AGH

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Generacja kodu pośredniego

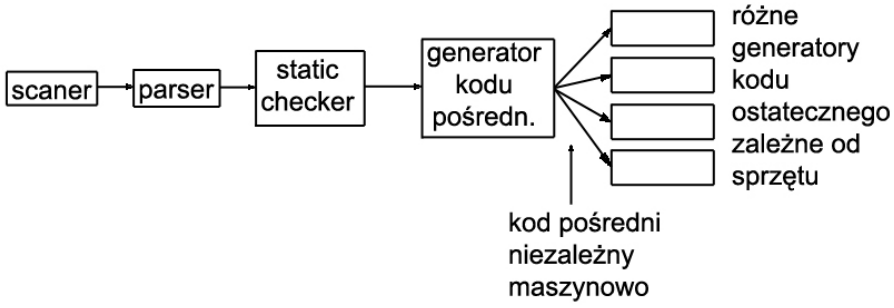
Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

AGH

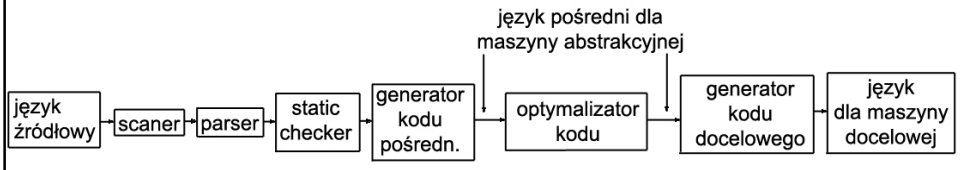
Przyczyny dwustopniowego tłumaczenia

- *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)

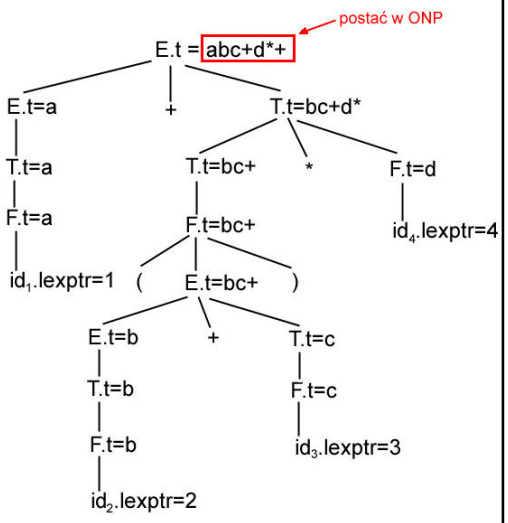
$E \rightarrow E_1 + T$	$E.t \leftarrow E_1.t \parallel T.t \parallel$ '+'
$E \rightarrow T$	$E.t \leftarrow T.t$
$T \rightarrow T_1 * F$	$T.t \leftarrow T_1.t \parallel F.t \parallel$ '*'
$T \rightarrow F$	$T.t \leftarrow F.t$
$F \rightarrow (E)$	$F.t \leftarrow E.t$
$F \rightarrow id$	$F.t \leftarrow name(id.lexptr)$

gdzie: || - operator konkatenacji tekstów

Rozważane słowo źródłowe: **a+(b+c)*d**

Po analizie leksykalnej: $id_1+(id_2+id_3)*id_4$

id.lexptr	name(id.lexptr)
1	a
2	b
3	c
4	d



Maszyna wirtualna działająca w oparciu o ONP

Przykład: maszyna wirtualna = maszyna stosowa

- źródło: **day := (1461 * y) div 4 + (153 * m + 2) div 5 + d**
- ONP: **day 1461 y * 4 div 153 m * 2 + 5 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: **day := (1461 * y) div 4 + (153 * m + 2) div 5 + d**
- ONP: **day 1461 y * 4 div 153 m * 2 + 5 div + 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

