

Techniki kompilacji

Grzegorz Jabłoński

Katedra Mikroelektroniki i Techniki
Informatycznych

tel. (631) 26-48

gwj@dmcs.p.lodz.pl

<http://neo.dmcs.p.lodz.pl/tk>

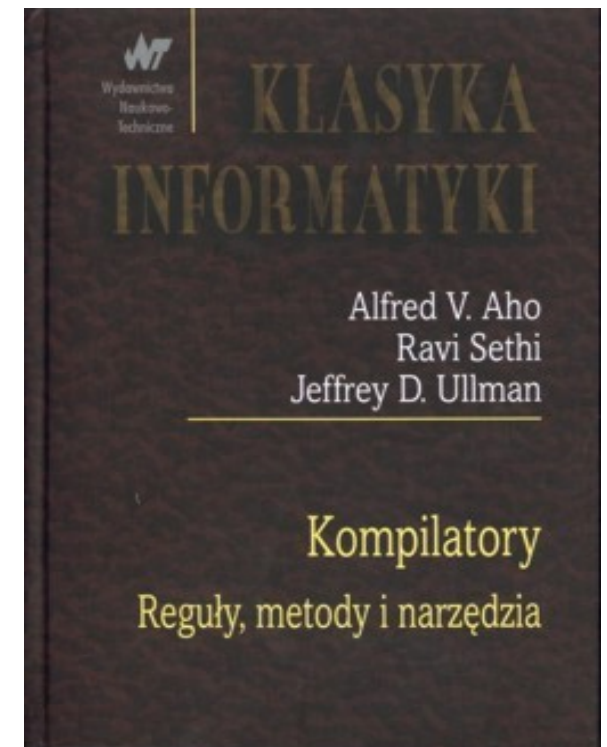
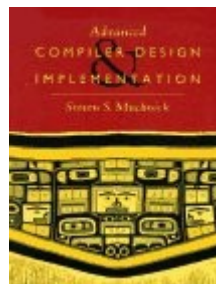
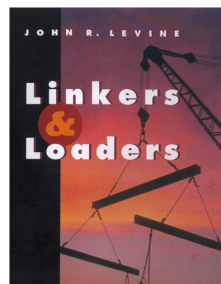
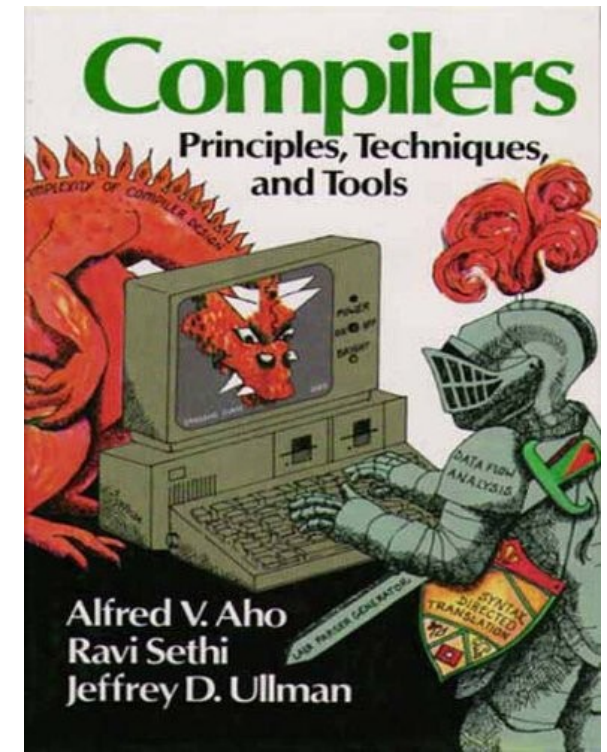
Literatura

- Podstawowa

- A.V. Aho, R. Sethi, J. D. Ullman, "Compilers - Principles, Techniques, and Tools", Addison-Wesley 1986 (polskie wydanie WNT 2002)

- Uzupełniająca

- John R. Levine, "Linkers and Loaders", Morgan Kaufmann Publishers 1999 (manuskrypt dostępny online)
- W. M. Waite, G. Goos, "Konstrukcja kompilatorów", WNT 1989
- S. S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers 1997



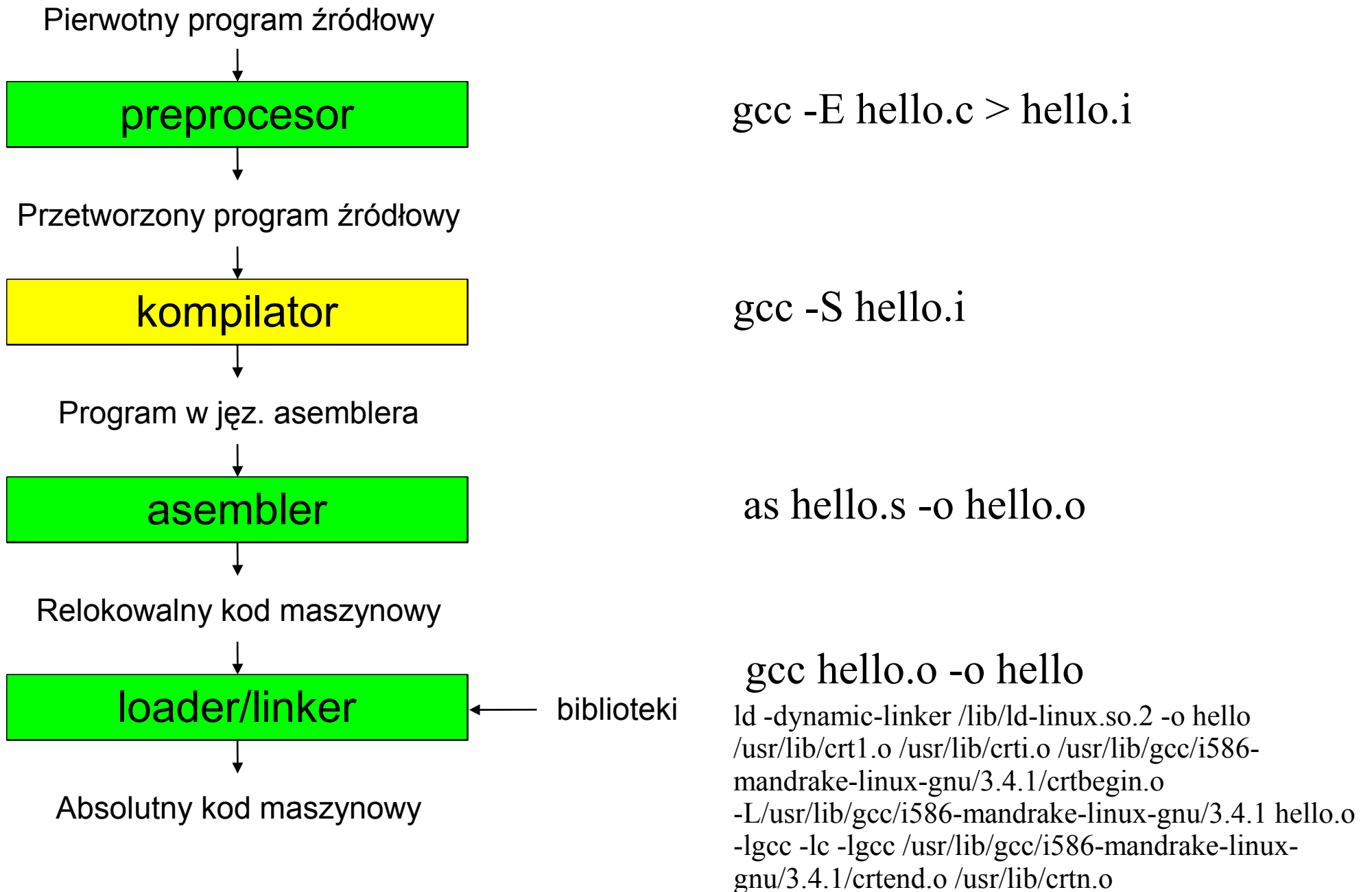
Zakres przedmiotu

- Budowa i działanie kompilatorów języków programowania
 - Definicje języków programowania
 - Poszczególne etapy kompilacji
 - Projekt prostego kompilatora dla języka proceduralnego

Napisaliśmy program, jak go uruchomić?

- Kompilator
 - Przetwarza cały program na postać zrozumiałą dla procesora
- Interpreter
 - Przetwarza osobno każdą pojedynczą instrukcję
- Rozwiązania pośrednie
 - Kod pośredni, interpretowany przez procesor wirtualny
 - Szczególny przypadek: kompilator JIT (Just-in-Time)

Przetwarzanie programu



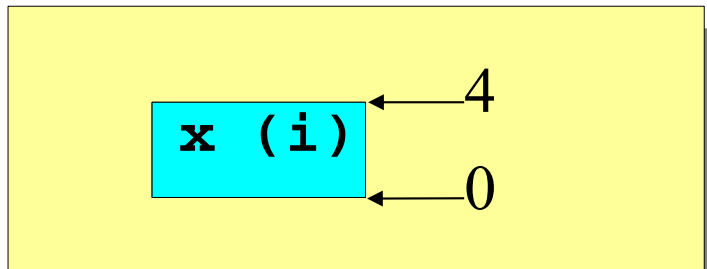
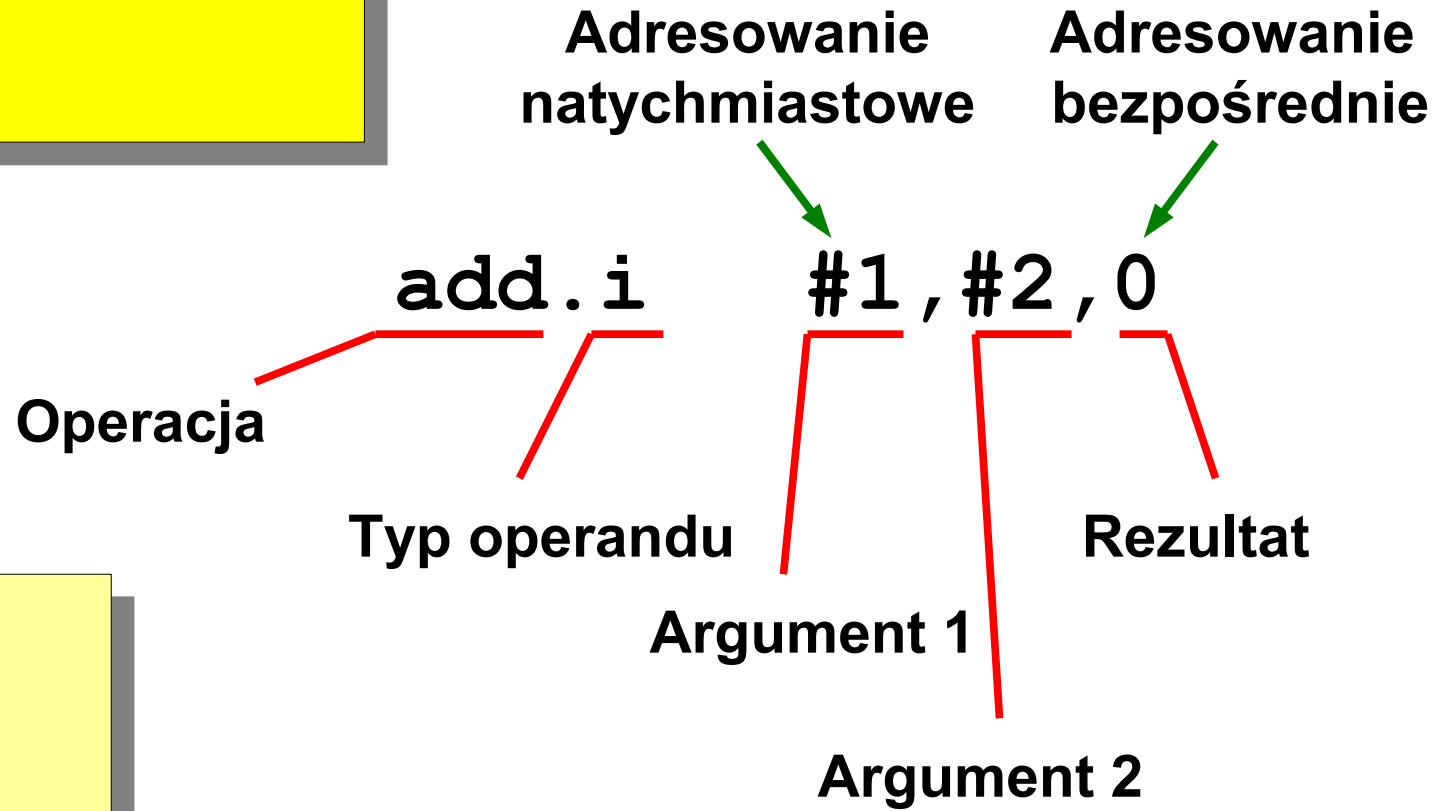
Projekt kompilatora

