wewnątrz pętli są osiągalne tylko i wyłącznie przez definicję  $s$

## Eliminacja obliczeń niezmienneńczych z pętli – przykład



W pętli z lewego grafu definicje  $d_2$ ,  $d_5$  i  $d_6$  obliczają w każdym przebiegu pętli tę samą wartość. Do przednagłówka można jednak przesunąć instrukcję  $d_2$ . Definicja  $d_5$  nie spełnia warunku 1) – po przesunięciu tej definicji do przednagłówka zmienna  $c$  przyjmowałaby wartość  $l+4$  za każdym razem, gdy sterowanie dochodzi do pętli, podczas gdy w rzeczywistości tak być nie musi. Nie jest spełniony również warunek 2). Po przeniesieniu  $d_5$  do przednagłówka po przejściu ścieżki  $B_2-B_3-B_3-B_2-B_4-B_5$  otrzymamy w „ulepszonej” w ten sposób wersji zmiennej  $c$  równą  $a+3$  podczas gdy w pierwotnej pętli otrzymamy wartość zmiennej  $c$  równą  $l+4$ . Definicja  $d_6$  nie spełnia warunku 3). Używana w  $B_3$  zmienna  $a$  w pierwszym przebiegu pętli ma wartość  $i+1$ , w każdym następnym  $m+5$ . Przeniesienie  $d_6$  do przednagłówka zmieniłoby wyraźnie sposób działania programu.



## Algorytm wykrywania obliczeń niezmienniczych w pętli

**Wejście:** pętla  $L$ , listy UDC

**Wyjście:** zbiór wyrażeń, które obliczają tę samą wartość dopóki sterowanie pozostaje w pętli

**Metoda:**

- (1) Zamarkuj jako „niezmiennicze” te instrukcje trójadresowe, których wszystkie operandy są bądź stałymi, bądź mają wszystkie swoje definicje poza pętlą  $L$ .
- (2) Powtarzaj krok (3) dopóki występują zmiany.
- (3) Zamarkuj jako „niezmiennicze” wszystkie takie instrukcje, które poprzednio nie były zamarkowane, a których wszystkie operandy:
  - (a) są stałymi, lub
  - (b) mają wszystkie definicje poza pętlą  $L$ , lub
  - (c) mają dokładnie jedną definicję osiągającą rozważaną instrukcję i definicja ta została zamarkowana jako wyrażenie „niezmiennicze” w pętli  $L$ .



## Algorytm eliminacji obliczeń niezmienniczych z pętli

**Wejście:** pętla  $L$ , listy UDC, informacje o relacji dominacji w grafie

**Wyjście:** przeredagowana pętla  $L$  z przednagłówkiem i (być może) pewnymi instrukcjami przeniesionymi do przednagłówka

**Metoda:**

- (1) Wykonaj poprzedni algorytm wykrywania obliczeń niezmienniczych w pętli  $L$ .
- (2) Dla każdej instrukcji  $s$  definiującej  $x$  i zamarkowanej jako „niezmiennicza” sprawdź, czy:
  - (a)  $s$  jest w bloku dominującym wszystkie wyjścia z pętli  $L$
  - (b) w pętli  $L$  nie ma definicji  $x$  innej niż  $s$
  - (c) wszystkie użycia  $x$  w  $L$  są osiągane tylko przez definicję  $s$
- (3) Przesuń (w porządku wyznaczonym przez poprzedni algorytm) do przednagłówka te instrukcje, które spełniają warunki (2a), (2b) i (2c) pod warunkiem, że jeżeli operandy  $s$  były definiowane wewnątrz  $L$ , to definicje te zostały wcześniej przeniesione do przednagłówka.



## Algorytm eliminacji obliczeń niezmienniczych z pętli

**Uwaga:** czasami można osłabić warunek (2a) powyższego algorytmu, formułując go następująco:

(2a) blok zawierający  $s$  albo dominuje wszystkie wyjścia z pętli  $L$ , albo zmienna  $x$  definiowana w  $s$  nie jest używana na zewnątrz pętli  $L$ .

Jest to przykład tzw. „optymalizacji niebezpiecznej”, gdyż czasami przenosi się do przednagłówka wyrażenie, które być może nigdy nie byłoby w pętli wykonane, a po przeniesieniu do przednagłówka będzie wykonane. Na przykład w pętli jest test:

„czy  $y$  jest różne od  $0$ ?” – jeśli tak wykonuje się dzielenie  $\frac{x}{y}$ . Po ewentualnym

wyniesieniu  $\frac{x}{y}$  do przednagłówka dzielenie wykona się zawsze, niezależnie od wartości dzielnika, mogąc spowodować błąd wykonania.



## Eliminacja zmiennych indukowanych i redukcja mocy

Zmienna  $x$  jest nazywana zmienną indukowaną pętli  $L$ , jeżeli każdorazowa zmiana jej wartości jest inkrementacją lub dekrementacją o pewną stałą.

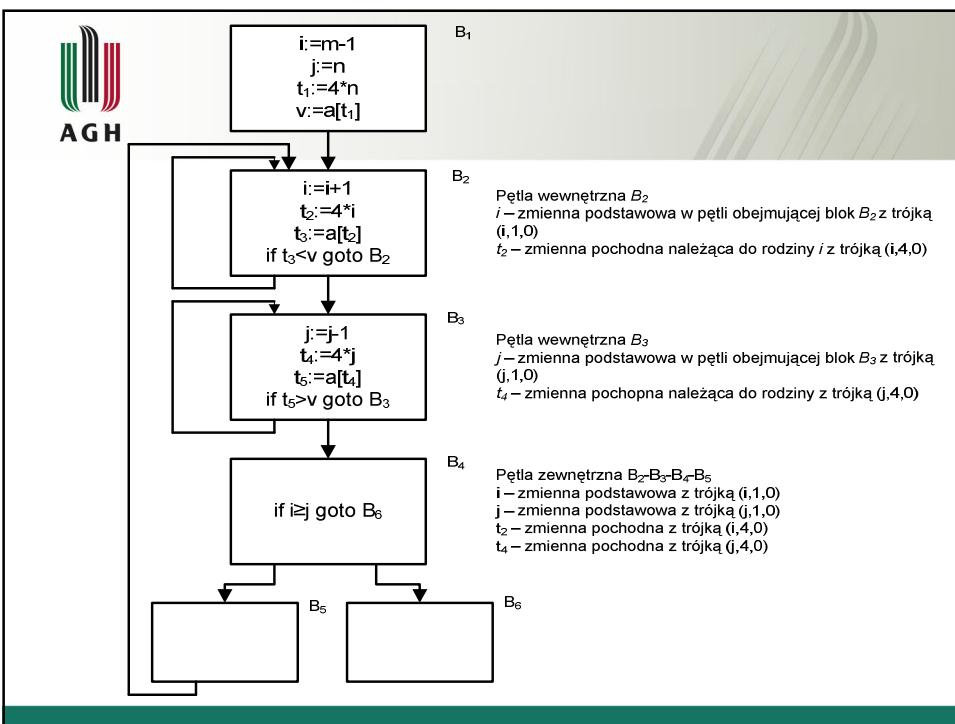
Podstawową zmienną indukowaną jest taka zmienna  $i$ , która jest dokładnie jeden raz definiowana wewnętrz pętli  $L$  i definicja ta ma postać  $i := i + c$  gdzie  $c$  jest pewną stałą (np. zmienna sterująca pętli).

Pochodną zmienną indukowaną jest taka zmienna  $j$ , która jest dokładnie jeden raz definiowana wewnętrz pętli  $L$ , i której wartość jest liniową funkcją pewnej podstawowej zmiennej indukowanej  $i$ . Mówimy wtedy, że zmienna  $j$  należy do rodziny zmiennej  $i$ .

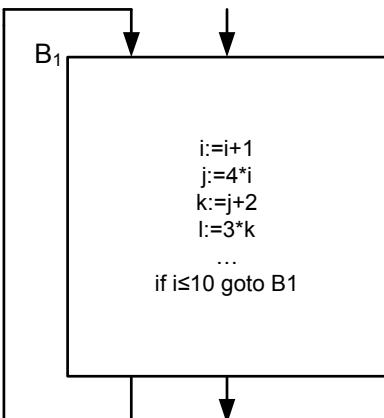
# Algorytm wykrywania zmiennych indukowanych (1)

**Wejście:** pętla  $L$  wraz z informacjami o zasięgu definicji i informacjami o wyrażeniach niezmienniczych w pętli  $L$

**Wyjście:** zbiór zmiennych indukowanych. Z każdą zmienną indukowaną  $j$  skojarzona jest trójką  $(i, c, d)$ , gdzie  $i$  jest podstawową zmienną indukowaną,  $c$  i  $d$  są stałymi, takimi że w miejscu definicji zmiennej  $j$  jej wartość wynosi  $c \cdot i + d$ . Mówimy, że  $j$  należy do rodziny  $i$ . Podstawowa zmienna indukowana  $i$  należy do swojej własnej rodziny.



## Przykład



Pętla obejmująca B1:

$i$  – zmienna podstawowa ( $i, 1, 0$ )  
 $j$  – zmienna pochodna z rodziny  $i$  z trójką ( $i, 4, 0$ )  
 $k$  – zmienna pochodna z rodziny  $i$  z trójką ( $i, 4, 2$ )  
 $[k=j+2=4*i+2]$   
 $l$  – zmienna pochodna z rodziny  $i$  z trójką ( $i, 12, 6$ )  
 $[l=3*k=3*(4*i+2)=12*i+6]$

## Algorytm wykrywania zmiennych indukowanych (2)

### Metoda:

(1) Znajdź wszystkie podstawowe zmienne indukowane  $i$  przeglądając instrukcje pętli  $L$ . Zmienna  $i$  musi mieć dokładnie jedną definicję w pętli  $L$  w postaci  $i := i + a$ , gdzie  $a$  jest stałą. Z każdą podstawową zmienną indukowaną  $i$  skojarz trójkę  $(i, 1, 0)$ .

(2) Poszukaj wszystkich zmiennych  $k$  mających dokładnie jedną definicję w pętli  $L$  w jednej z poniższych postaci:

$$k := j \cdot b, \quad k := b \cdot j, \quad k := \frac{j}{b}, \quad k := j \pm b, \quad k := b \pm j$$

gdzie:  $b$  – stała,  $j$  – zmienna indukowana, podstawowa lub pochodna.

Jeśli  $j$  jest podstawową zmienną indukowaną, wówczas  $k$  jest pochodną zmienną indukowaną i należy do rodziny  $j$ . Trójka dla  $k$  zależy od postaci instrukcji definiującej tę zmienną. Na przykład jeśli  $k$  definiowane jest jako

$k := j \cdot b$ , to trójka dla  $k$  ma postać  $(j, b, 0)$ ; jeżeli  $k := \frac{j}{b}$ , to trójka dla  $k$  ma

postać  $(j, \frac{1}{b}, 0)$ ; jeżeli  $k := j + b$ , wówczas trójka dla  $k$  to  $(j, 1, b)$ .



## Algorytm wykrywania zmiennych indukowanych (3)

Jeśli  $j$  nie jest podstawową zmienną indukowaną, to wtedy  $j$  jest zmienną pochodną i należy do rodziny zmiennej podstawowej  $i$ . Wówczas należy sprawdzić dodatkowe wymagania:

- (a) nie ma być podstawienia dla zmiennej  $i$  pomiędzy jedyną definicją  $j$  w  $L$ , a jedyną definicją  $k$  w  $L$ .
- (b) żadna definicja  $j$  spoza pętli  $L$  nie osiąga definicji  $k$ .