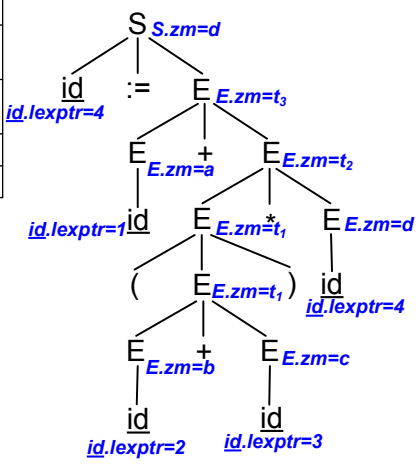
$S \rightarrow id := E$	***S.zm ← name(id.lexptr)*** ***gen(S.zm ‖ ":=" ‖ E.zm)***
$E \rightarrow E_1 + E_2$	***E.zm ← new_temp( )*** ***gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "+" ‖ E_2.zm)***
$E \rightarrow E_1 * E_2$	***E.zm ← new_temp( )*** ***gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "*" ‖ E_2.zm)***
$E \rightarrow (E_1)$	***E.zm ← E_1.zm***
$E \rightarrow id$	***E.zm ← name(id.lexptr)***

gdzie: ‖ - operator konkatenacji tekstów

Rozważane słowo źródłowe: **d:=a+(b+c)*d**

Po analizie leksykalnej: $id_4 := id_1 + (id_2 + id_3) * id_4$

id.lexptr	name(id.lexptr)
1	a
2	b
3	c
4	d



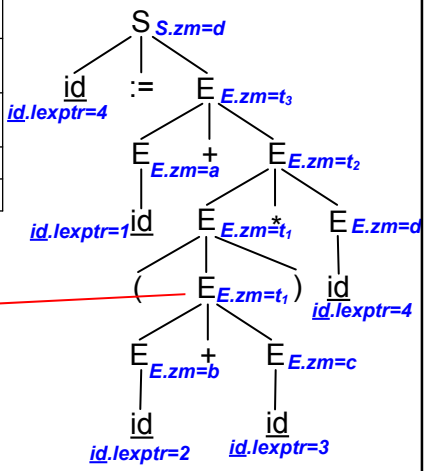
Przykład – translacja instrukcji przypisania do kodu trójadresowego

$S \rightarrow \underline{id} := E$	**S.zm ← name(id.lexptr)** **gen(S.zm ‖ ":=" ‖ E.zm)**
$E \rightarrow E_1 + E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "+" ‖ E_2.zm)**
$E \rightarrow E_1 * E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "*" ‖ E_2.zm)**
$E \rightarrow (E_1)$	**E.zm ← E_1.zm**
$E \rightarrow \underline{id}$	**E.zm ← name(id.lexptr)**

słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

$t_1 := b + c$



Przykład – translacja instrukcji przypisania do kodu trójadresowego

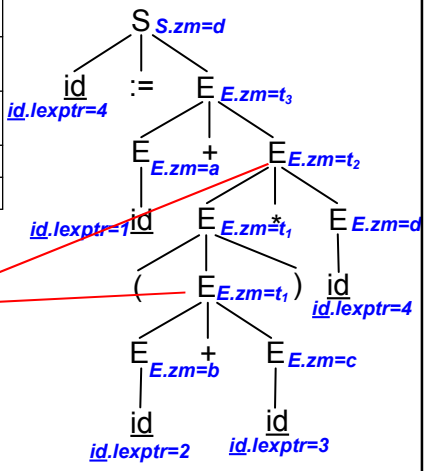
$S \rightarrow \underline{id} := E$	**S.zm ← name(id.lexptr)** **gen(S.zm ‖ ":=" ‖ E.zm)**
$E \rightarrow E_1 + E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "+" ‖ E_2.zm)**
$E \rightarrow E_1 * E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "*" ‖ E_2.zm)**
$E \rightarrow (E_1)$	**E.zm ← E_1.zm**
$E \rightarrow \underline{id}$	**E.zm ← 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

$S \rightarrow \underline{id} := E$	**S.zm ← name(id.lexptr)** **gen(S.zm ‖ ":=" ‖ E.zm)**
$E \rightarrow E_1 + E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "+" ‖ E_2.zm)**
$E \rightarrow E_1 * E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "*" ‖ E_2.zm)**
$E \rightarrow (E_1)$	**E.zm ← E_1.zm**
$E \rightarrow \underline{id}$	**E.zm ← name(id.lexptr)**