- Kompilator języka o składni zbliżonej do języka Pascal
- Procesor docelowy
 - Architektura harwardzka
 - Architektura pamięć-pamięć
 - Kompilacja do postaci asemblera symbolicznego
 - z adresami bezwzględnymi zmiennych
 - z adresami symbolicznymi rozkazów
 - Interpreter asemblera częściowo zaimplementowany
- Dostępna implementacja wzorcowa kompilatora – bez źródeł

Najprostszy program

```
program example1(input, output);  
var x: integer;  
  
begin  
  x:=1+2;  
  write(x)  
end.
```

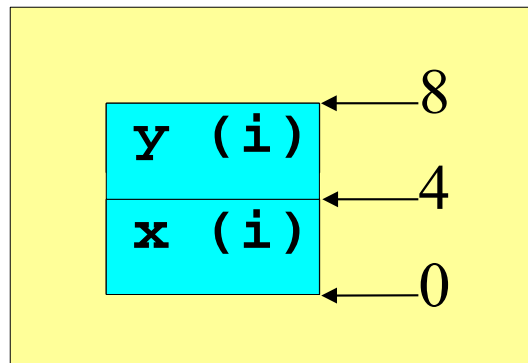
```
add.i    #1,#2,0  
write.i  0  
exit
```



Dwie zmienne

```
program example2(input, output);  
var x,y: integer;  
  
begin  
  x:=1+2;  
  y:=x+1;  
  write(y)  
end.
```

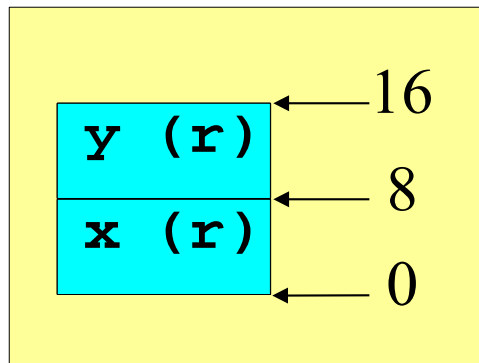
```
add.i    #1,#2,0  
add.i    0,#1,4  
write.i  4  
exit
```



Zmienne rzeczywiste

```
program example3(input, output);  
var x,y: real;  
  
begin  
  x:=1.0;  
  y:=x+2.0;  
  write(y)  
end.
```

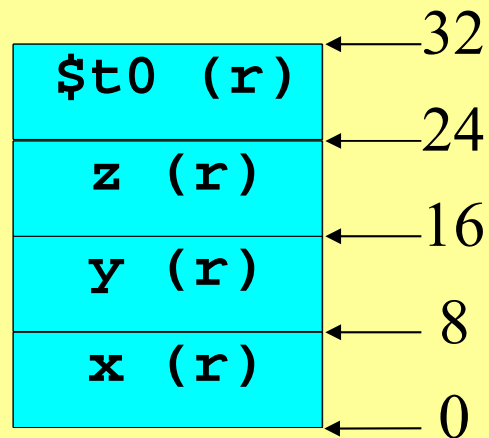
```
mov.r    #1,0  
add.r    0,#2,8  
write.r  8  
exit
```



Wyrażenie złożone

```
program example4(input, output);  
var x,y,z: real;  
  
begin  
  x:=1.0;  
  y:=2.5;  
  z:=x+2.0*y;  
  write(z)  
end.
```

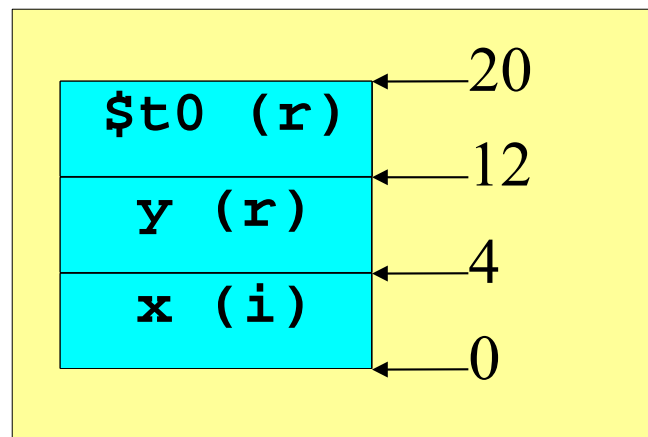
```
mov.r    #1,0  
mov.r    #2.5,8  
mul.r    #2,8,24  
add.r    0,24,16  
write.r  16  
exit
```



Wyrażenia mieszane

```
program example5(input, output);  
var x: integer;  
var y: real;  
  
begin  
  x:=1;  
  y:=x+2.0;  
  write(y)  
end.
```

```
mov.i      #1,0  
inttoreal 0,12  
add.r      12,#2,4  
write.r    4  
exit
```

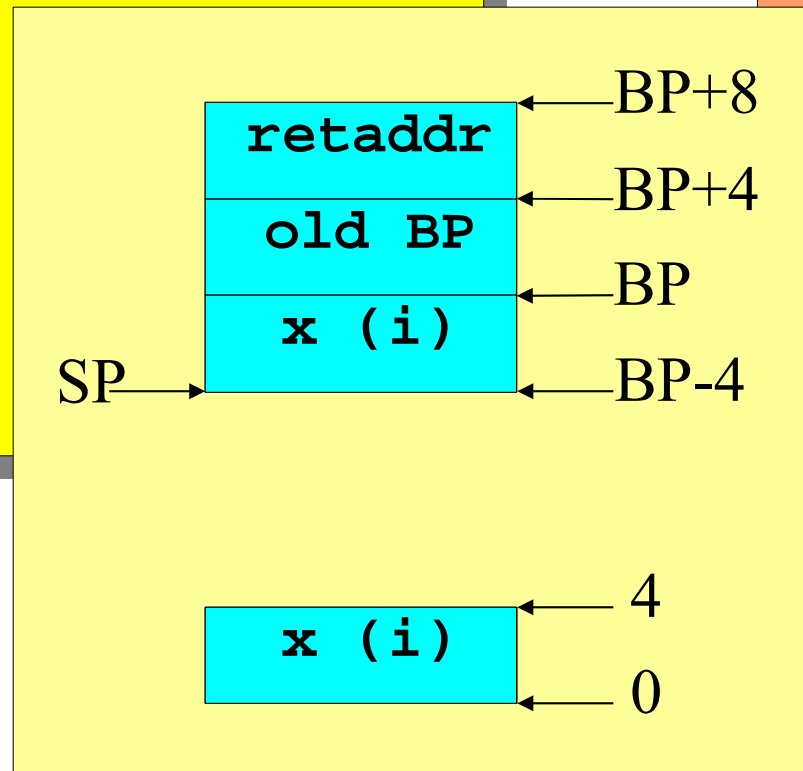


Procedury i zmienne lokalne

```
program example6(input, output);
var x: integer;
```

```
procedure p;
var x: integer;
begin
  x:=0;
  write(x)
end;
```

```
begin
  x:=1;
  p;
  write(x)
end.
```



```
      jump.i   #lab0
p:
      enter.i  #4
      mov.i   #0, BP-4
      write.i BP-4
      leave
      return
lab0:
      mov.i   #1, 0
      call.i  #p
      write.i 0
      exit
```

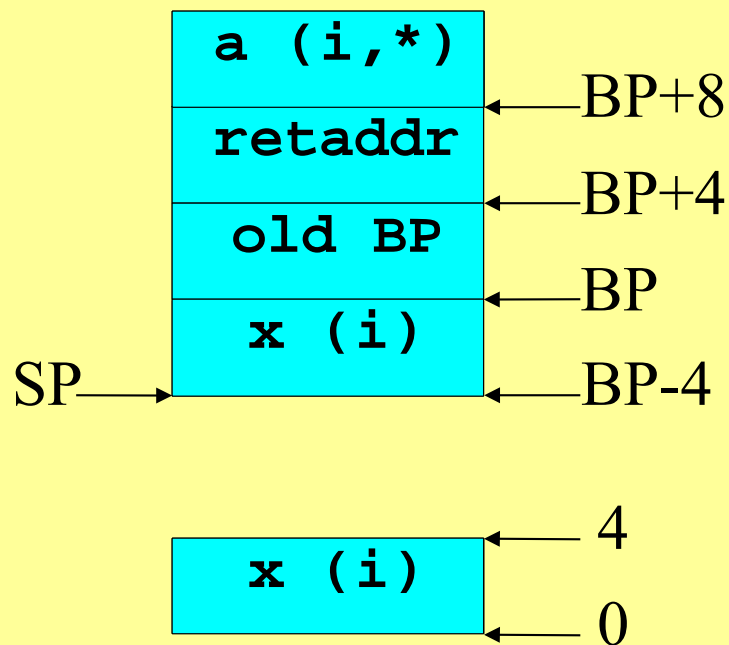
```
enter.i n
  push.i #BP
  mov.i  #SP, #BP
  sub.i  #SP, #n, #SP
leave
  mov.i  #BP, #SP
  pop.i  #BP
```