Gdy wymagania te są spełnione,  $k$  jest zmienną pochodną należącą do rodziny  $i$ .

Wtedy tworzymy trójkę dla zmiennej  $k$  na podstawie trójki  $(i, c, d)$  dla zmiennej  $j$  i na podstawie instrukcji definiującej zmienną  $k$ . Na przykład definicja  $k := b \cdot j$  prowadzi do trójki  $(i, b \cdot c, b \cdot d)$  dla zmiennej  $k$ .



## Algorytm redukcji mocy wykorzystujący zmienne indukowane

**Wejście:** pętla  $L$  wraz z informacją o zasięgu definicji oraz informacje o rodzinach zmiennych indukowanych uzyskane w wyniku wykonania poprzedniego algorytmu

**Wyjście:** przeredagowana pętla

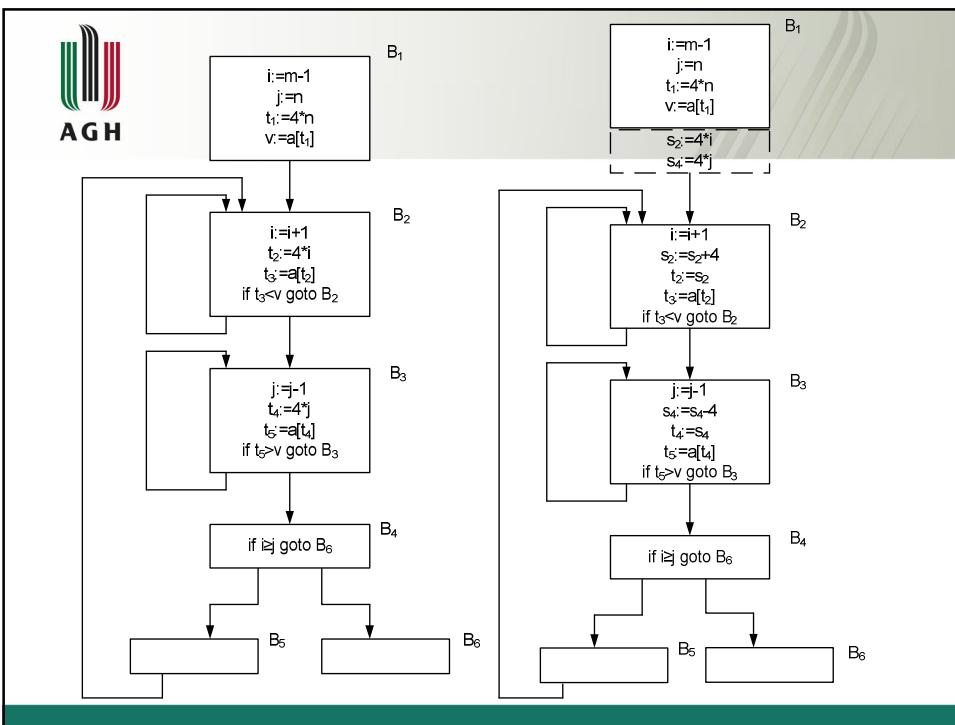
### Metoda:

Rozważamy kolejno każdą podstawową zmienną indukowaną  $i$ . Dla każdej pochodnej zmiennej indukowanej  $j$  z rodziny zmiennej  $i$  (zmienna  $j$  scharakteryzowana trójką  $(i, c, d)$ ) wykonaj:

- (1) Utwórz nową zmienną  $s$  (jeśli dwie zmienne pochodne  $j_1$  i  $j_2$  mają takie same trójki, utwórz dla obu tylko jedną nową zmienną).
- (2) Zamień podstawienie do zmiennej  $j$  na  $j := s$ .
- (3) Bezpośrednio po podstawieniu  $i := i + a$  (gdzie:  $a$  – stała) dodaj  $s := s + c \cdot a$ , gdzie wyrażenie  $c \cdot a$  jest obliczone i podane jako stała, gdyż  $c$  oraz  $a$  są stałymi. Umieść zmienną  $s$  w rodzinie zmiennej podstawowej  $i$  z odpowiadającą jej trójką  $(i, c, d)$ .
- (4) Wstaw na koniec przednagłówka pętli

$$\begin{aligned}s &:= c \cdot i & (s := i \text{ gdy } c = 1) \\ s &:= s + d & (\text{opuszczyć, gdy } d = 0)\end{aligned}$$

Zauważ, że  $s$  jest także pochodną zmienną indukowaną z rodziny zmiennej  $i$ .



## Algorytm eliminacji zmiennych indukowanych (1)

**Wejście:** pętla  $L$  wraz z informacją o zasięgu definicji, o wyrażeniach niezmienniczych i o życiu zmiennych.

**Wyjście:** przeredagowana pętla.

**Metoda:**

- (1) Rozważamy kolejno każdą podstawową zmienną indukowaną, która używana jest wyłącznie do obliczania innych zmiennych indukowanych ze swojej rodziny oraz w wyrażeniu w skokach warunkowych wewnętrz pętli. Weź pewną zmienną  $j$  z rodziny zmiennej  $i$ , najlepiej taką, aby  $c$  i  $d$  z jej trójki były możliwe najprostsze (na przykład  $c = 1$ ,  $d = 0$ ). Zmodyfikuj każde wyrażenie w skokach warunkowych tak, aby zamiast  $i$  było tam używane  $j$ . Założymy dalej, że  $c > 0$ .

Skok warunkowy if i rel\_op x goto B, gdzie  $x$  nie jest zmienną indukowaną, jest zamieniany na:

$r := c * x$  ( $r := x$ , gdy  $c = 1$ )  
 $r := r + d$  (opusć, gdy  $d = 0$ )

if j rel\_op r goto B

gdzie  $r$  jest nową zmienną tymczasową.



## Algorytm eliminacji zmiennych indukowanych (2)

Skok warunkowy

if  $i \text{ rel\_op } a \text{ goto } B$ , gdzie  $a$  jest stałą jest zamieniany na:

if  $j \text{ rel\_op } c \cdot a + d \text{ goto } B$ , gdzie wyrażenie  $c \cdot a + d$  jest wyliczone. W przypadku, kiedy w teście skoku warunkowego są używane dwie zmienne indukowane  $i_1$  i  $i_2$  (if  $i_1 \text{ rel\_op } i_2 \text{ goto } B$ ), sprawdzamy, czy obie mogą być usunięte. W najprostszym przypadku, gdy  $j_1$  ma trójkę  $(i_1, c_1, d_1)$ ,  $j_2$  zaś  $(i_2, c_2, d_2)$  oraz  $c_1 = c_2$  i  $d_1 = d_2$ , wtedy if  $i_1 \text{ rel\_op } i_2$  jest równoważne if  $j_1 \text{ rel\_op } j_2$ .

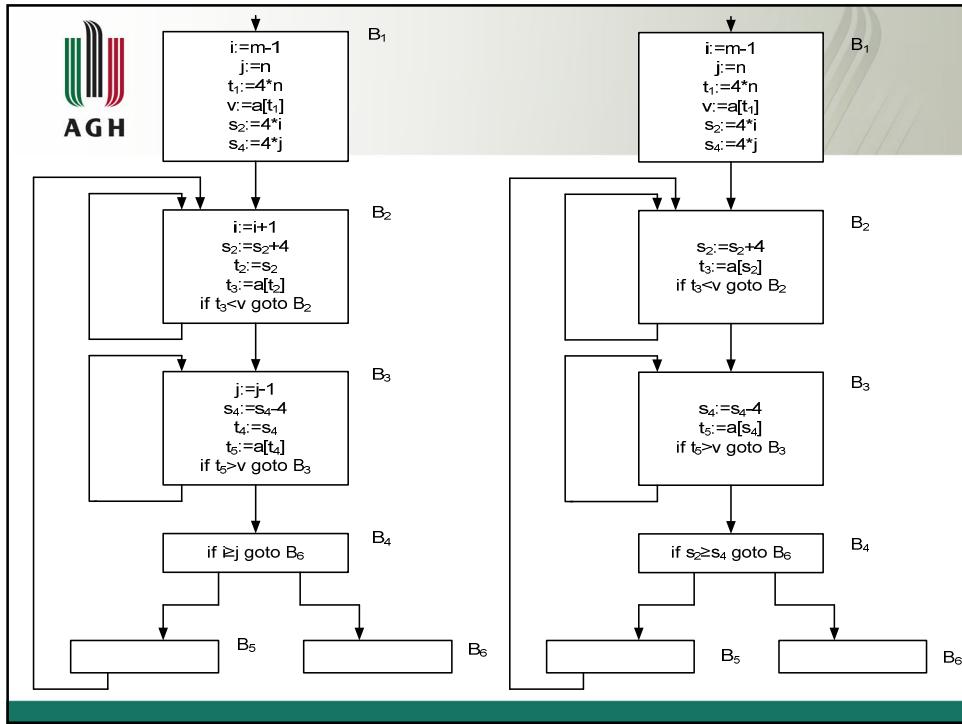
W bardziej złożonych przypadkach modyfikacja testu przy skokach warunkowych może nie być warta zachodu, gdyż musielibyśmy (w ogólnym przypadku) wprowadzić dwa dodatkowe mnożenia i jedno dodawanie.

Po przekształceniu skoków warunkowych usuwamy wszystkie podstawienia do wyeliminowanych podstawowych zmiennych indukowanych, gdyż zmienne te są teraz bezużyteczne.



## Algorytm eliminacji zmiennych indukowanych (3)

- (2) Rozważamy teraz każdą zmienną indukowaną  $j$ , dla której podstawienie  $j := s$  zostało wstawione podczas wykonania poprzedniego algorytmu. Sprawdzamy, czy nie ma jakiegoś podstawienia do zmiennej  $s$  pomiędzy instrukcją  $j := s$  i jakimkolwiek użyciem zmiennej  $j$ . Jeśli nie ma, zastępujemy wszystkie użycia  $j$  przez  $s$  i usuwamy podstawienie  $j := s$ .



## Zmienne indukowane a wyrażenia niezmiennicze w pętli

W algorytmach wykrywania zmiennych indukowanych i redukcji mocy można dopuścić wyrażenia niezmiennicze oprócz stałych. Jednak wówczas określenie  $(i, c, d)$  dla zmiennej indukowanej  $j$  wymagałoby znajomości wartości wyrażenia niezmiennicznego. Obliczenie trójki musiałoby się odbyć w przednagłówku, w pewnych przypadkach wymagałoby kilku instrukcji trójadresowych. Algorytm eliminacji zmiennych indukowanych wymaga z kolei znajomości znaku  $c$ ; gdyby  $c$  było ujemne, należałoby zmienić operator relacji w teście skoku warunkowego. Dlatego też z reguły ograniczamy się tylko do stałych przy wykrywaniu zmiennych indukowanych.



**AGH**

AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

# **Postępowanie z synonimami**

## **Teoria komplikacji**

**Dr inż. Janusz Majewski  
Katedra Informatyki**

# Postępowanie z synonymami (aliases) związanymi ze wskaźnikami

Jeśli dwa lub więcej wyrażenia oznaczają te samą lokalizację w pamięci, nazywamy je synonymami. Będziemy rozważać dwa źródła powstawania synonymów: wskaźniki i procedury.