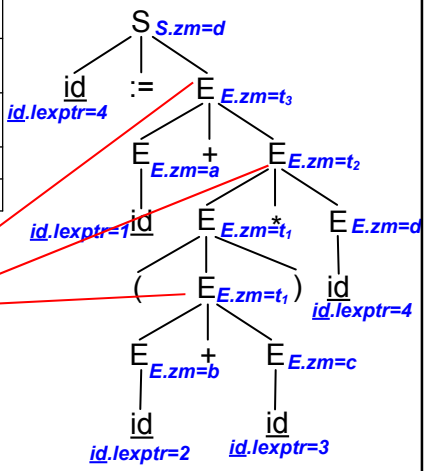
słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

$t_1 := b + c$

$t_2 := t_1 * d$

$t_3 := a + t_2$



Przykład – translacja instrukcji przypisania do kodu trójadresowego

$S \rightarrow \underline{id} := E$	**S.zm ← name(id.lexptr)** **gen(S.zm ‖ ":=" ‖ E.zm)**
$E \rightarrow E_1 + E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "+" ‖ E_2.zm)**
$E \rightarrow E_1 * E_2$	**E.zm ← new_temp( )** **gen(E.zm ‖ ":=" ‖ E_1.zm ‖ "*" ‖ E_2.zm)**
$E \rightarrow (E_1)$	**E.zm ← E_1.zm**
$E \rightarrow \underline{id}$	**E.zm ← name(id.lexptr)**

słowo źródłowe: **d:=a+(b+c)*d**

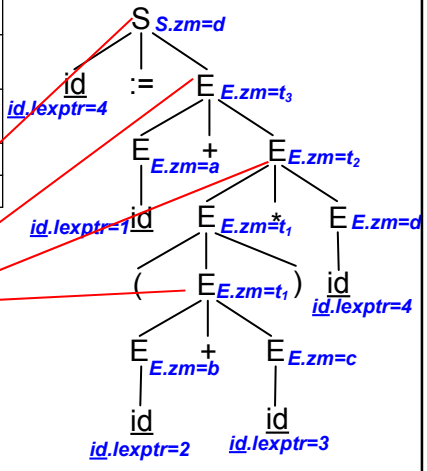
Tłumaczenie:

$t_1 := b + c$

$t_2 := t_1 * d$

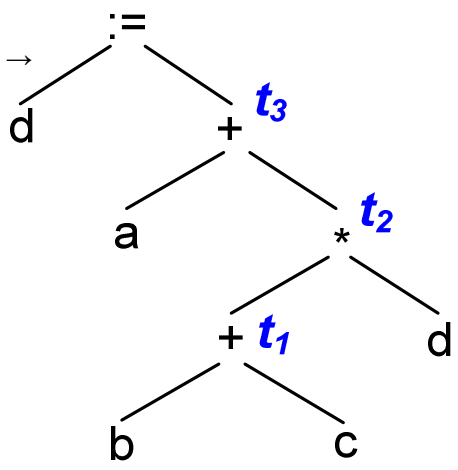
$t_3 := a + t_2$

$d := t_3$



Przykład – translacja instrukcji przypisania do kodu trójadresowego

Drzewo syntaktyczne →



Słowo źródłowe: **d:=a+(b+c)*d**

Tłumaczenie:

$t_1 := b + c$

$t_2 := t_1 * d$

$t_3 := a + t_2$

$d := t_3$

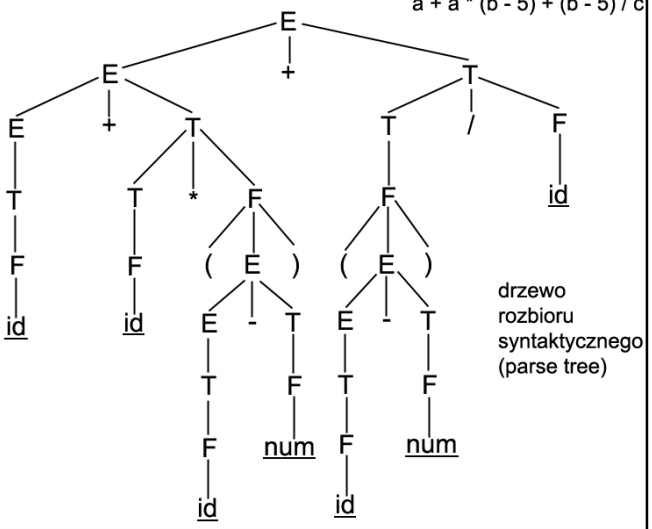
Przypomnienie: drzewa rozbioru

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

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

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

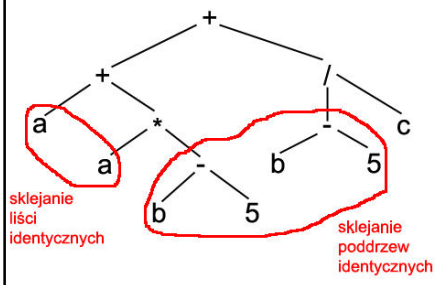
a + a * (b - 5) + (b - 5) / c



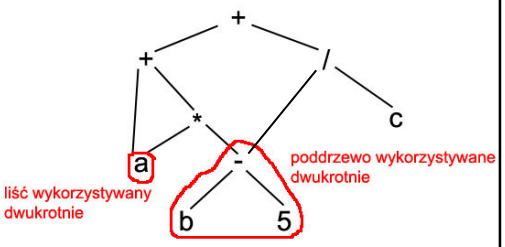
drzewo rozbioru syntaktycznego (parse tree)

Drzewa syntaktyczne i dagi

a + a * (b - 5)+ (b - 5) / c



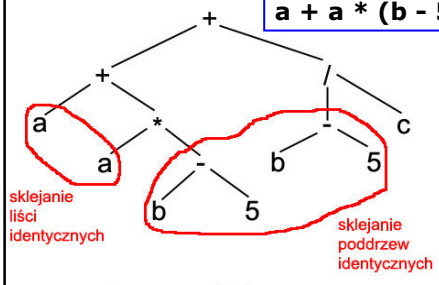
drzewo syntaktyczne
(syntax tree)



skierowany graf acykliczny
(directed acyclic graph - dag)

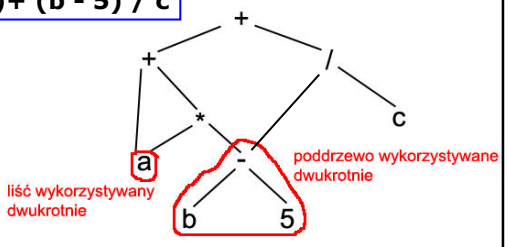
Drzewa syntaktyczne i dagi a kod trójadresowy

a + a * (b - 5)+ (b - 5) / c



drzewo syntaktyczne
(syntax tree)

$t_1 := b - 5$
$t_2 := a * t_1$
$t_3 := a + t_2$
$t_4 := b - 5$
$t_5 := t_4 / c$
$t_6 := t_3 + t_5$



skierowany graf acykliczny
(directed acyclic graph - dag)

$t_1 := b - 5$
$t_2 := a * t_1$
$t_3 := a + t_2$
$t_4 := t_1 / c$
$t_5 := t_3 + t_4$

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)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Zmienne tymczasowe muszą być w tablicy symboli

pointery do tabl. symboli

Trójki:

	instr
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Reprezentacja pośrednia umożliwiająca przestawianie instrukcji

	oper	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(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.nptr ← mknode('+',E_1.nptr,T.nptr)}
$E \rightarrow E_1 - T$	{E.nptr ← mknode('-',E_1.nptr,T.nptr)}
$E \rightarrow T$	{E.nptr ← T.nptr}
$T \rightarrow T_1 * F$	{T.nptr ← mknode('*',T_1.nptr, F.nptr)}
$T \rightarrow T_1 / F$	{T.nptr ← mknode('/',T_1.nptr,F.nptr)}
$T \rightarrow F$	{T.nptr ← F.nptr}
$F \rightarrow (E)$	{F.nptr ← E.nptr}
$F \rightarrow$ id	{F.nptr ← mkleaf(id,id.lexptr)}
$F \rightarrow$ num	{F.nptr ← mkleaf(num,num.val)}