Procedury i parametry

```
program example7(input, output);  
var x: integer;  
  
procedure p(a:integer);  
var x: integer;  
begin  
  x:=a+1;  
  write(x)  
end;
```

```
begin  
  x:=1;  
  p(x);  
  write(x)  
end.
```

```
jump.i #lab0  
  
p:  
  enter.i #4  
  add.i *BP+8,#1,BP-4  
  write.i BP-4  
  leave  
  return  
  
lab0:  
  mov.i #1,0  
  push.i #0  
  call.i #p  
  incsp.i #4  
  write.i 0  
  exit
```

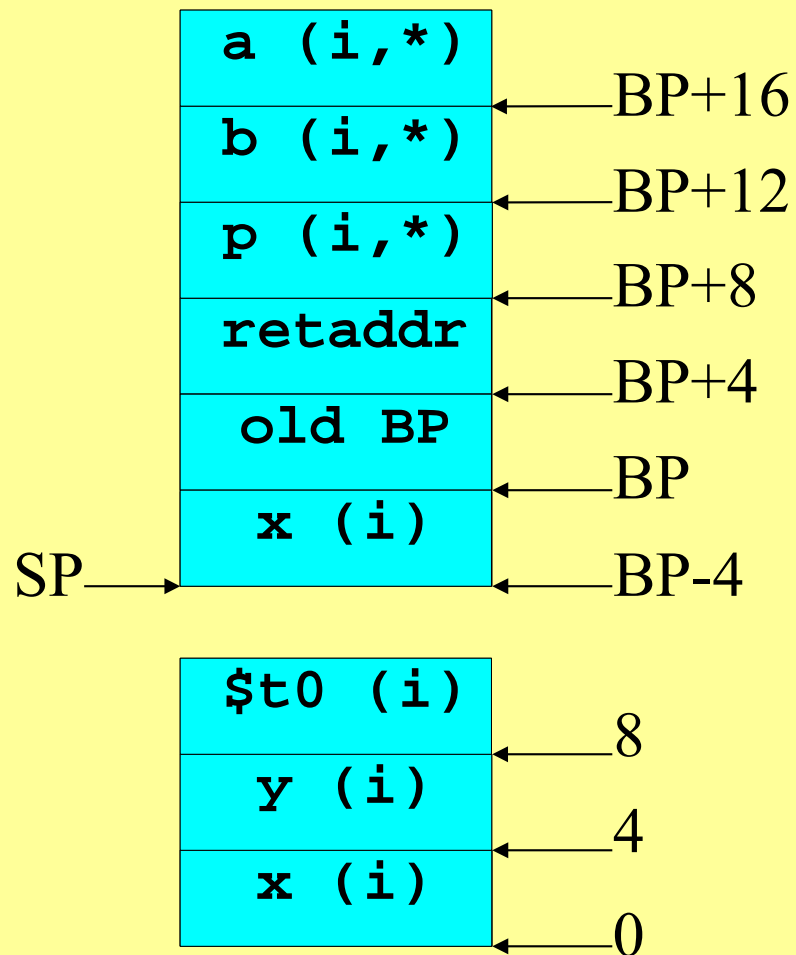


Funkcje

```
program example8(input, output);  
var x,y: integer;  
function p(a,b:integer): integer;  
var x: integer;
```

```
begin  
  x:=a+1;  
  p:=x+b  
end;
```

```
begin  
  x:=1;  
  y:=2;  
  x:=p(x,y);  
  write(x)  
end.
```

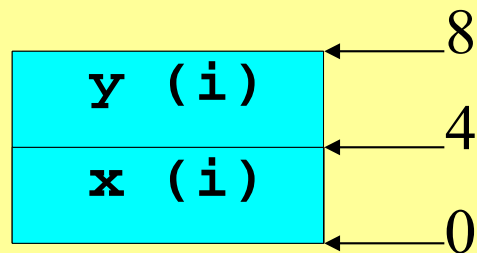


```
jump.i #lab0  
p:  
  enter.i #4  
  add.i *BP+16,#1,BP-4  
  add.i BP-4,*BP+12,*BP+8  
  leave  
  return  
lab0:  
  mov.i #1,0  
  mov.i #2,4  
  push.i #0  
  push.i #4  
  push.i #8  
  call.i #p  
  incsp.i #12  
  mov.i 8,0  
  write.i 0  
  exit
```

Instrukcje warunkowe

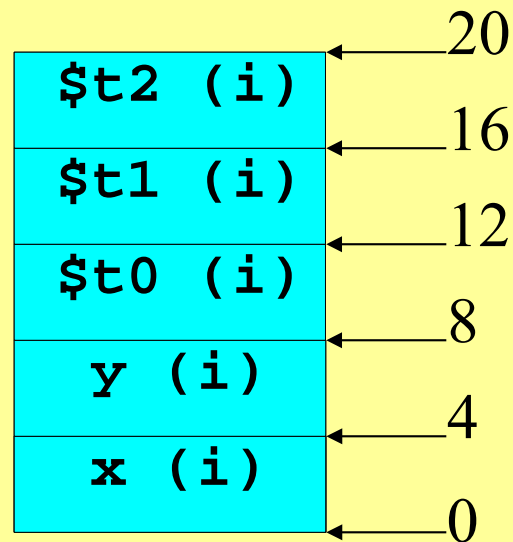
```
program example9(input, output);  
var x,y: integer;  
  
begin  
  read (x,y);  
  if x>y then  
    write (x)  
  else  
    write (y)  
end.
```

```
read.i 0  
read.i 4  
jg.i 0,4,#then  
write.i 4  
jump.i #endif  
then:  
write.i 0  
endif:  
exit
```



Złożone instrukcje warunkowe

```
program example10(input, output);  
var x,y: integer;  
  
begin  
  read (x,y);  
  if (x>y) and (x>0) then  
    write (x)  
  else  
    write (y)  
end.
```

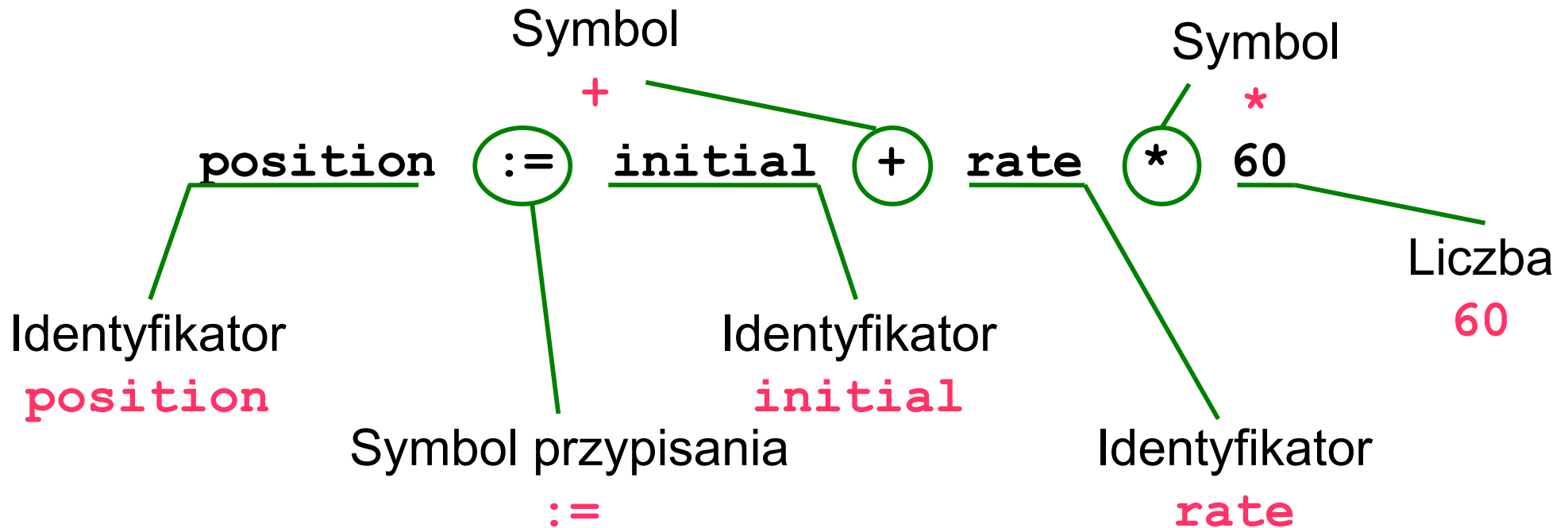


```
read.i 0  
read.i 4  
jg.i 0,4,#lab1  
mov.i #0,8  
jump.i #lab2  
lab1: mov.i #1,8  
lab2: jg.i 0,#0,#lab3  
mov.i #0,12  
jump.i #lab4  
lab3: mov.i #1,12  
lab4: and.i 8,12,16  
je.i 16,#0,#lab5  
write.i 0  
jump.i #lab6  
lab5: write.i 4  
lab6: exit
```