Jeśli nie wiemy nic na temat: gdzie wskazuje pointer, jedynym bezpiecznym założeniem jest, że pośrednie podstawienie poprzez wskaźnik ( $*p := x$ ) może potencjalnie zmienić (definiować) każdą zmienną, a także że użycie danej wskazywanej przez pointer ( $x := *p$ ) może potencjalnie dotyczyć każdej zmiennej. Rezultatem takiego założenia jest nierealistycznie duża liczba dostępnych definicji i żywych zmiennych oraz nierealistycznie mała liczba dostępnych wyrażeń.

# Postępowanie z synonymami (aliases) związanymi ze wskaźnikami

Dlatego istotne jest uzyskanie wiarygodnej informacji na temat: na co w danym miejscu programu może wskazywać pointer.

Informację tę możemy uzyskać metodą analizy przepływu danych, co będzie zilustrowane na przykładzie prostego języka dopuszczającego wykorzystanie wskaźników. Założymy, że język ten operuje na elementarnych danych, z których każda zajmuje jedną komórkę pamięci i na tablicach, których elementami są dane zajmujące pojedyncze komórki pamięci. Pointery mogą wskazywać na dane elementarne lub tablice, ale nie mogą wskazywać na inne pointery. Wystarczającą informacją dla nas będzie, że pointer wskazuje na jakiś element tablicy, bez precyzowania, który konkretny element jest wskazywany przez pointer.

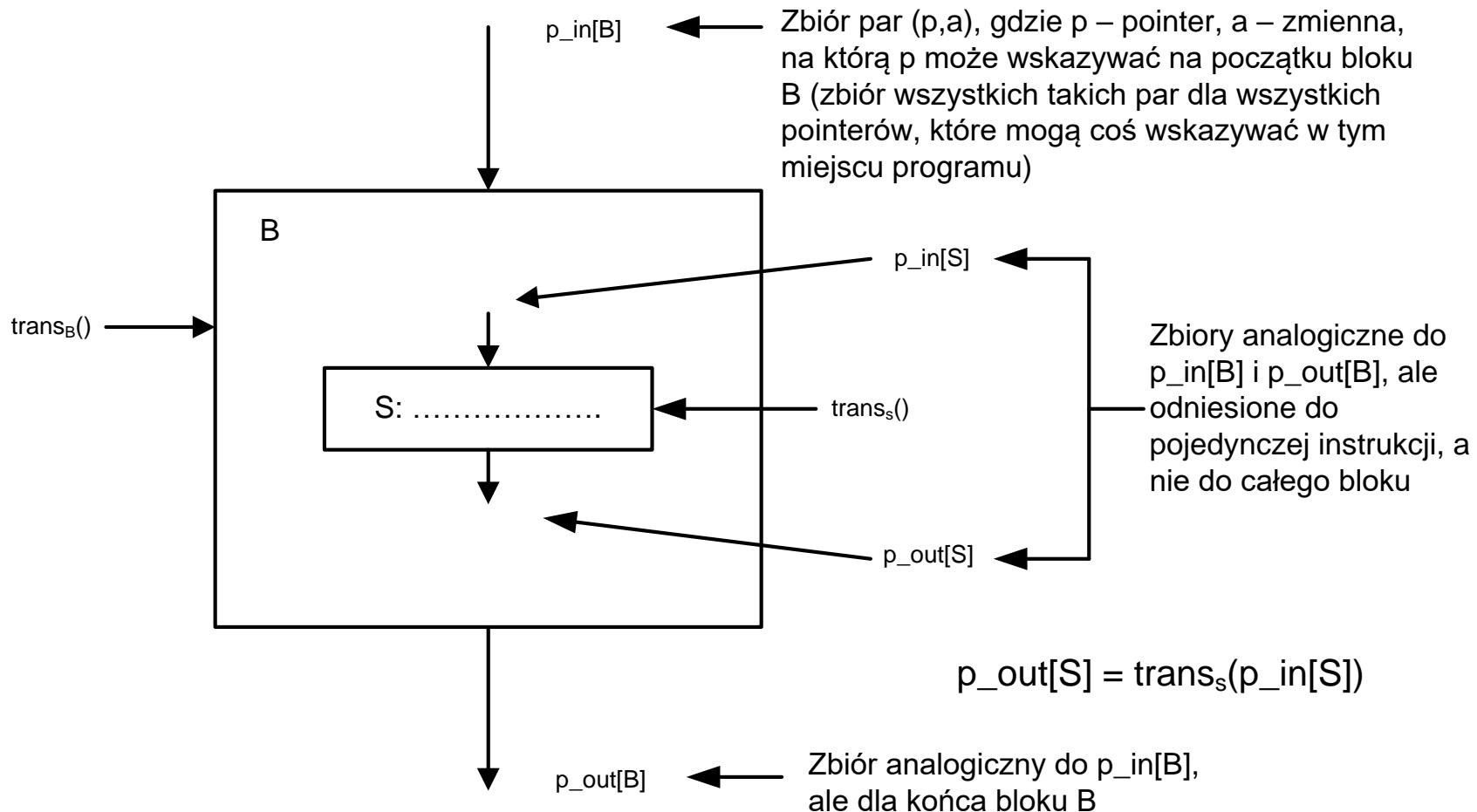
# Postępowanie z synonymami (aliases) związanymi ze wskaźnikami

Załóżmy ponadto, że dozwolone jest dodanie lub odjęcie liczby całkowitej tylko do pointera wskazującego na tablicę i że po takiej operacji arytmetycznej pointer w dalszym ciągu skazuje na pewien element tej tablicy. Dozwolone jest także podstawienie do pointera innego wskaźnika i przypisanie pointerowi adresu zmiennej elementarnej lub tablicy, a także wykorzystywanie pointera do odwołań pośrednich do wskazywanych przez niego danych. Zasady określające, co dany pointer wskazuje w danym miejscu programu, zostaną przedstawione na kolejnym slajdzie:

# Postępowanie z synonymami (aliases) związanymi ze wskaźnikami

- (1) Bezpośrednio po instrukcji  $s : p := \&a$  pointer  $p$  wskazuje tylko na  $a$ . Jeśli  $a$  jest tablicą, to pointer  $p$  bezpośrednio po instrukcji  $s : p := \&a \pm c$  (gdzie  $c$  jest całkowitą stałą lub zmienną) wskazuje tylko na tablicę  $a$  (na jakiś jej element).
- (2) Bezpośrednio po instrukcji  $s : p := q \pm c$  (gdzie  $p$  i  $q$  są pointerami, zaś  $c$  jest całkowite i różne od zera) pointer wskazuje na wszystkie tablice, na które wskazywał  $q$  przed wykonaniem  $s$  i na nic więcej (nie wskazuje na żadną ze zmiennych elementarnych, na którą wskazywał  $q$ ).
- (3) Bezpośrednio po instrukcji  $s : p := q$  pointer  $p$  wskazuje na wszystko, na co wskazywał  $q$  przed wykonaniem  $s$ .
- (4) Po każdym innym podstawieniu do  $p$  – nie ma żadnego obiektu, na który  $p$  mógłby wskazywać.
- (5) Po każdym podstawieniu do zmiennej innej niż  $p$ , pointer  $p$  wskazuje na ten sam zbiór obiektów, na który wskazywał przed takim podstawieniem (przypomnienie: pointer nie może wskazywać na inny pointer).

# Analiza przepływu danych dla problemu wskaźników



# Analiza przepływu danych dla problemu wskaźników

Określanie funkcji przejścia  $trans_S(p\_in[S])$ :

- (1) Jeśli  $S$  ma postać  $p := \&a$  lub  $p := \&a \pm c$  to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p, b) \mid \text{dla każdego } b\} \cup \{(p, a)\}$$

- (2) Jeśli  $S$  ma postać  $p := q \pm c$ , gdzie  $q$  – pointer,  $c$  – całkowite  $\neq 0$ , to:

$$\begin{aligned} trans_S(p\_in[S]) &= p\_in[S] - \{(p, b) \mid \text{dla każdego } b\} \\ &\cup \{(p, d) \mid (q, d) \in p\_in[S] \wedge d \text{ jest tablicą}\} \end{aligned}$$

- (3) Jeśli  $S$  ma postać  $p := q$ , to:

$$\begin{aligned} trans_S(p\_in[S]) &= p\_in[S] - \{(p, b) \mid \text{dla każdego } b\} \\ &\cup \{(p, d) \mid (q, d) \in p\_in[S]\} \end{aligned}$$

- (4) Jeśli w  $S$  podstawia się do pointera  $p$  coś innego, to:

$$trans_S(p\_in[S]) = p\_in[S] - \{(p, b) \mid \text{dla każdego } b\}$$

- (5) Jeśli  $S$  nie jest podstawieniem do pointera, to:

$$trans_S(p\_in[S]) = p\_in[S]$$

# Analiza przepływu danych dla problemu wskaźników

Funkcję przejścia dla bloku  $B$ , składającego się z instrukcji  $S_1, S_2, \dots, S_k$  określa się następująco:

$$trans_B(p\_in[B]) =$$

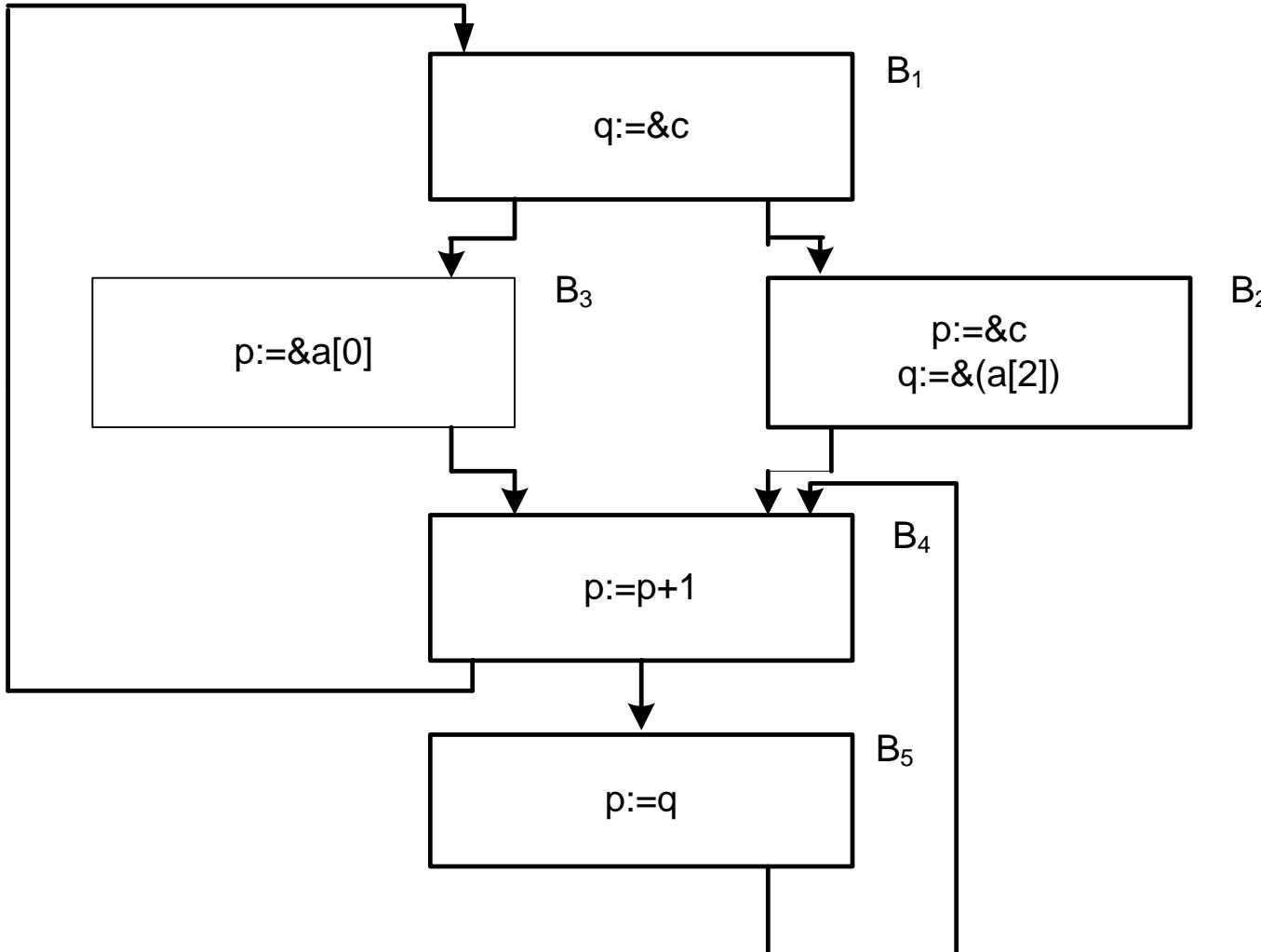
$$trans_{S_k}(trans_{S_{k-1}}(\dots(trans_{S_2}(trans_{S_1}(p\_in[B])))\dots))$$