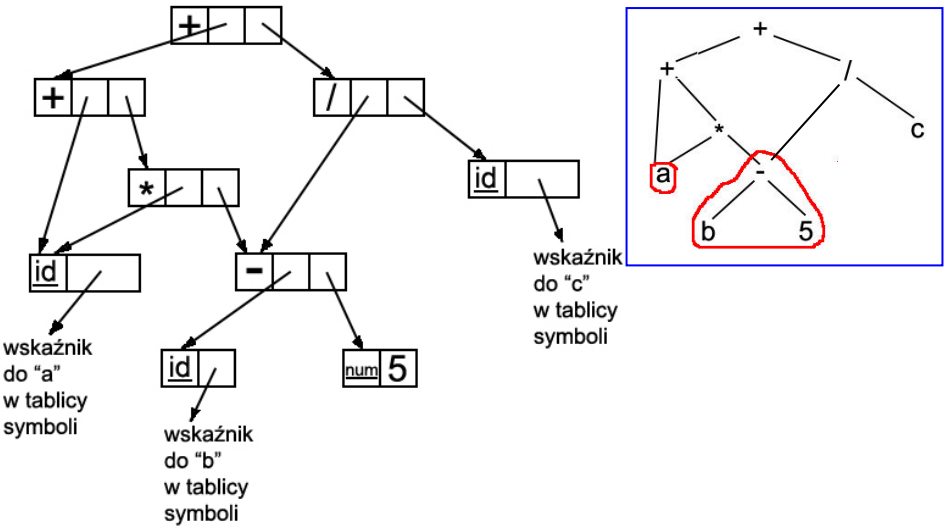
Konstruowanie drzew syntaktycznych lub skierowanych grafów acyklicznych dla wyrażeń

(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

Konstruowanie drzew syntaktycznych lub skierowanych grafów acyklicznych dla wyrażeń

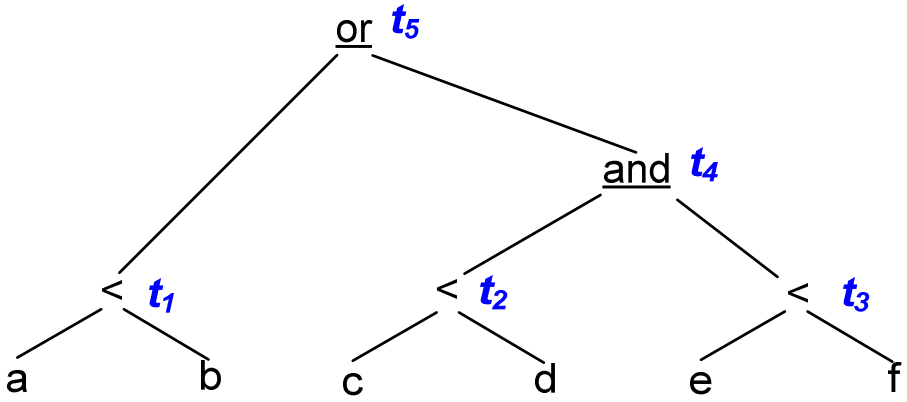


Translacja wyrażeń logicznych „do końca”

$E \rightarrow E_1$ or E_2	E.place ← newtemp gen(E.place ‘:=’ E_1.place ‘or’ E_2.place)
$E \rightarrow E_1$ and E_2	E.place ← newtemp gen(E.place ‘:=’ E_1.place ‘and’ E_2.place)
$E \rightarrow$ not E_1	E.place ← newtemp gen(E.place ‘:=’ ‘not’ E_1.place)
$E \rightarrow id_1$ relop id_2	E.place ← newtemp gen(‘if’ id_1.place relop.op id_2.place ‘goto’ nexstat + 3) gen(E.place ‘:=’ ‘0’) gen(‘goto’ nextstat + 2) gen(E.place ‘:=’ ‘1’)
$E \rightarrow$ true	E.place ← newtemp; gen(E.place ‘:=’ ‘1’)
$E \rightarrow$ false	E.place ← newtemp; gen(E.place ‘:=’ ‘0’)
$E \rightarrow (E_1)$	E.place ← E_1.place

Translacja wyrażeń logicznych „do końca”