Etapy kompilacji



Analiza leksykalna

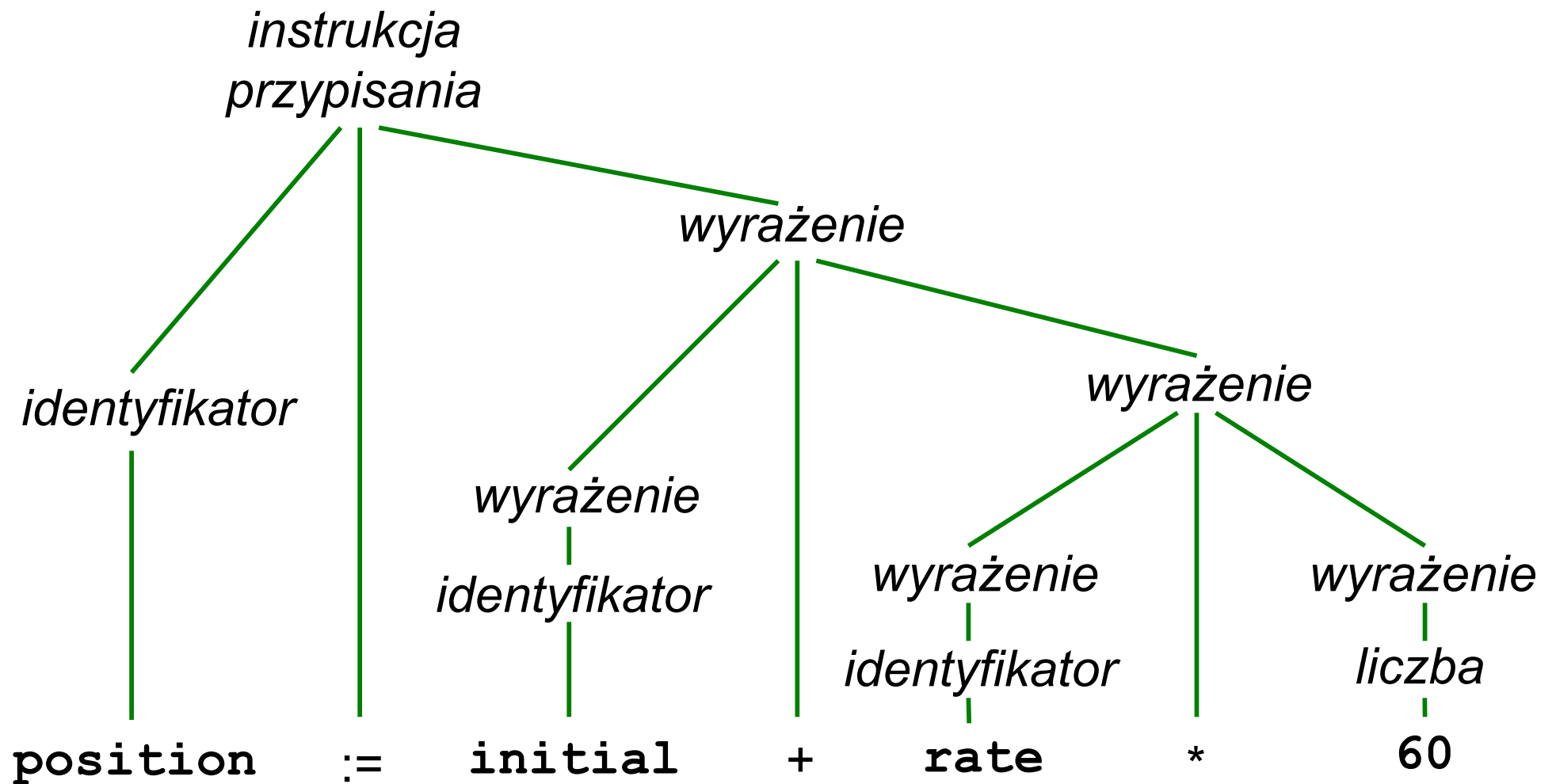


Tablica symboli

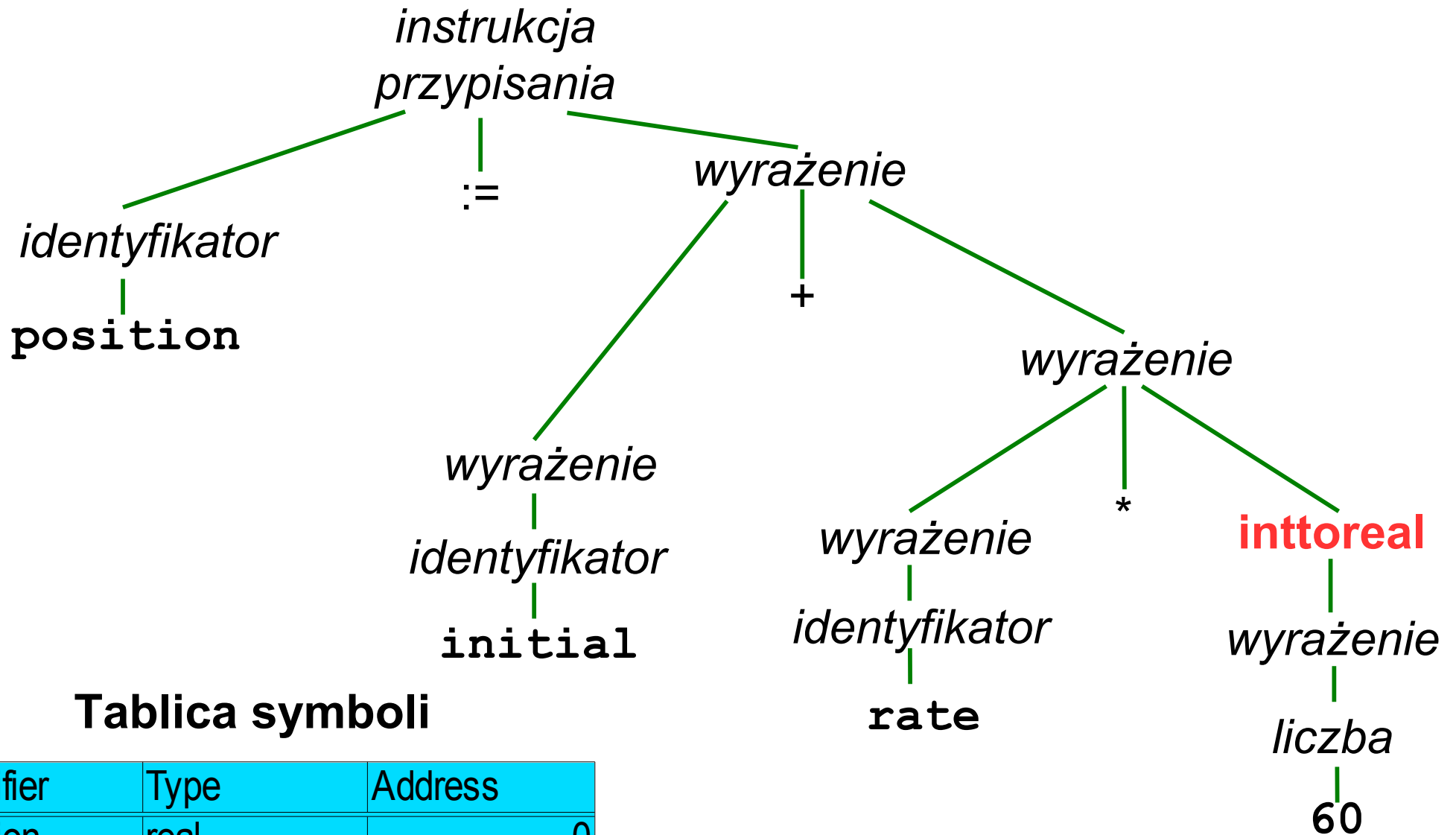
Identifier	Type	Address
position	real	0
initial	real	8
rate	real	16

Analiza składniowa

```
instrukcja przypisania -> identyfikator ' := ' wyrażenie
wyrażenie -> liczba
wyrażenie -> identyfikator
wyrażenie -> wyrażenie '+' wyrażenie
wyrażenie -> wyrażenie '*' wyrażenie
```



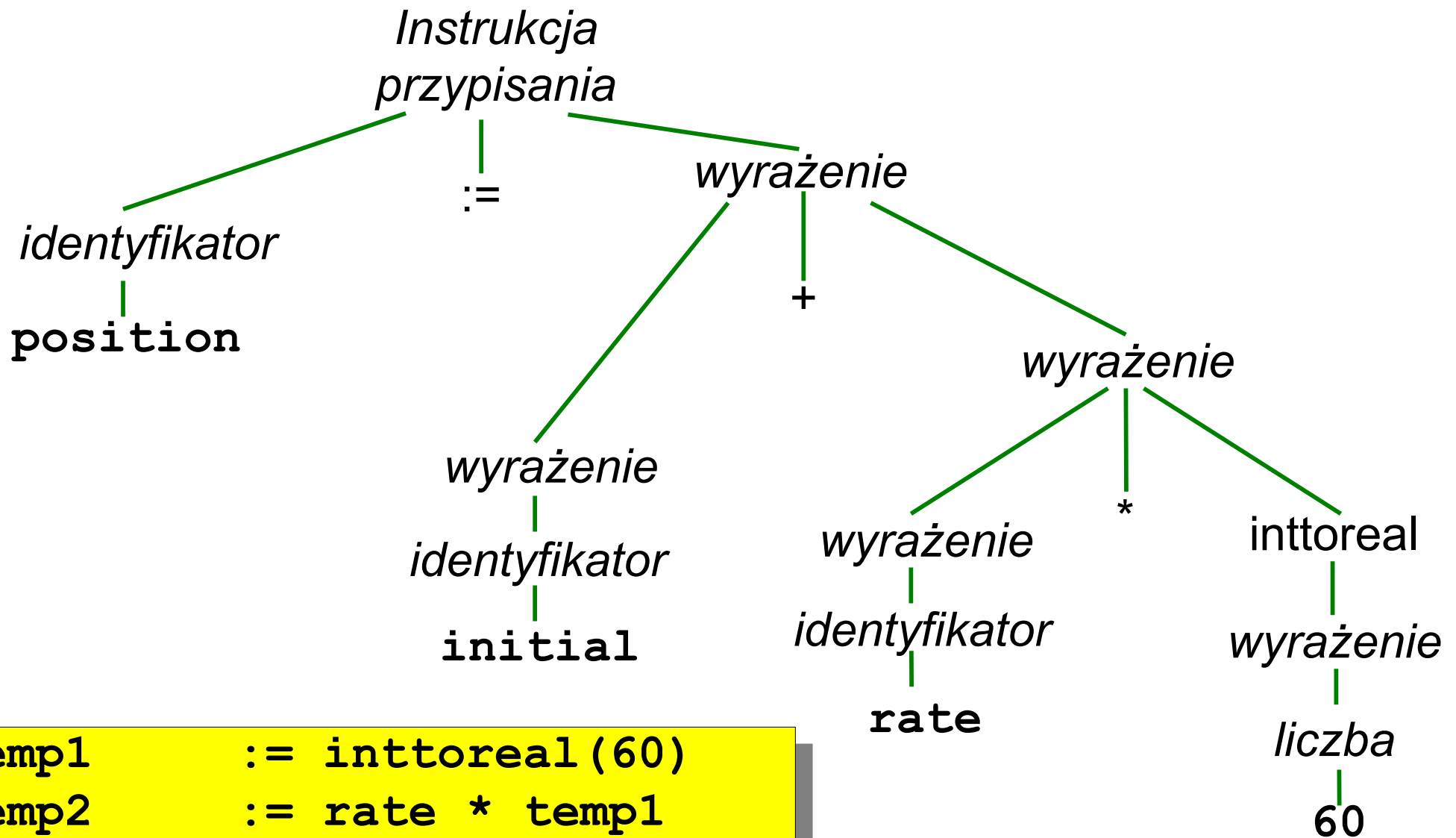
Analiza semantyczna



Tablica symboli

Identifer	Type	Address
position	real	0
initial	real	8
rate	real	16

Generacja kodu pośredniego



```
temp1    := inttoreal(60)
temp2    := rate * temp1
temp3    := initial + temp2
position := temp3
```

Optymalizacja i generacja kodu

```
temp1    := inttoreal(60)
temp2    := rate * temp1
temp3    := initial + temp2
position := temp3
```

```
temp1    := rate * 60.0
position := initial + temp1
```

Tablica symboli

Identifier	Type	Address
position	real	0
initial	real	8
rate	real	16
temp1	real	24

```
MOVF 16, R2
MULF #60.0, R2
MOVF 8, R1
ADDF R2, R1
MOVF R1, 0
```