Informację dotyczącą problemu wskaźników wyznacza się poprzez wykonanie iteracyjnego algorytmu analizy przepływu danych, analogicznego do algorytmu zasięgu definicji, wykorzystując równania:

$$p\_out[B] = trans_B(p\_in[B])$$

$$P\_in[B] = \bigcup_{P - \text{poprzednik bloku } B} P\_out[P]$$

# Analiza przepływu danych dla problemu wskaźników – przykład



# Analiza przepływu danych dla problemu wskaźników – przykład

Pierwszy przebieg:

Zakł. początkowo:  $p\_in[B_1] = \emptyset$

$$p\_out[B_1] = transB_1(p\_in[B_1]) = transB_1(\emptyset) = \{(q,c)\}$$

$$p\_in[B_2] = p\_out[B_1] = \{(q,c)\}$$

$$p\_out[B_2] = transB_2(p\_in[B_2]) = transB_2(\{q,c\}) = \{(p,c), (q,a)\}$$

$$p\_in[B_3] = p\_out[B_1] = \{q,c\}$$

$$p\_out[B_3] = transB_3(p\_in[B_3]) = transB_3(\{q,c\}) = \{(p,a), (q,c)\}$$

$$p\_in[B_4] = p\_out[B_2] \cup p\_out[B_3] \cup p\_out[B_5];$$

zakł.  $p\_out[B_5] = \emptyset$

$$p\_in[B_4] = \{(p,a), (p,c), (q,a), (q,c)\}$$

$$p\_out[B_4] = trans_{B_4}(p\_in[B_4]) = \{(p,a), (q,a), (q,c)\};$$

$c$  nie jest tablicą

$$p\_in[B_5] = p\_out[B_4] = \{(p,a), (q,a), (q,c)\}$$

$$p\_out[B_5] = trans_{B_5}(p\_in[B_5]) = \{(p,a), (p,c), (q,a), (q,c)\}$$

# Analiza przepływu danych dla problemu wskaźników – przykład

Drugi przebieg:

$$p\_in[B_1] = p\_out[B_4] = \{(p, a), (q, a), (q, c)\}$$

$$p\_out[B_1] = trans_{B_1}(p\_in[B_1]) = \{(p, a), (q, c)\}$$

$$p\_in[B_2] = p\_out[B_1] = \{(p, a), (q, c)\}$$

$$p\_out[B_2] = trans_{B_2}(p\_in[B_2]) = \{(p, c), (q, a)\}$$

$$p\_in[B_3] = p\_out[B_1] = \{(p, a), (q, c)\}$$

$$p\_out[B_3] = trans_{B_3}(p\_in[B_3]) = \{(p, a), (q, c)\}$$

$$p\_in[B_4] = p\_out[B_2] \cup p\_out[B_3] \cup p\_out[B_4] = \{(p, a), (p, c), (q, a), (q, c)\}$$

...dalej bez zmian.

Mając  $p\_in[B]$  i  $p\_out[B]$  dla wszystkich bloków możemy wyznaczyć  $p\_in[S]$  i  $p\_out[S]$  dla wszystkich instrukcji wewnątrz bloków.

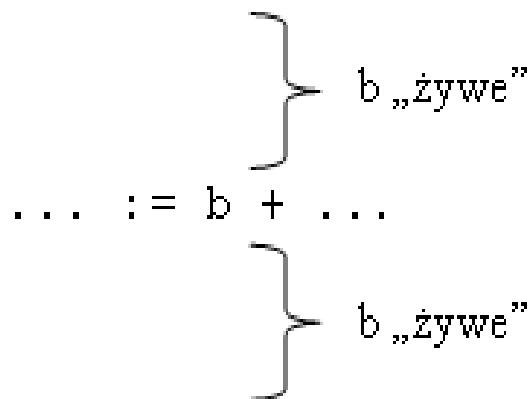
# Analiza przepływu danych dla problemu wskaźników – przykład

Przykład:

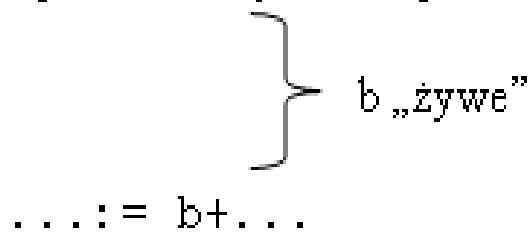
Problem życia zmiennych rozwiązywany jest przez wcześniej podany algorytm; należy jedynie rozważyć sposób wyznaczania zbiorów *def* i *use* dla instrukcji typu  $*p := a$  oraz  $a := *p$ . Instrukcja podstawienia pośredniego  $*p := a$  używa tylko  $p$  i  $a$ . Zakładamy natomiast, że  $*p := a$  definiuje  $b$  tylko wtedy, jeśli  $b$  jest jedyną zmienną, na którą może wskazywać pointer  $p$ , czyli gdy jesteśmy pewnie, że  $b$  rzeczywiście jest definiowalne. W ten sposób nigdy nie dopuścimy do sytuacji, że w pewnym punkcie programu przyjmujemy, że zmienna jest martwa, podczas gdy faktycznie jest ona żywa.

# Analiza przepływu danych dla problemu wskaźników – przykład

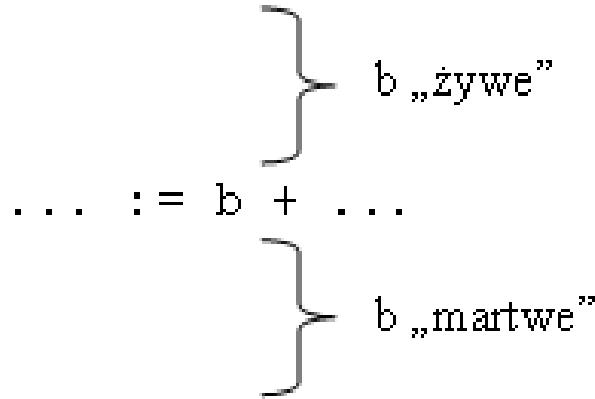
S1:     $b := \dots$



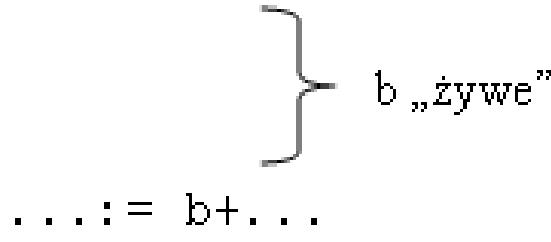
S2:     $*p := a \rightarrow$  być może p wskazuje b



S1:     $b := \dots$



S2:     $*p := a \rightarrow$  na pewno p wskazuje b



# Analiza przepływu danych dla problemu wskaźników – przykład

Instrukcja  $a := *p$  definiuje tylko  $a$ , zaś używa  $p$  i każdej zmiennej  $b$ , na którą pointer  $p$  może wskazywać. W ten sposób mamy być może zbyt dużo zmiennych żywych, ale zmienne martwe można w trakcie np. generacji kodu odkładać do pamięci, a zmiennych żywych nie należy, powinny być w rejestrze tak długo, jak tylko można.

# Wykorzystanie informacji o wskaźnikach

Algorytm poprzedni daje informacje typu: „co może wskazywać każdy pointer w miejscach, gdzie jest używany w pośrednich odwołaniach rodzaju  $*p := a$ ,  $a := *p$ , itd.”. Te informacje można wykorzystać w poprzednio omawianych problemach analizy przepływu danych. Zawsze należy postępować tak, aby ewentualne błędy miały charakter zachowawczy (aby nic nie popsuły).

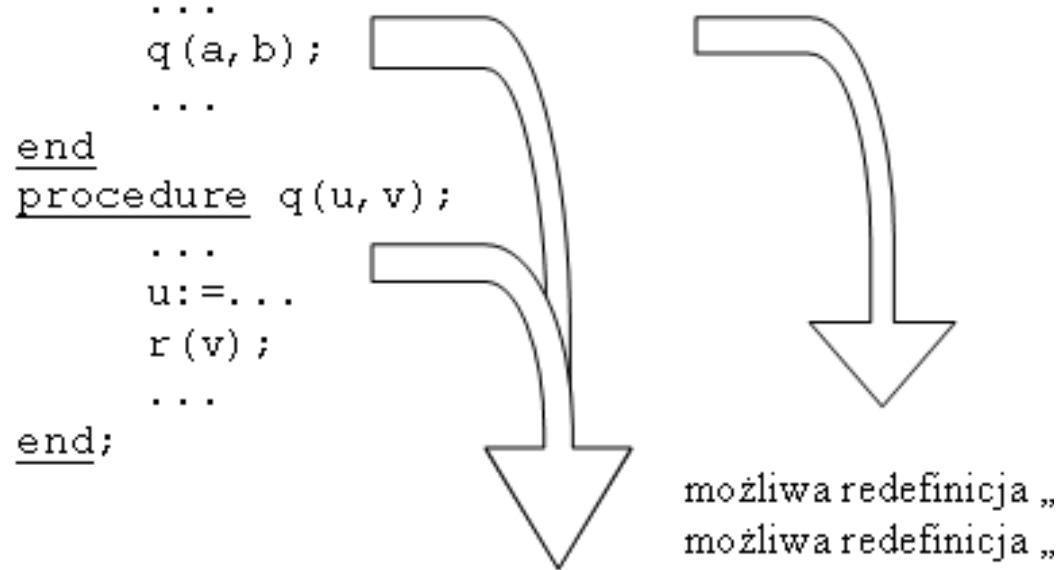
# Wykorzystanie informacji o wskaźnikach – przykład

Problem zasięgu definicji jest rozwiązywany przy wykorzystaniu wcześniejszego algorytmu; należy tylko zmodyfikować sposób wyznaczenia zbiorów *gen* i *kill*. Przy wyznaczaniu zbioru *gen* przyjmuje się, że pośrednie podstawienie  $*p := a$  generuje definicje każdej zmiennej *b*, na którą pointer *p* może wskazywać. Jest to podejście zachowawcze, gdyż lepiej przyjąć, że pewna definicja osiąga dany punkt, podczas gdy rzeczywiście tak nie jest, niż na odwrót – uznać, że pewna definicja nie osiąga danego punktu, kiedy jednak naprawdę go osiąga. Podczas wyznaczania zbioru *kill* zakładamy, że podstawienie pośrednie  $*p := a$  zabija definicję zmiennej *b* tylko wtedy, gdy *b* nie jest tablicą i równocześnie *b* jest jedyną zmienną, na którą pointer *p* może wskazywać w tym miejscu programu. Czyli  $*p := a$  zabija definicję *b* wskazywanej przez *p* tylko wówczas, gdy jesteśmy tego absolutnie pewni.