Przykład: **a < b or c < d and e < f**



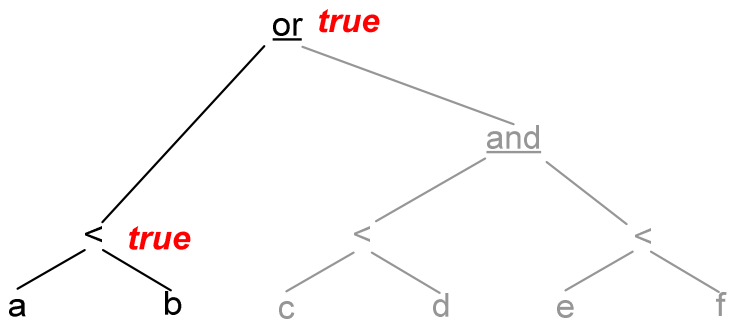
Translacja wyrażeń logicznych „do końca”

Przykład: **a < b or c < d and e < f**

100 : if a < b goto 103
101 : t_1 := 0
102 : goto 104
103 : t_1 := 1
104 : if c < d goto 107
105 : t_2 := 0
106 : goto 108
107 : t_2 :=1
108 : if e < f goto 111
109 : t_3 := 0
110 : goto 112
111 : t_3 := 1
112 : t_4 := t_2 and t_3
113 : t_5 := t_1 or t_4

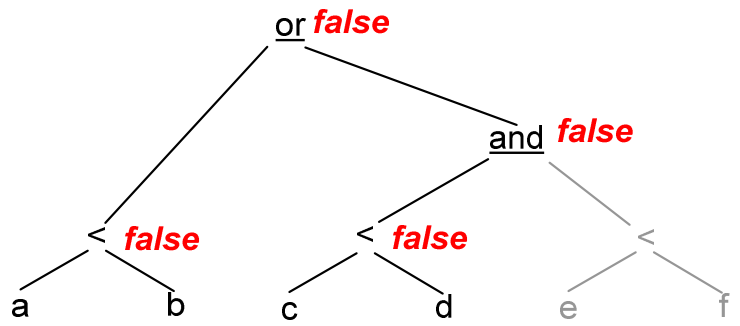
Translacja wyrażeń logicznych („short – circuit” code)

Przykład: **a < b or c < d and e < f**



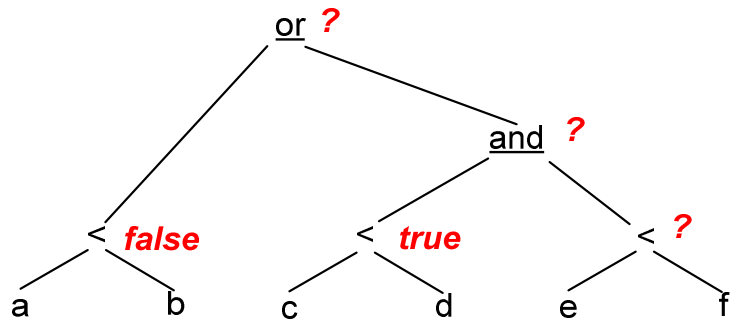
Translacja wyrażeń logicznych („short – circuit” code)

Przykład: **a < b or c < d and e < f**

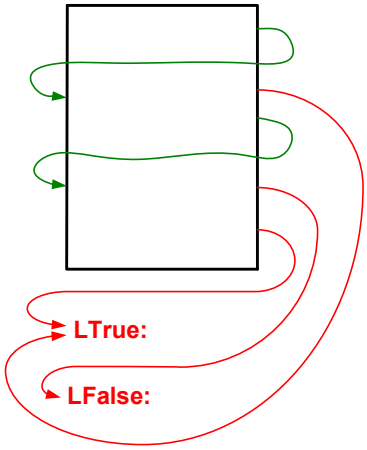


Translacja wyrażeń logicznych („short – circuit” code)

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 $E.false$

atrybut dziedziczony symbolu E = etykieta, do której przechodzi sterowanie, gdy $a < b$ jest *„true”*

Jeśli E jest postaci E_1 *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”*