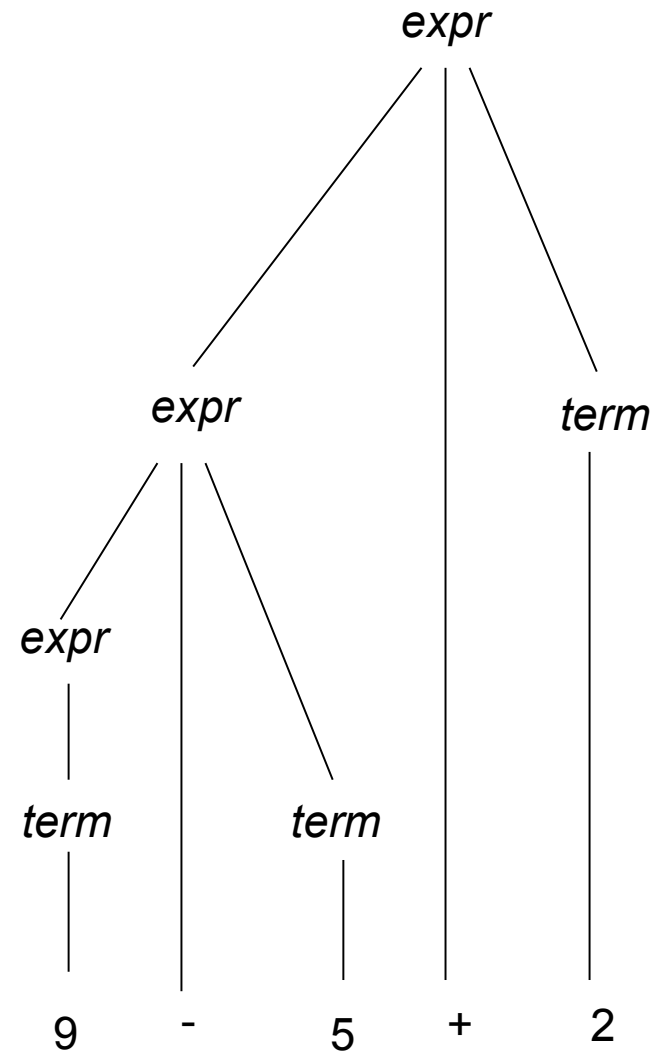
Formalna gramatyka bezkontekstowa

- Zbiór tokenów zwanych symbolami terminalnymi
- Zbiór symboli nieterminalnych
- Zbiór produkcji, z których każda składa się z symbolu nieterminalnego, zwanego lewą stroną produkcji, strzałki oraz sekwencji tokenów i symboli nieterminalnych, zwanej prawą stroną produkcji
- Jednego wyróżnionego symbolu nieterminalnego, zwanego symbolem startowym

Drzewa wyprowadzeń

- Wyrażenia arytmetyczne z operacjami dodawania i odejmowania

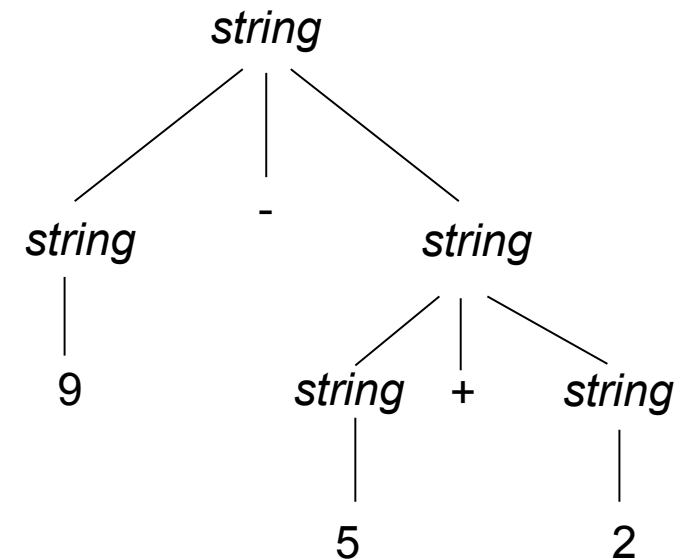
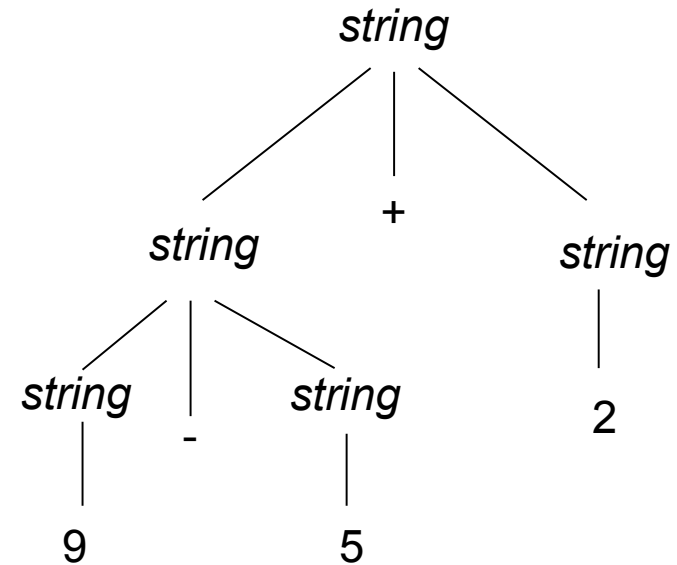
```
expr -> expr + term  
expr -> expr - term  
expr -> term  
term -> 0  
term -> 1  
...  
term -> 9
```



Niejednoznaczność

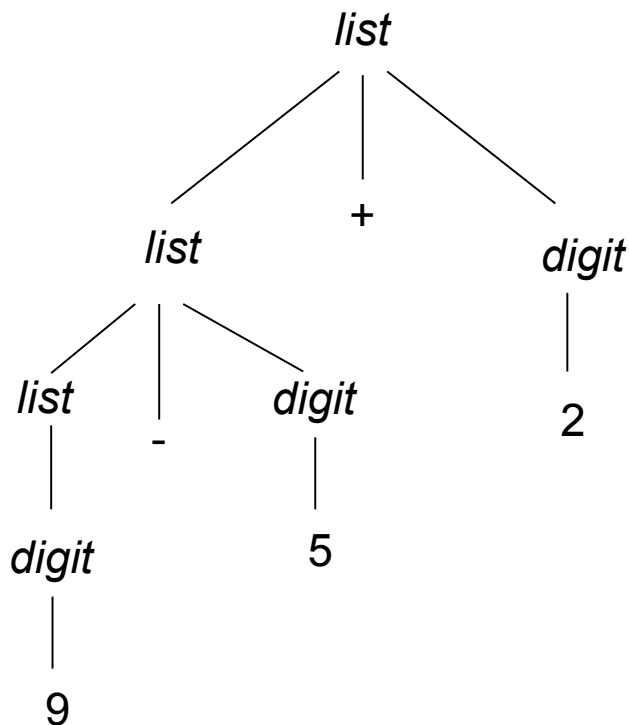
- Nie każda gramatyka jest jednoznaczna
- Jednemu ciągowi może odpowiadać kilka drzew wyprowadzenia

```
string -> string + string  
string -> string - string  
string -> 0  
string -> 1  
...  
string -> 9
```

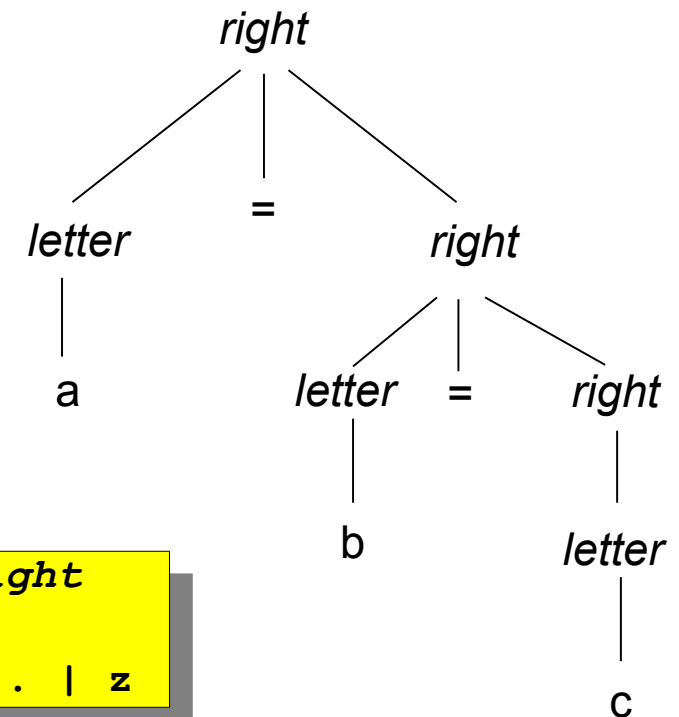


Łączność operatorów

- Operator nazywamy lewostronnie łącznym, jeżeli w sytuacji, w której po obu stronach operandu występuje operator o jednakowym priorytecie, najpierw wykonywany jest operator lewy

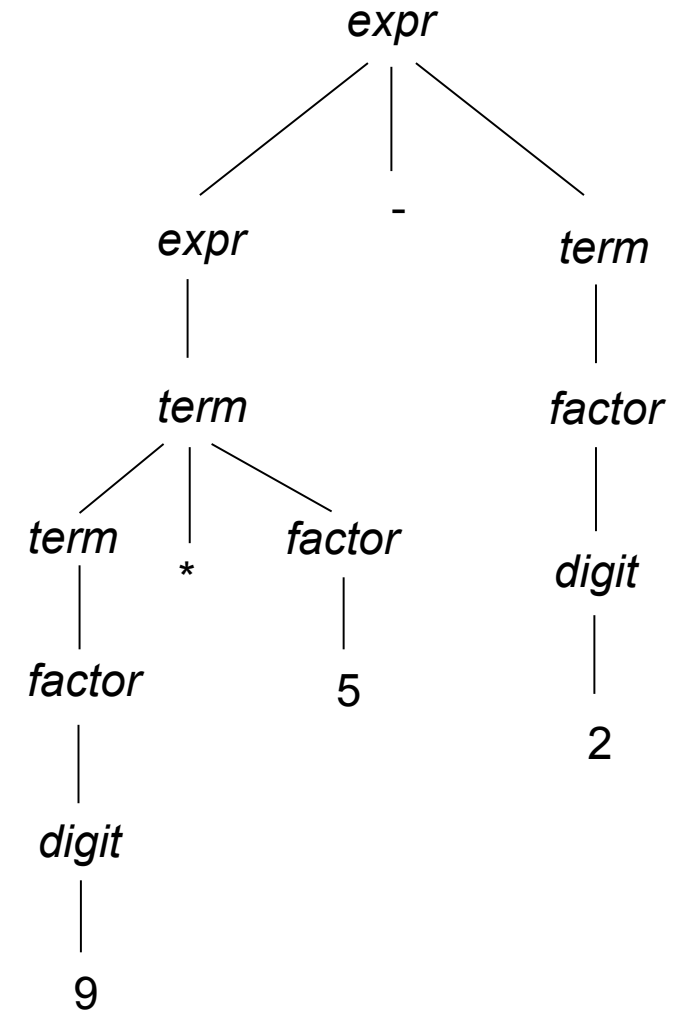


```
right -> letter = right  
right -> letter  
letter -> a | b | ... | z
```