# Synonimy związane z procedurami

Przykład: (przekazywanie parametrów przez referencje)

```
global a; ← zmienne globalne
procedure p(c, d);
 local b; ← zmienne lokalne w rozważa-
 ...
 q (a, b);
 ...
end
procedure q(u, v);
 ...
 u:=...
 r (v);
 ...
end;
```



możliwa redefinicja „  
możliwa redefinicja „

potencjalne  
synonimy:  
 $u \equiv a$   
 $v \equiv b$

# Synonimy związane z procedurami

Założenia dotyczące przykładowego języka:

- (1) Dopuszczalne rekurencyjne wywołania procedur,
- (2) Procedury mogą operować na zmiennych globalnych oraz swoich własnych zmiennych lokalnych (nie ma blokowej struktury  $\equiv$  zagłębiania procedur),
- (3) Parametry są przekazywane przez referencje (adres),
- (4) Każda procedura ma pojedynczy punkt wejścia i pojedynczy punkt wyjścia.

# Prosty algorytm znajdowania synonimów

Wejście:

Wyjście:

zbiór procedur i zmiennych globalnych  
relacja równoważności  $\equiv$  o tej własności, że  
zawsze jeśli w programie  $x$  oraz  $y$  są  
wzajemnie synonymami, to  $x \equiv y$ .  
Zależność odwrotna nie musi być  
prawdziwa. Relację  $\equiv$  można nazwać: „może  
być synonymem”.

# Prosty algorytm znajdowania synonimów

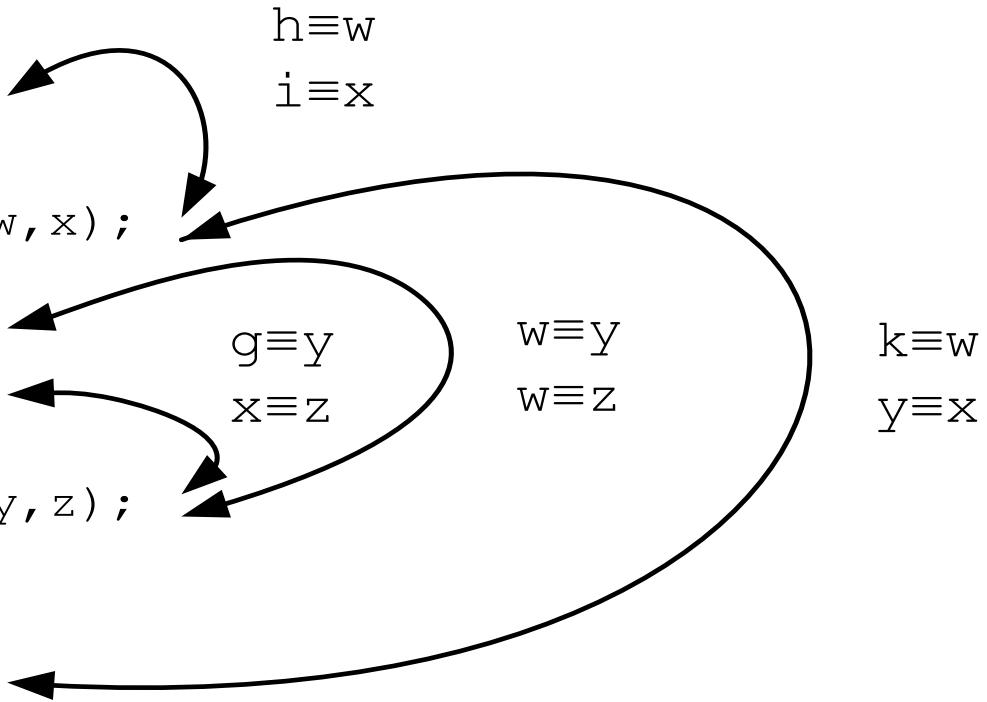
## Metoda:

- (1) Jeśli jest to konieczne, zmień nazwy zmiennych tak, aby żadne dwie procedury nie używały tych samych identyfikatorów zmiennych lokalnych i parametrów formalnych oraz aby identyfikatory zmiennych globalnych były różne od identyfikatorów zmiennych lokalnych lub parametrów formalnych procedur.
- (2) Jeśli jest procedura  $p(x_1, x_2, \dots, x_n)$  oraz wywołanie  $p(y_1, y_2, \dots, y_n)$  tej procedury, dołącz do zbioru relacji  $\equiv$  pary  $x_i \equiv y_i$  dla wszystkich  $i$ , gdyż każdy parametr formalny może być synonimem odpowiadającego mu parametru aktualnego.
- (3) Stwórz przechodnie i zwrotne domknięcie relacji  $\equiv$ , dodając:
  - (a)  $x \equiv x$  dla wszystkich parametrów aktualnych i formalnych,
  - (b)  $x \equiv y$  gdy  $y \equiv x$ ,
  - (c)  $x \equiv z$  gdy  $x \equiv y$  oraz  $y \equiv z$  dla pewnego  $y$ .

# Prosty algorytm znajdowania synonimów

## Przykład

```
global g,h;
procedure main();
 local i;
 g:=...;
 one(h,i);
end;
procedure one(w,x);
 x:=...;
 two(w,w);
 two(g,x);
end;
procedure two(y,z);
 local k;
 h:=...;
 one(k,y)
end;
```



# Prosty algorytm znajdowania synonimów

|   | g | h | i | w | x | y | z | k |
|---|---|---|---|---|---|---|---|---|
| g |   |   |   |   |   | 1 |   |   |
| h |   |   |   | 1 |   |   |   |   |
| i |   |   |   |   | 1 |   |   |   |
| w |   |   |   |   |   | 1 | 1 |   |
| x |   |   |   |   |   |   | 1 |   |
| y |   |   |   |   |   |   | 1 |   |
| z |   |   |   |   |   |   |   |   |
| k |   |   |   | 1 |   |   |   |   |

Relacja  $\equiv$  po kroku (2)

|   | g | h | i | w | x | y | z | k |
|---|---|---|---|---|---|---|---|---|
| g | 1 |   |   |   |   | 1 |   |   |
| h |   | 1 |   |   |   | 1 |   |   |
| i |   |   | 1 |   |   | 1 |   |   |
| w |   |   |   | 1 |   | 1 | 1 | 1 |
| x |   |   |   |   | 1 |   | 1 | 1 |
| y | 1 |   |   |   |   | 1 | 1 | 1 |
| z |   |   |   |   | 1 | 1 | 1 | 1 |
| k |   |   |   | 1 |   |   |   | 1 |

Relacja  $\equiv$  po kroku (3)(b)

Po wykonaniu kroku (3)(c) okaże się, że każda z rozważanych zmiennych może być synonimem każdej innej. Tak uzyskana relacja jest „zbyt duża” (np.  $g$  i  $h$  na pewno zajmują, jako zmienne globalne, odrębne miejsce w pamięci).

# Synonimy związane z procedurami

Problem synonimów wnoszonych przez wywołania procedur zilustrujemy na przykładzie modyfikacji procesu eliminacji wspólnych podwyrażeń.

Dla każdej procedury  $p$  definiujemy zbiór  $change[p]$ , którego elementami są zmienne globalne i formalne parametry procedury  $p$ , które mogą być zmieniane w trakcie wykonania procedury  $p$ . Niech  $def[p]$  będzie zbiorem zmiennych globalnych i parametrów formalnych procedury  $p$ , które mają w tej procedurze bezpośrednie definicje (poprzez instrukcje podstawienia, a nie poprzez parametry aktualne).

Wtedy:

$$change[p] = def[p] \cup A \cup G$$

gdzie:

- A jest zbiorem takich zmiennych globalnych i formalnych parametrów procedury  $p$ , które są aktualnymi parametrami wywołania procedur  $q$  z wnętrza  $p$ , a odpowiadające im parametry formalne procedury  $q$  należą do  $change[q]$ .
- G jest zbiorem zmiennych globalnych należących do  $change[q]$ , przy czym  $q$  są procedurami wywoływanymi z wnętrza  $p$ .

# Synonimy związane z procedurami

Zależy nam na rozwiązaniu równania

$$change[p] = def[p] \cup A \cup G$$

dla wszystkich procedur (metodą iteracyjną), przy czym chcemy uzyskać rozwiązanie „najmniejsze”. Tworzymy graf wywołań, którego węzły są procedurami, zaś krawędź od  $p$  do  $q$  jest obecna wtedy i tylko wtedy, gdy  $p$  wywołuje  $q$ . O ile to możliwe, w algorytmie rozpoczynamy przeszukiwanie grafu wywołań od węzła, który odpowiada procedurze nie wywołującej żadnej innej procedury.