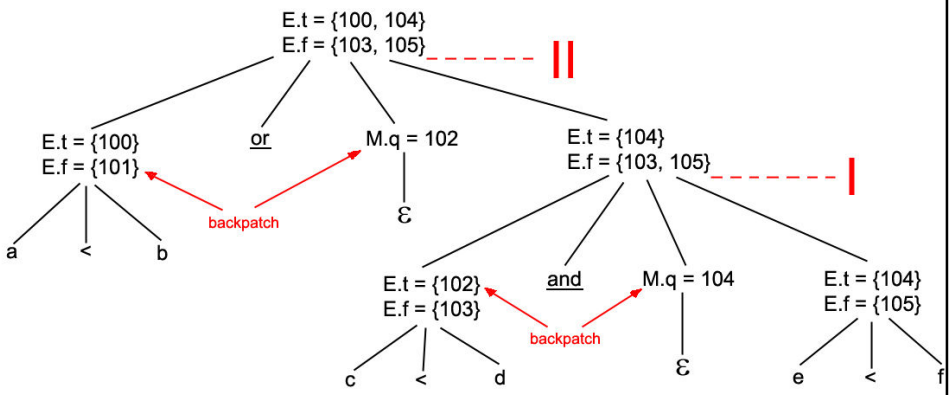
$E_2.true \leftarrow E.true$

$E_2.false \leftarrow E.false$

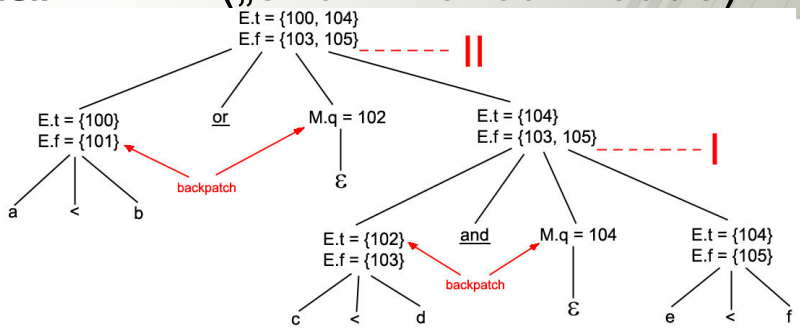
Translacja wyrażeń logicznych („short – circuit” code)

$E \rightarrow E_1$ or $M\ E_2$	backpatch(E_1.falselist, M.quad); E.truelist ← merge(E_1.truelist,E_2.truelist); E.falselist ← E_2.falselist;
$E \rightarrow E_1$ and $M\ E_2$	backpatch(E_1.truelist , M.quad); E.truelist ← E_2.truelist; E.falselist ← merge(E_1.falselist,E_2.falselist);
$E \rightarrow$ not E_1	E.truelist ← E_1.falselist; E.falselist ← E_1.truelist;
$E \rightarrow (E_1)$	E.truelist ← E_1.truelist; E.falselist ← E_1.falselist;
$E \rightarrow id_1$ relop id_2	E.truelist ← makelist(nextquad()) E.falselist ← makelist(nextquad() + 1) gen(‘if’ id_1.place relop.op id_2.place ‘goto _’); gen(‘goto _ ‘);
$E \rightarrow$ true	E.truelist ← makelist(nextquad()); gen(‘goto _ ‘);
$E \rightarrow$ false	E.falselist ← makelist (nextquad()); gen (‘goto _’);
$M \rightarrow \varepsilon$	M.quad ← 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 _

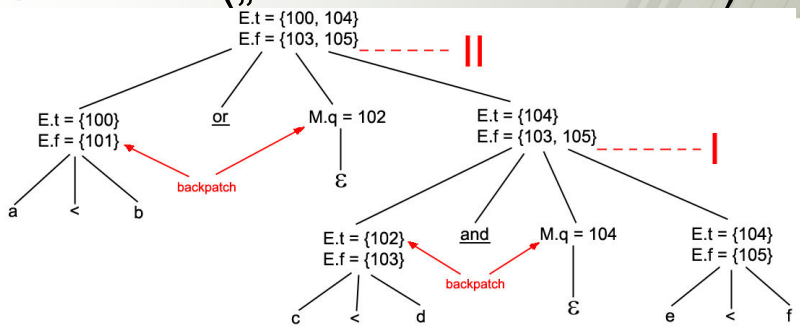
Po redukcji (I)
Przed redukcją (II)

100: if a < b goto _
101: goto _
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
```