Priorytet operatorów

```
expr -> expr + term | expr - term | term  
term -> term * factor | term / factor | factor  
factor -> digit | ( expr )  
digit -> 0 | 1 | ... | 9
```



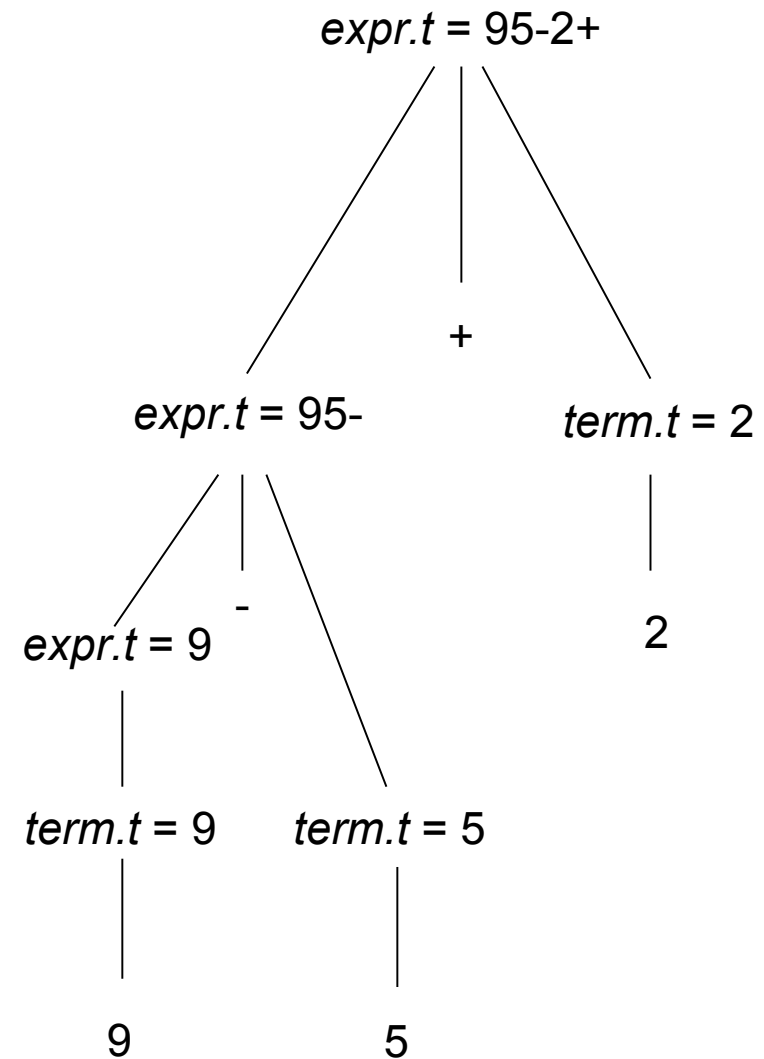
Notacja przyrostkowa

- Notacja przyrostkowa dla wyrażenia E może być zdefiniowana rekursywnie w następujący sposób:
 - Jeżeli E jest zmienną lub stałą, notacją przyrostkową dla E jest samo E
 - Jeżeli E jest wyrażeniem w postaci $E_1 \text{ op } E_2$, gdzie op jest operatorem binarnym, notacją przyrostkową dla E jest $E_1' E_2' \text{ op}$, gdzie E_1' i E_2' są notacjami przyrostkowymi odpowiednio dla E_1 i E_2
 - Jeżeli E jest wyrażeniem postaci (E_1) , wówczas notacja przyrostkowa dla E_1 jest również notacją przyrostkową dla E

Translacja sterowana składnią

Produkcja	Reguła semantyczna
$expr \rightarrow expr_1 + term$	$expr.t := expr1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
...	...
$term \rightarrow 9$	$term.t := '9'$

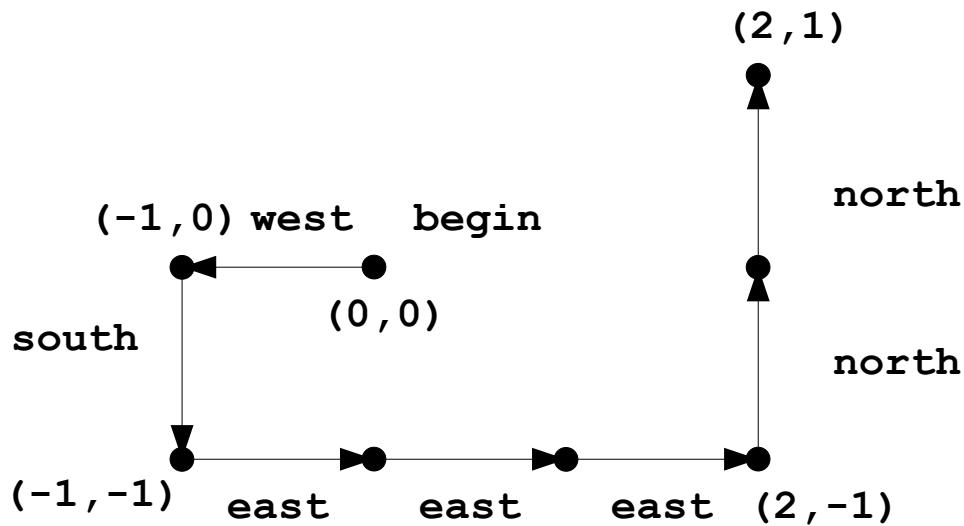
- Prosty kompilator - przekształca notację wrostkową na przyrostkową (odwrotną notację polską)
- $9-5+2 \rightarrow 95-2+$



Mobilny robot

```
seq -> seq instr | begin
instr -> east | north | west | south
```

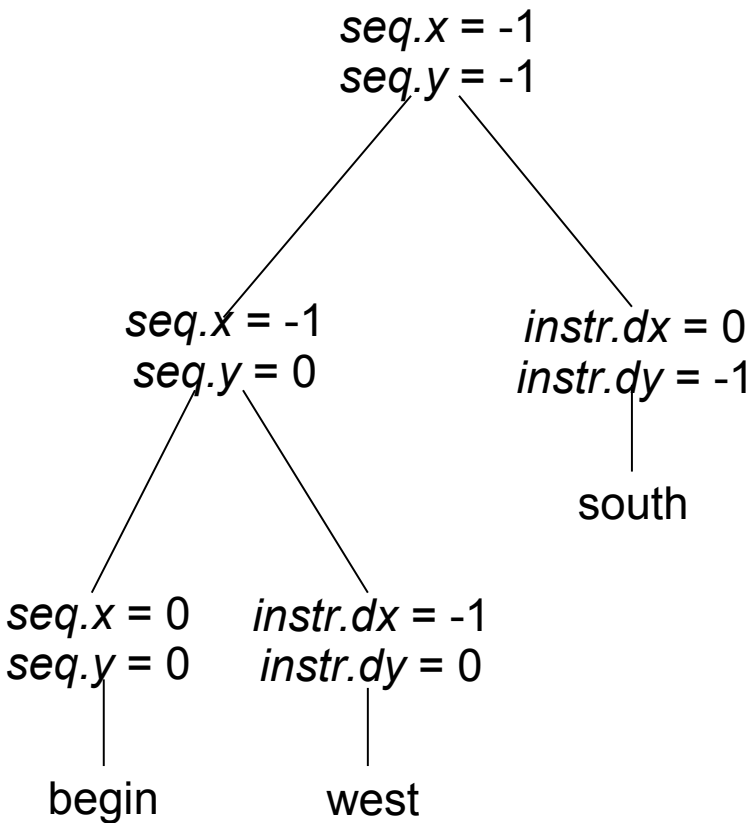
```
begin west south east east east north north
```



Produkcja	Reguła semantyczna
$seq \rightarrow \mathbf{begin}$	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1 \mathit{instr}$	$seq.x := seq_1.x + \mathit{instr}.dx$ $seq.y := seq_1.y + \mathit{instr}.dy$
$\mathit{instr} \rightarrow \mathbf{east}$	$\mathit{instr}.dx := 1$ $\mathit{instr}.dy := 0$
$\mathit{instr} \rightarrow \mathbf{north}$	$\mathit{instr}.dx := 0$ $\mathit{instr}.dy := 1$
$\mathit{instr} \rightarrow \mathbf{west}$	$\mathit{instr}.dx := -1$ $\mathit{instr}.dy := 0$
$\mathit{instr} \rightarrow \mathbf{south}$	$\mathit{instr}.dx := 0$ $\mathit{instr}.dy := -1$

Mobilny robot

begin west south



Produkcja	Reguła semantyczna
$seq \rightarrow \mathbf{begin}$	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1 \mathbf{instr}$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow \mathbf{east}$	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow \mathbf{north}$	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow \mathbf{west}$	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow \mathbf{south}$	$instr.dx := 0$ $instr.dy := -1$

Przechodzenie drzewa w głąb

- Definicja sterowana składnią nie narzuca kolejności obliczania atrybutów
 - każda kolejność uwzględniająca zależności między atrybutami jest poprawna
- W dotychczasowych przykładach atrybuty można było wyliczyć według kolejności przechodzenia w głąb

```
procedure visit(n: node);  
begin  
  for each child m of n, from left to right do  
    visit(m);  
  evaluate semantic rules at node n  
end
```