# Algorytm wyznaczania zbiorów *change[p]* dla procedur

- Wejście:** zbiór procedur  $p_1, p_2, \dots, p_n$ . Jeżeli graf wywołań jest acykliczny, zakładamy, że  $p_i$  wywołuje  $p_j$  tylko gdy  $j < i$ . W przeciwnym wypadku nie stawiamy takiego wymagania.
- Wyjście:** dla każdej procedury  $p$  tworzony jest zbiór  $change[p]$  zawierający zmienne globalne i parametry formalne, które mogą być zmienione w procedurze  $p$ .
- Metoda:** wyznaczamy dla każdej procedury  $p$  zbiór  $def[p]$  zgodnie z definicją. Następnie:



AGH

# Synonimy związane z procedurami

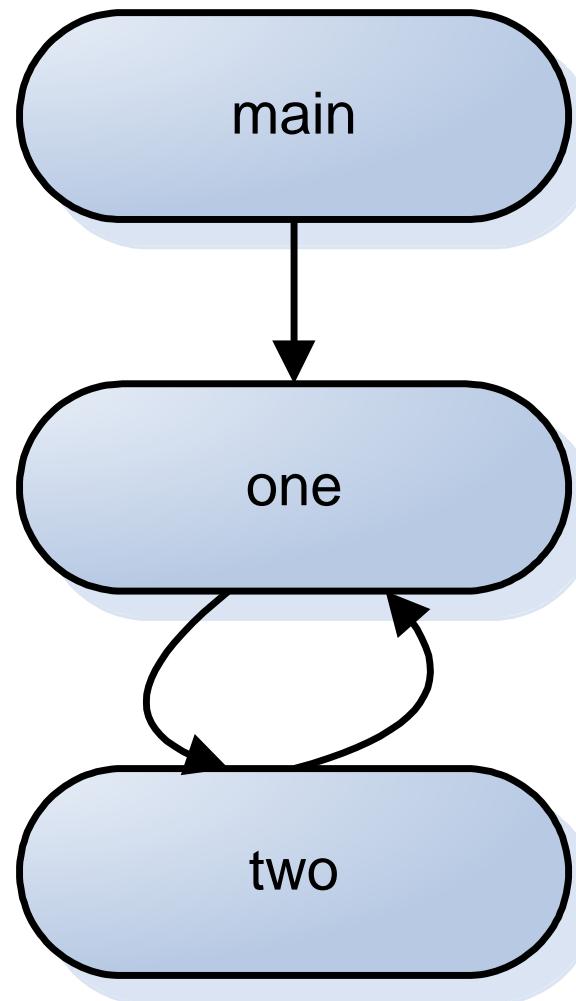
(1)    for każda procedura p do change[p] :=def[p];  
(2)    while zmiany występują w jakimkolwiek change[p] do  
(3)       for i:=1 to n do  
(4)          for każda procedura q wywoływana przez [p<sub>i</sub>] do  
            begin  
(5)             dołącz każdą zmienną globalną z change[q] do  
               change [p<sub>i</sub>];  
(6)             for każdy parametr formalny x (j-ty) procedury  
               q do  
(7)                if x należy do change[q] then  
(8)                   for każde wywołanie q w p<sub>i</sub> do  
(9)                      if a - j-ty aktualny parametr  
               wywołania jest zmienną globalną lub  
               parametrem formalnym w p<sub>i</sub>  
               then dołącz a do zbioru change[p<sub>i</sub>]  
(10)  
(11)          end

Powyższy algorytm działa niezależnie od poprzedniego i nie korzysta z tamtych wyników.

# Synonimy związane z procedurami

Przykład:

```
global g,h;
procedure main () ;
 local i;
 g:=...;
 one (h,i);
end;
procedure one (w,x) ;
 x:=...;
 two (w,w);
 two (g,x);
end;
procedure two (y,z) ;
 local k;
 h:=...;
 one (k,y);
end;
```



# Synonimy związane z procedurami

Ustalamy kolejność:

*two, one, main*

Początkowo:

*def[two] = {h}*

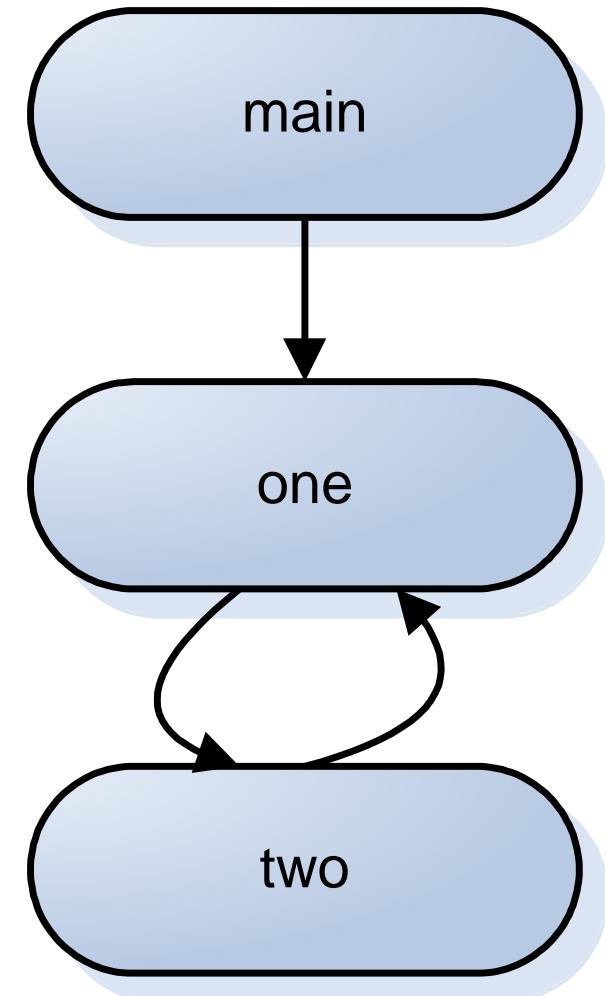
*def[one] = {x}*

*def[main] = {g}*

*change[two] = {h}*

*change[one] = {x}*

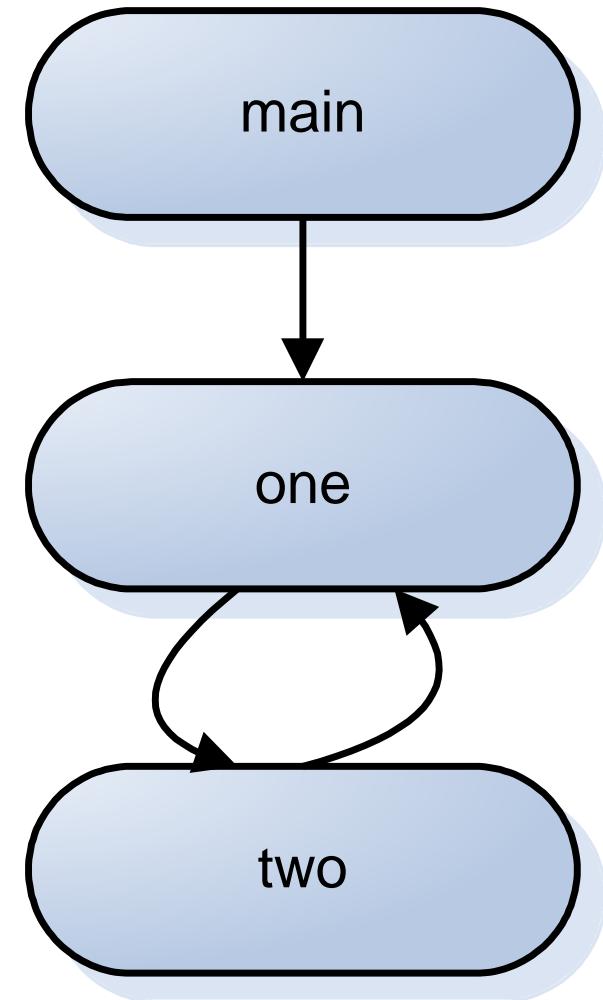
*chane[main] = {g }*



# Synonimy związane z procedurami

Przebieg pierwszy:

- (a)  $p_1 = \text{two}$   
(4)  $q = \text{one}$   
(5) nic nowego  
(6),(7) drugi formalny parametr procedury *one*  
(10)  $\text{change}[\text{two}] = \{h, y\}$
- (b)  $p_2 = \text{one}$   
(4)  $q = \text{two}$   
(5)  $\text{change}[\text{one}] = \{h, y\}$   
(6),(7) pierwszy formalny parametr procedury *one*  
(10)  $\text{change}[\text{one}] = \{h, y, w, g\}$
- (c)  $p_3 = \text{main}$   
(4)  $q = \text{one}$   
(5)  $\text{change}[\text{main}] = \{g, h\}$   
dalej bez zmian



# Synonimy związane z procedurami

Przebieg drugi:

(a)  $p_1 = \text{two}$

(4)  $q = \text{one}$

(5)  $\text{change}[\text{two}] = \{h, y, g\}$

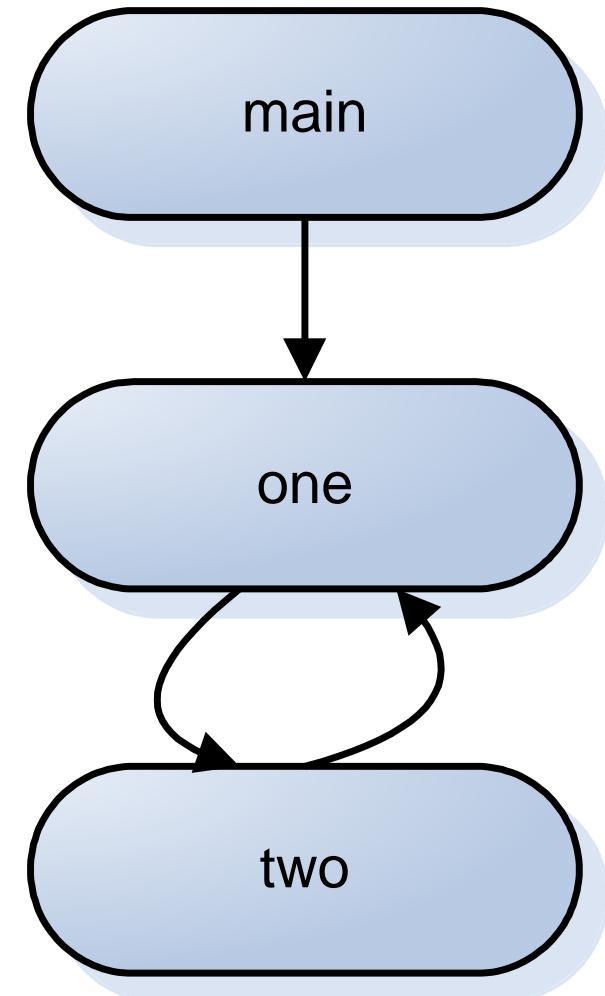
dalej bez zmian

Ostatecznie:

$\text{change}[\text{two}] = \{g, h, y\}$

$\text{change}[\text{one}] = \{g, h, w, x\}$

$\text{change}[\text{main}] = \{g, h\}$





AGH

# Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Założymy, że mamy wyznaczyć zbiór  $e\_kill[B]$  dla pewnego bloku  $B$  należącego do procedury  $p$ . Definicja zmiennej  $a$  musi być uznana za zabijającą każde wyrażenie odwołujące się do  $a$  lub odwołujące się do pewnego  $x$  które może być synonimem  $a$ . Jednak wywołanie w bloku  $B$  procedury  $q$  nie może zabić wyrażenia zawierającego  $a$ , chyba, że  $a$  jest synonimem (zauważ, że  $a$  jest synonimem samego siebie) pewnej zmiennej ze zbioru  $change[q]$ . Wyznaczenie zbioru  $e\_kill[B]$  wymaga więc znajomości relacji  $\equiv$  oraz zbiorów  $change[]$  dla wszystkich procedur.



AGH

# Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Drugim problemem jest kwestia rozstrzygnięcia, kiedy wywołanie procedury może generować wyrażenie  $a \text{ op } b$ . Powinniśmy założyć, że  $a \text{ op } b$  jest generowane przez wywołanie  $q$  wtedy i tylko wtedy, gdy na każdej ścieżce od punktu wejściowego procedury  $q$  do jej punktu wyjściowego  $a \text{ op } b$  jest wyliczane i nie ma później żadnej redefinicji ani  $a$  ani  $b$ . Poszukując wystąpienia  $a \text{ op } b$  nie możemy zaakceptować  $x \text{ op } y$  jako takiego wystąpienia dopóki nie jesteśmy pewni, że w każdym wywołaniu  $q$   $x$  jest (co nie znaczy „może być”!) synonimem  $a$  oraz  $y$  synonimem  $b$ . Najprościej więc przyjąć, że wywołanie  $q$  nie generuje niczego.



AGH

# Wykorzystanie zebranych informacji w problemie dostępności wyrażeń i eliminacji wspólnych podwyrażeń

Bardziej skomplikowanym, lecz bliższym rzeczywistości rozwiązaniem jest iteracyjne wyznaczenie zbioru  $gen[p]$  (zbioru generowanych dostępnych wyrażeń) dla każdej procedury  $p$ . W algorytmie podobnym do tego, który wyznaczał zbiory  $change[p]$  możemy zainicjować zbiory  $gen[p]$  tak, aby zawierały wszystkie wyrażenia dostępne na końcu bloku wyjściowego z danej procedury  $p$ . Musimy jednak pamiętać, że obliczając zbiory  $gen[p]$  nie wolno nam uwzględniać informacji o synonimach;  $a \ op \ b$  reprezentuje tutaj tylko samego siebie, nawet gdyby inne zmienne mogły być synonimami  $a$  lub  $b$ . Wyznaczanie zbiorów  $gen[p]$  prowadzi do iteracyjnego przeglądania wszystkich bloków z wszystkich procedur i znajdowania wyrażeń dostępnych. Wywołania  $q(a,b)$  procedury  $q$  w procedurze  $p$  generuje te wyrażenia, które zawarte są w  $gen[q]$  (z uwzględnieniem substytucji odpowiednich parametrów formalnych). Zbiory  $e\_kill[B]$  pozostają bez zmiany. Nowa zawartość  $gen[p]$  dla każdej procedury  $p$  może być określona, jeśli znajdziemy wyrażenia dostępne w węźle wyjściowym procedury  $p$ . Iterację prowadzimy tak długo, jak długo pojawiają się jakiekolwiek zmiany zbiorów wyrażeń dostępnych w poszczególnych węzłach.



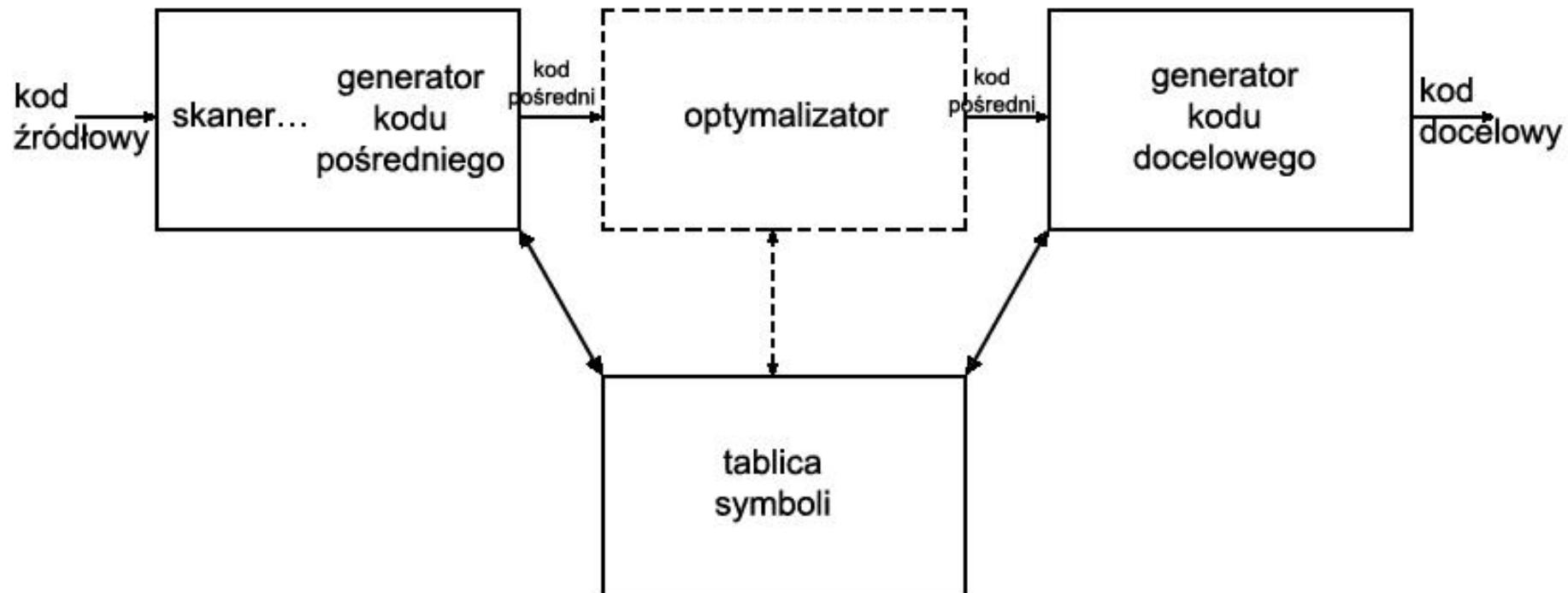
**AGH**

AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA W KRAKOWIE

# **Generacja kodu ostatecznego**

**Dr inż. Janusz Majewski  
Teoria kompilacji**

# Generacja kodu ostatecznego



# Generator kodu docelowego

We : - kod pośredni      *niezależny sprzętowo*

- kod assemblera

Wy : - kod relokowalny

- kod absolutny

*zależny sprzętowo*

# Rozkazy maszynowe

- Przyjmujemy następujący format rozkazu maszynowego:

**operacja docelowy, źródłowy**

Przykład:

**add R0, R1 /\* dodaj zawartość rejestru R1 do rejestru R0, wynik pozostaw w R0 \*/**

- Rozkazy maszynowe są na ogół instrukcjami dwuadresowymi

**a := a + 5**

**add a, 5**

# Rozkazy maszynowe

- Na ogólnie co najwyżej jeden argument rozkazu może być argumentem ulokowanym w pamięci

**a := a + b**

```
mov R0, b /* załaduj b do rejestru R0 */
add a, R0 /* dodaj zawartość R0 do a */
```

**a := b + c**

```
mov R0, b /* załaduj b do rejestru R0 */
add R0, c /* dodaj c do zawartości R0 */
mov a, R0 /* zapamiętaj zawartość R0 w a */
```

# Generacja optymalnego kodu

- Problem generacji optymalnego kodu zależnego sprzętowo jest matematycznie nierozwiązywalny bądź też praktycznie niemożliwy do rozwiązania. Dlatego też w praktyce zadowalamy się technikami heurystycznymi dającymi dobry kod, ale nie zawsze optymalny.

# Problemy:

- wybór rozkazów, minimalizacja kosztów (czasowych)

Gdybyśmy tłumaczyli każdą instrukcję trójadresową o postaci:  $x := y + z$ , gdzie  $x$ ,  $y$  i  $z$  mają statycznie przydzieloną pamięć, na następujący kod:

|                  |                                     |
|------------------|-------------------------------------|
| <b>mov R0, y</b> | <i>/* ładuj y do rejestru R0 */</i> |
| <b>add R0, z</b> | <i>/* dodaj z do R0 */</i>          |
| <b>mov x, R0</b> | <i>/* zapamiętaj R0 w x */</i>      |

# Problemy:

... to następujące instrukcje:

**a := b + c**

**d := a + e**

zostałyby przetłumaczone na:

**mov R0, b**      /\* ładuj b do rejestru R0 \*/

**add R0, c**      /\* dodaj c do R0 \*/

**mov a, R0**      /\* zapamiętaj R0 w a \*/

...



AGH

# Problemy:

... to następujące instrukcje:

**a := b + c**

**d := a + e**

zostałyby przetłumaczone na:

|                  |                              |
|------------------|------------------------------|
| <b>mov R0, b</b> | /* ładuj b do rejestru R0 */ |
| <b>add R0, c</b> | /* dodaj c do R0 */          |
| <b>mov a, R0</b> | /* zapamiętaj R0 w a */      |
| <b>mov R0, a</b> | /* ładuj a do rejestru R0 */ |
| <b>add R0, e</b> | /* dodaj e do R0 */          |
| <b>mov d, R0</b> | /* zapamiętaj R0 w d */      |



AGH

# Problemy:

... to następujące instrukcje:

**a := b + c**

**d := a + e**

zostałyby przetłumaczone na:

**mov R0, b**                   /\* ładuj b do rejestru R0 \*/

**add R0, c**                   /\* dodaj c do R0 \*/

**mov a, R0**                   /\* zapamiętaj R0 w a \*/

**mov R0, a**                   /\* ładuj a do rejestru R0 \*/ **niepotrzebne**

**add R0, e**                   /\* dodaj e do R0 \*/

**mov d, R0**                   /\* zapamiętaj R0 w d \*/



AGH

# Problemy:

... to następujące instrukcje:

**a := b + c**

**d := a + e**

zostałyby przetłumaczone na:

**mov R0, b**                   /\* ładuj b do rejestru R0 \*/

**add R0, c**                   /\* dodaj c do R0 \*/

**mov a, R0**                   /\* zapamiętaj R0 w a \*/ **niepotrzebne, jeśli a nie jest używane nigdzie dalej**

**mov R0, a**                   /\* ładuj a do rejestru R0 \*/ **niepotrzebne**

**add R0, e**                   /\* dodaj e do R0 \*/

**mov d, R0**                   /\* zapamiętaj R0 w d \*/

# Problemy:

... widać więc, że jeśli nie interesujemy się efektywnością kodu wynikowego, to wybór rozkazów jest bardzo prosty. Jednakże naiwne tłumaczenie może prowadzić do poprawnego, acz niedopuszczalnie nieefektywnego kodu wynikowego.



AGH

# Problemy:

- wybór rozkazów, minimalizacja kosztów (czasowych)  
Maszyna docelowa z bogatym zbiorem rozkazów może dostarczać wielu metod implementacji danej operacji. Ponieważ różnice kosztu między implementacjami mogą być znaczące, więc naiwne tłumaczenie może prowadzić do nieefektywnego kodu wynikowego.