Schemat translacji

$expr \rightarrow expr_1 + term \{ \text{print}('+') \}$

$expr \rightarrow expr_1 - term \{ \text{print}('-') \}$

$expr \rightarrow term$

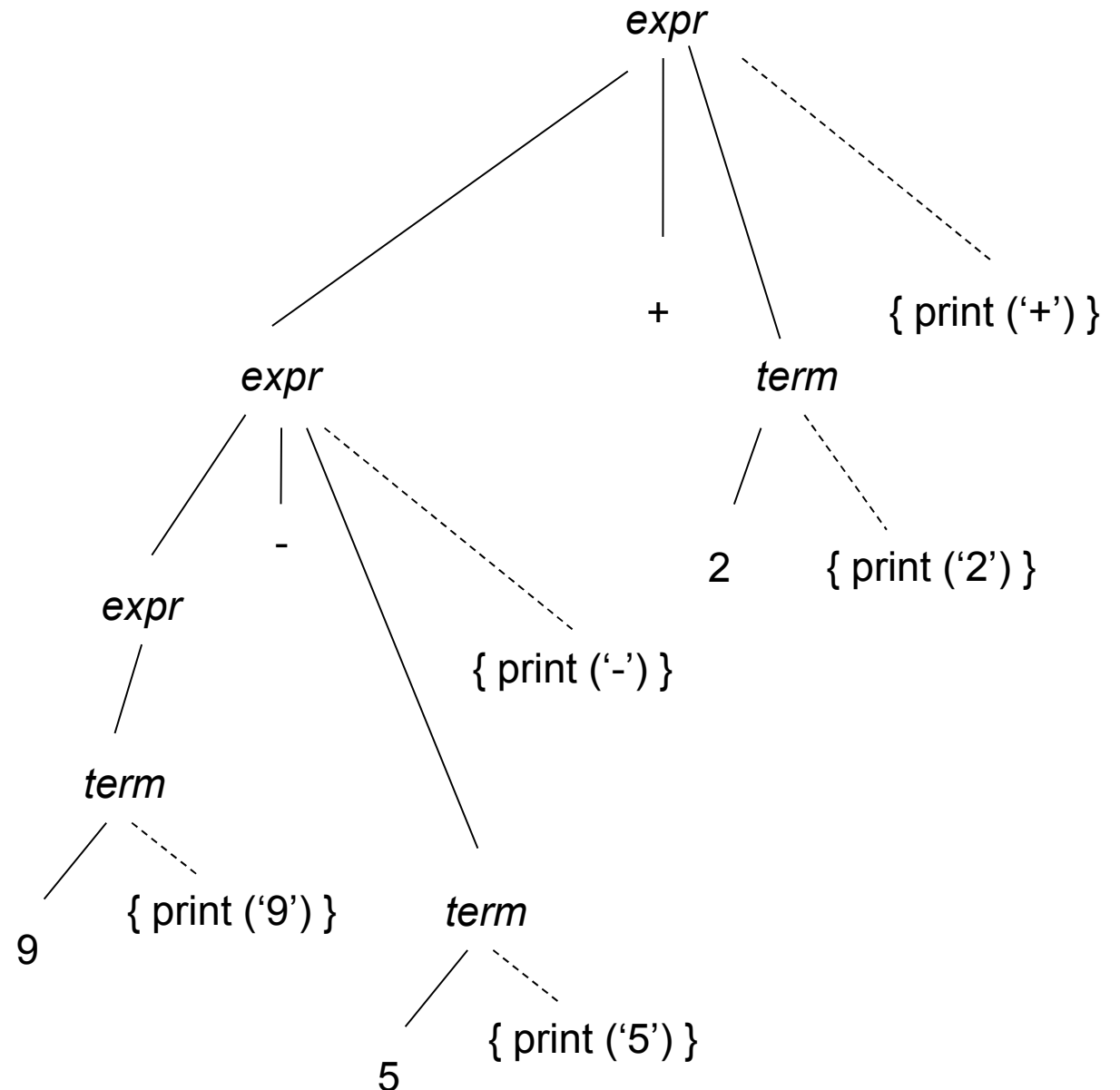
$term \rightarrow 0 \{ \text{print}('0') \}$

$term \rightarrow 1 \{ \text{print}('1') \}$

...

$term \rightarrow 9 \{ \text{print}('9') \}$

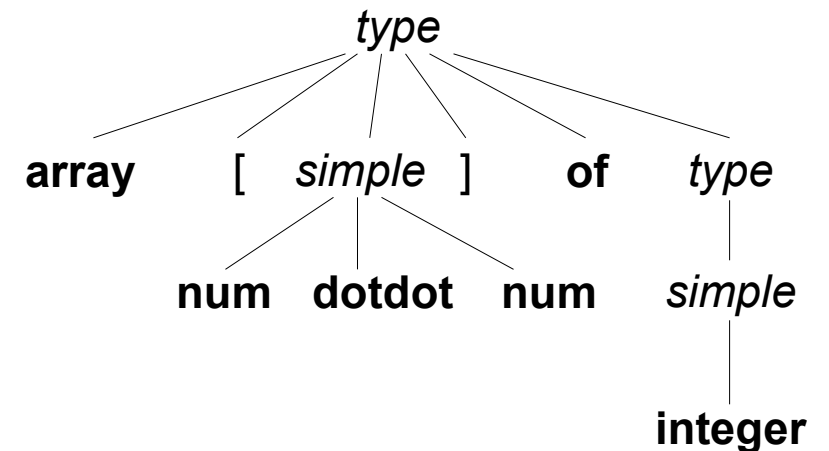
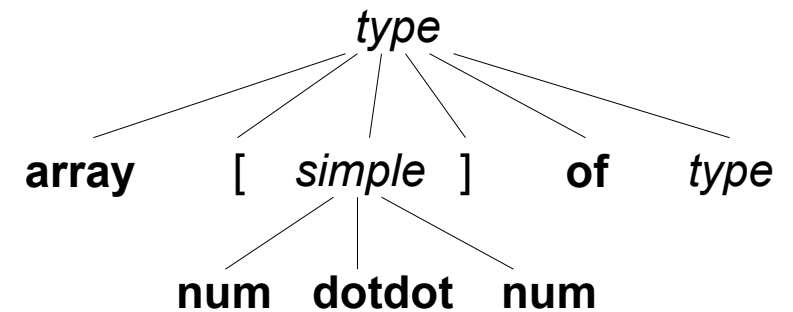
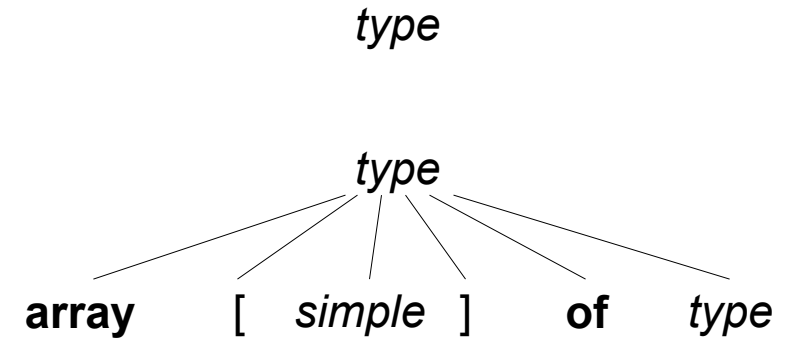
Translacja 9-5+2 na 95-2+



Analiza składniowa metodą zstępującą

array [num dotdot num] of integer

```
type   -> simple
        | ^id
        | array [ simple ] of type
simple -> integer
        | char
        | num dotdot num
```



Parser predykcyjny

```
procedure match(t: token)
```

```
begin
```

```
  if lookahead = t then
```

```
    lookahead := nexttoken
```

```
  else error
```

```
end;
```

```
procedure type;
```

```
begin
```

```
  if lookahead is in { integer, char, num } then
```

```
    simple
```

```
  else if lookahead = '^' then begin
```

```
    match('^'); match(id)
```

```
  end
```

```
  else if lookahead = array then begin
```

```
    match(array); match('['); simple; match(')'); match(of); type
```

```
  end
```

```
  else error
```

```
end;
```

```
procedure simple;
```

```
begin
```

```
  if lookahead = integer then
```

```
    match(integer)
```

```
  else if lookahead = char then
```

```
    match(char)
```

```
  else if lookahead = num then begin
```

```
    match(num); match(dotdot); match(num)
```

```
  end
```

```
  else error
```

```
end;
```

```
type    -> simple
```

```
        | ^id
```

```
        | array [ simple ] of type
```

```
simple -> integer
```

```
        | char
```

```
        | num dotdot num
```

```
FIRST(simple) = { integer, char, num }
```

```
FIRST(^id) = { ^ }
```

```
FIRST(array [ simple ] of type) = { array }
```

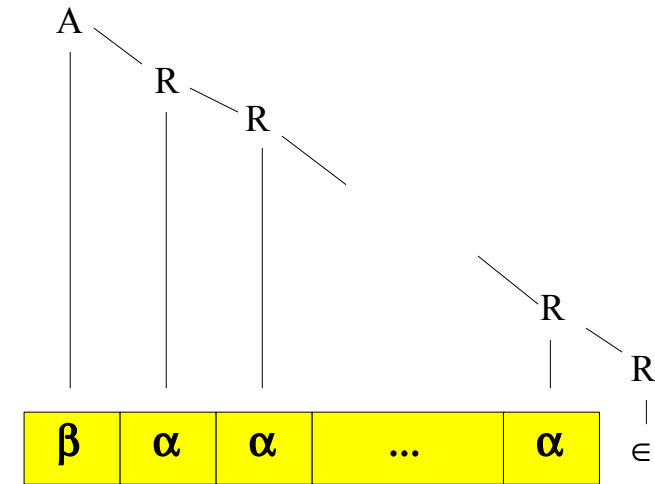
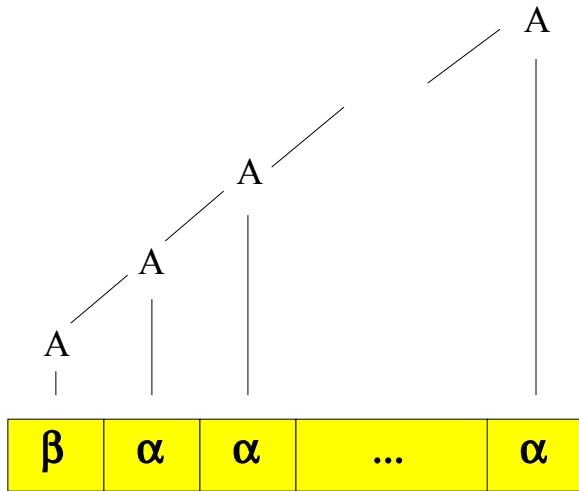
Produkcje puste

```
stmt → begin opt_stmts end  
opt_stmts → stmt_list | ∈
```

- Produkcja pusta jest wybierana wtedy, kiedy nie udało się dopasować żadnej innej

Lewostronna rekursja

expr \rightarrow *expr* + *term*



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

$expr \rightarrow expr + term \mid term$
 $A = expr; \alpha = + term; \beta = term$

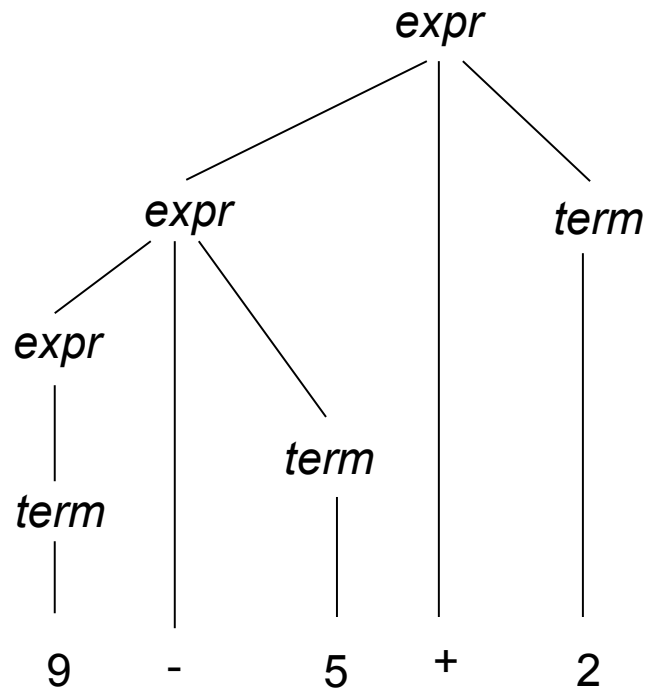
$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$