Przykład: **a := a + 1**

```
mov R0, a
add R0, 1
mov a, R0
```

≡

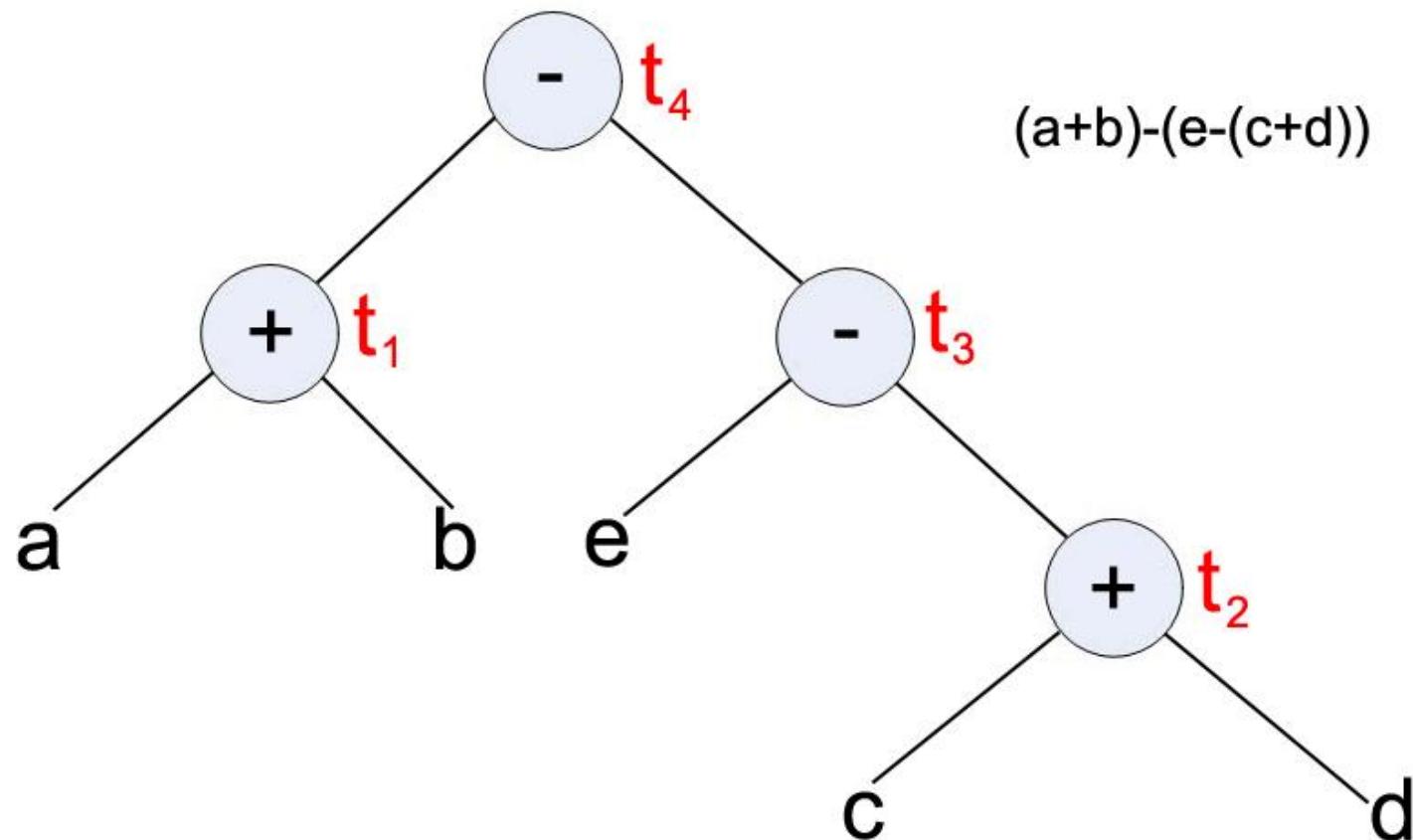
```
add a, 1
```

≡

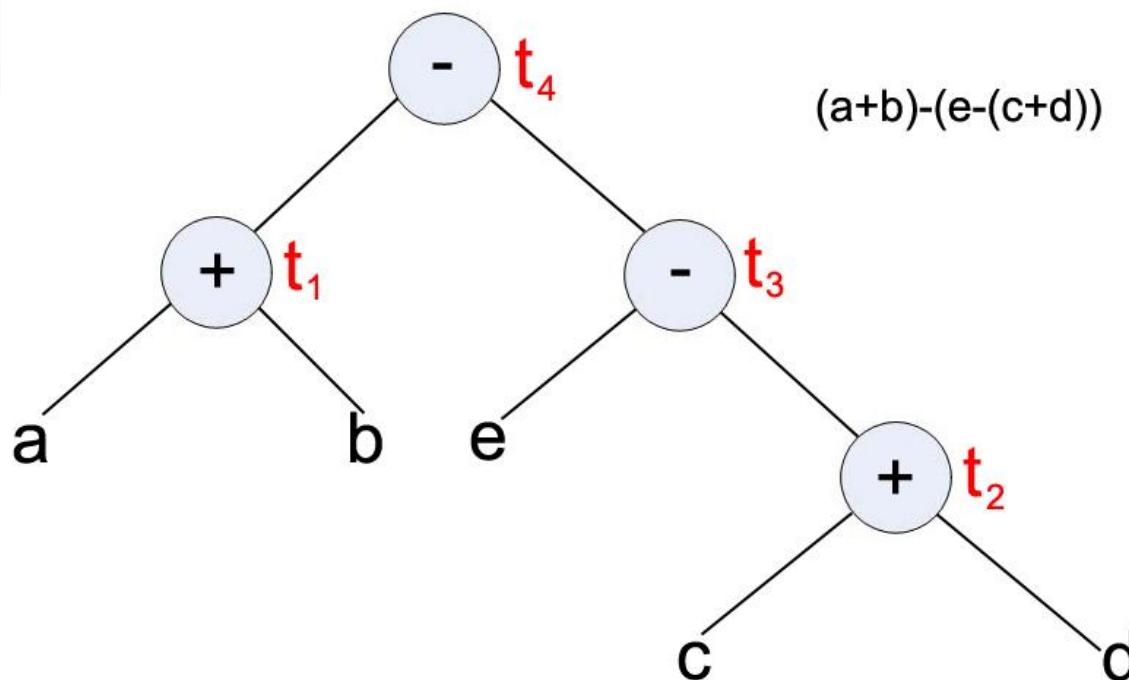
```
inc a
```

# Problemy:

- wybór kolejności obliczeń



# Wybór kolejności obliczeń



Kolejność naturalna:

$$t_1 := a + b$$

$$t_2 := c + d$$

$$t_3 := e - t_2$$

$$t_4 := t_1 - t_3$$

Kolejność zmieniona:

$$t_2 := c + d$$

$$t_3 := e - t_2$$

$$t_1 := a + b$$

$$t_4 := t_1 - t_3$$



# Wybór kolejności obliczeń

AGH

## Kolejność naturalna:

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

## Kolejność zmieniona:

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

**Założenie: dostępne są tylko dwa rejestrów: R0 i R1**

|     |                                                               |
|-----|---------------------------------------------------------------|
| MOV | R0 , a                                                        |
| ADD | R0 , b /* w R0 jest t <sub>1</sub> */                         |
| MOV | R1, c                                                         |
| ADD | R1 , d /* w R1 jest t <sub>2</sub> */                         |
| MOV | t <sub>1</sub> , R0 /* odłożenie t <sub>1</sub> do pamięci */ |
| MOV | R0 , e                                                        |
| SUB | R0 , R1 /* w R0 jest t <sub>3</sub> */                        |
| MOV | R1 , t <sub>1</sub> /* zabranie t <sub>1</sub> z pamięci */   |
| SUB | R1 , R0 /* w R1 jest t <sub>4</sub> */                        |
| MOV | t <sub>4</sub> , R1                                           |

|     |                                        |
|-----|----------------------------------------|
| MOV | R0 , c                                 |
| ADD | R0 , d /* w R0 jest t <sub>2</sub> */  |
| MOV | R1 , e                                 |
| SUB | R1 , R0 /* w R1 jest t <sub>3</sub> */ |
| MOV | R0 , a                                 |
| ADD | R0 , b /* w R0 jest t <sub>1</sub> */  |
| SUB | R0 , R1 /* w R0 jest t <sub>4</sub> */ |
| MOV | t <sub>4</sub> , R0                    |

**zaoszczędzenie dwóch rozkazów**

# Problemy

– „łatanie wstecz” dla skoków

Kod pośredni:

100: a := a+5

...

120: goto 100

Kod docelowy:

adres: add a, 5

...

jmp adres

Kod pośredni:

100: goto 120

...

120: a := a+5

Kod docelowy:

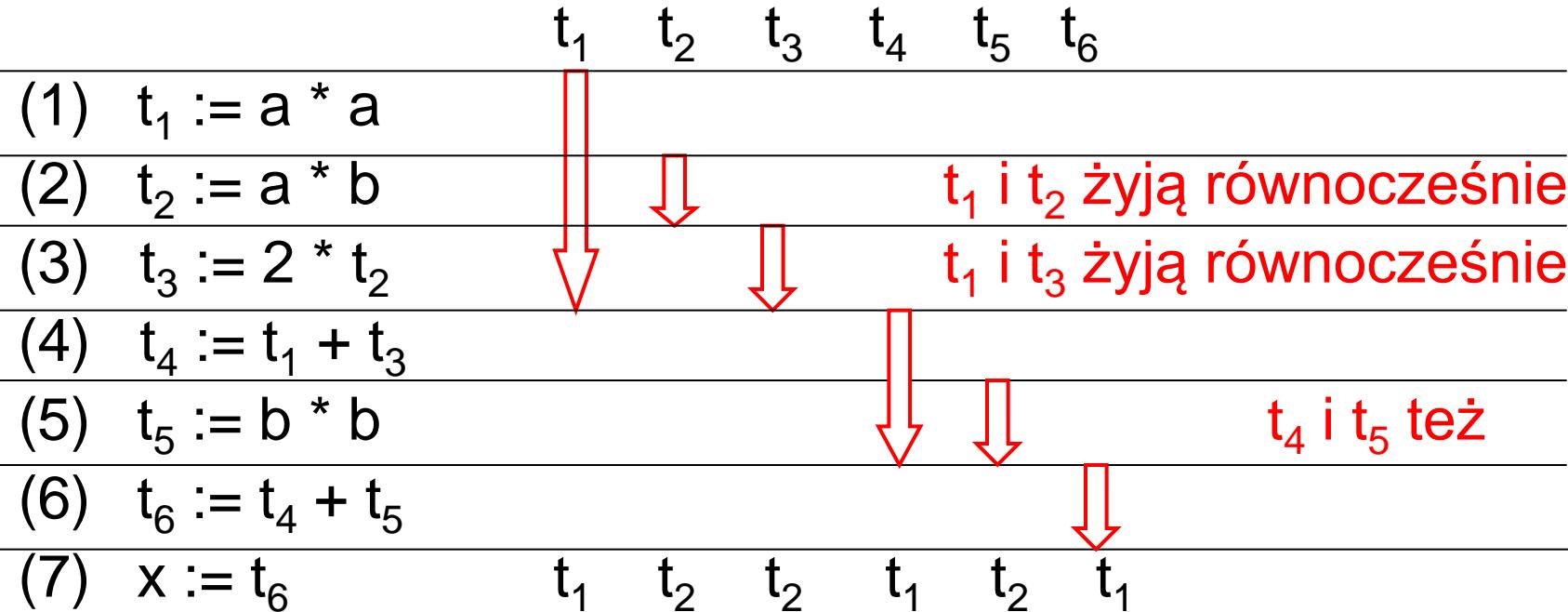
jmp

...

adres: add a, 5

**łatanie wstecz**

# "Życie" zmiennych tymczasowych



Na podstawie informacji o "życiu" można dokonać lokalizacji zmiennych tymczasowych według generalnej zasady, że dwie zmienne mogą zajmować tę samą lokalizację, gdy nie są równocześnie "żywe"

# Redukcja zmiennych tymczasowych

Przed:

- (1)  $t_1 := a * a$
- (2)  $t_2 := a * b$
- (3)  $t_3 := 2 * t_2$
- (4)  $t_4 := t_1 + t_3$
- (5)  $t_5 := b * b$
- (6)  $t_6 := t_4 + t_5$
- (7)  $x := t_6$

Po:

- (1)  $t_1 := a * a$
- (2)  $t_2 := a * b$
- (3)  $t_2 := 2 * t_2$
- (4)  $t_1 := t_1 + t_2$
- (5)  $t_2 := b * b$
- (6)  $t_1 := t_1 + t_2$
- (7)  $x := t_1$