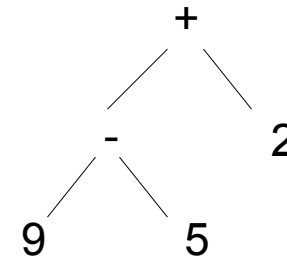
$A \rightarrow A\alpha \mid A\beta \mid \gamma$

$R \rightarrow \alpha R \mid \beta R \mid \epsilon$

Drzewa konkretne i abstrakcyjne



Drzewo konkretne

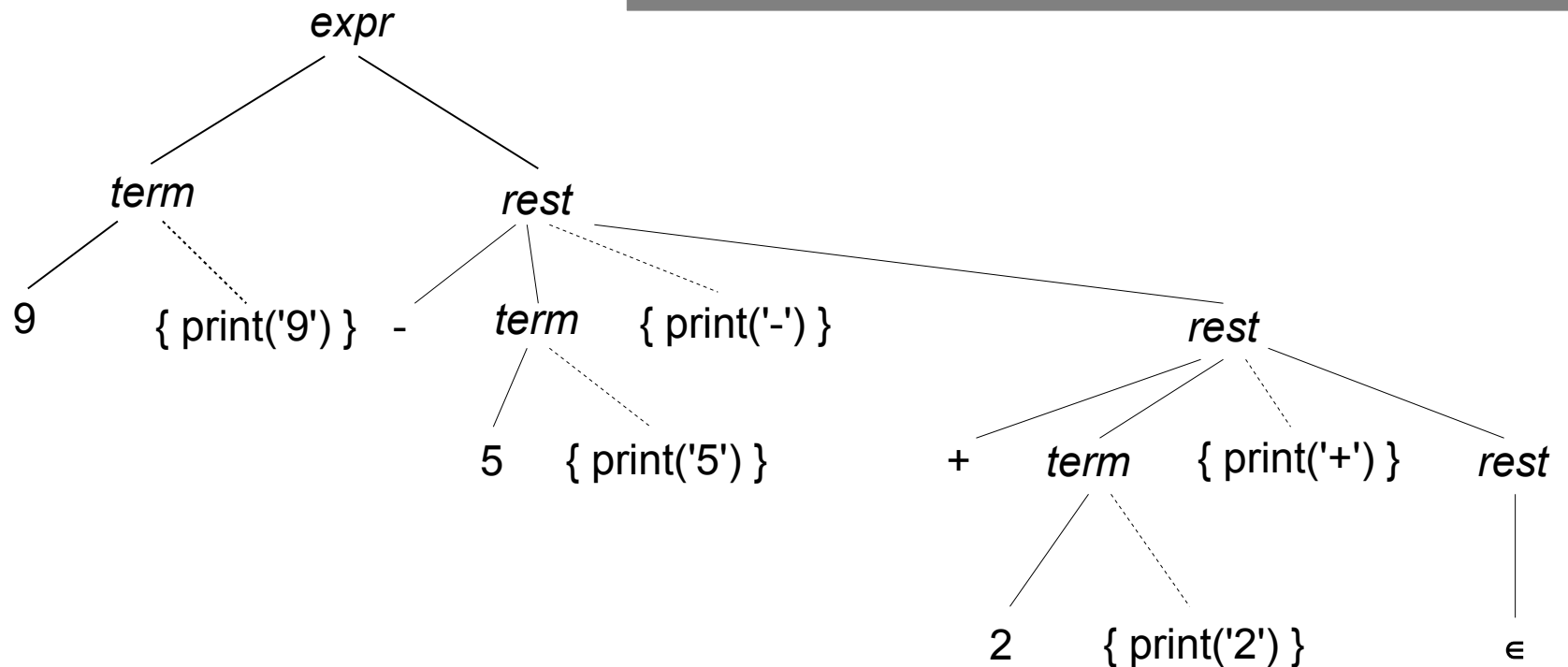


Drzewo abstrakcyjne

Adaptacja gramatyki do analizy za pomocą parsera predykcyjnego

```
expr -> expr + term { print('+') }  
expr -> expr - term { print('-') }  
expr -> term  
term -> 0 { print ('0') }  
term -> 1 { print ('1') }  
...  
term -> 9 { print ('9') }
```

```
expr -> term rest  
rest -> + term { print('+') } rest  
      | - term { print('-') } rest  
      | ε  
term -> 0 { print ('0') }  
term -> 1 { print ('1') }  
...  
term -> 9 { print ('9') }
```



Program parsera predykcyjnego

```
expr -> term rest
rest -> + term { print('+') } rest
      | - term { print('-') } rest
      | ε
term  -> 0 { print ('0') }
term  -> 1 { print ('1') }
      ...
term  -> 9 { print ('9') }
```

```
expr()
{
    term(); rest();
}

rest()
{
    if(lookahead == '+') {
        match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); rest();
    }
    else ;
}

term()
{
    if(isdigit(lookahead)) {
        putchar(lookahead); match(lookahead);
    }
    else error();
}

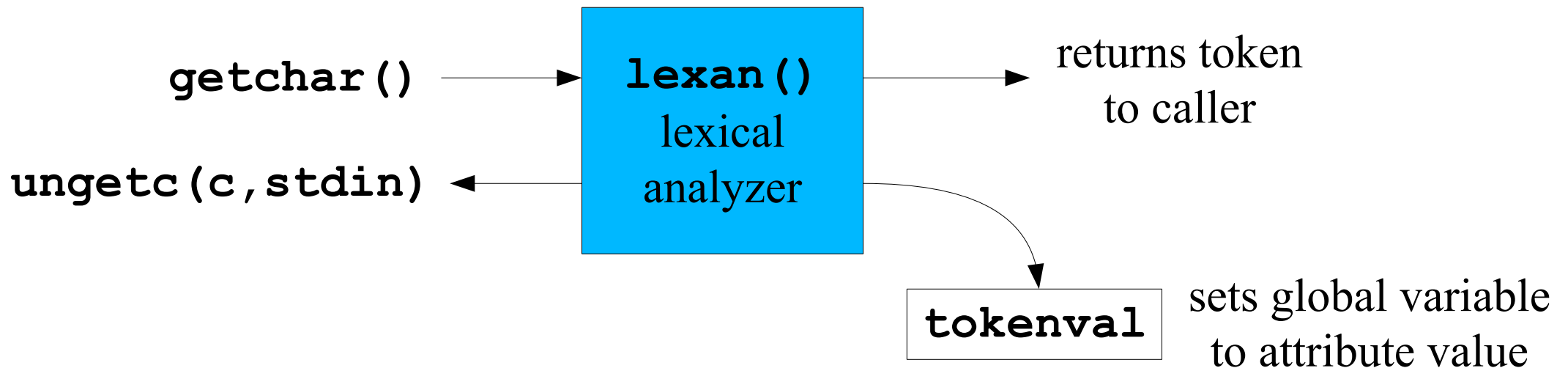
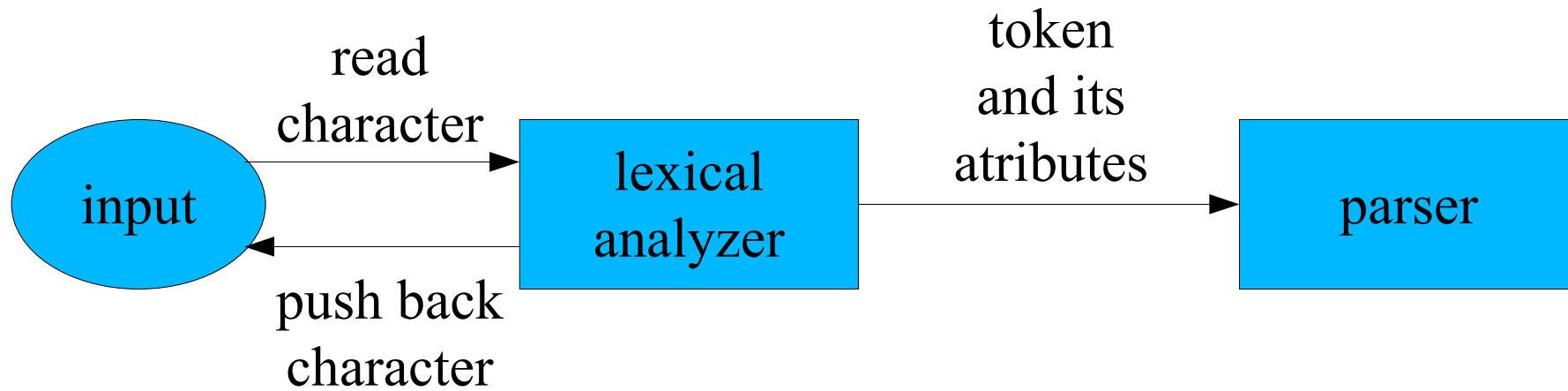
match(int t)
{
    if (lookahead == t)
        lookahead = getchar();
    else error();
}
```

Eliminacja rekursji końcowej

```
rest()
{
L:   if(lookahead == '+') {
        match('+'); term(); putchar('+'); goto L;
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); goto L;
    }
    else ;
}
```

```
expr()
{
    term();
while(1)
    if (lookahead == '+') {
        match('+'); term(); putchar('+');
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-');
    }
    else break;
}
```

Interfejs analizatora leksykalnego



Program analizatora leksykalnego

```
int lineno = 1;
int tokenval = NONE;

int lexan ()
{
    int t;
    while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t')
            ; /* strip out blanks and tabs */
        else if (t == '\n')
            lineno++;
        else if (isdigit (t)) {
            tokenval = t - '0';
            t = getchar();
            while (isdigit(t)) {
                tokenval = tokenval*10 + t - '0';
                t = getchar();
            }
            ungetc (t, stdin);
            return NUM;
        }
        else {
            tokenval = NONE;
            return t;
        }
    }
}
```

Tablica symboli

```
int insert(const char* s, int t); /* returns index of new entry for string s, token t */  
int lookup(const char* s); /* returns index of the entry for string s, or 0 if s is not found */
```

- Tablica symboli jest wykorzystywana do przechowywania zmiennych oraz słów kluczowych
- Jest inicjalizowana poprzez wstawienie do niej słów kluczowych:

```
insert("div", DIV);  
insert("mod", MOD);
```

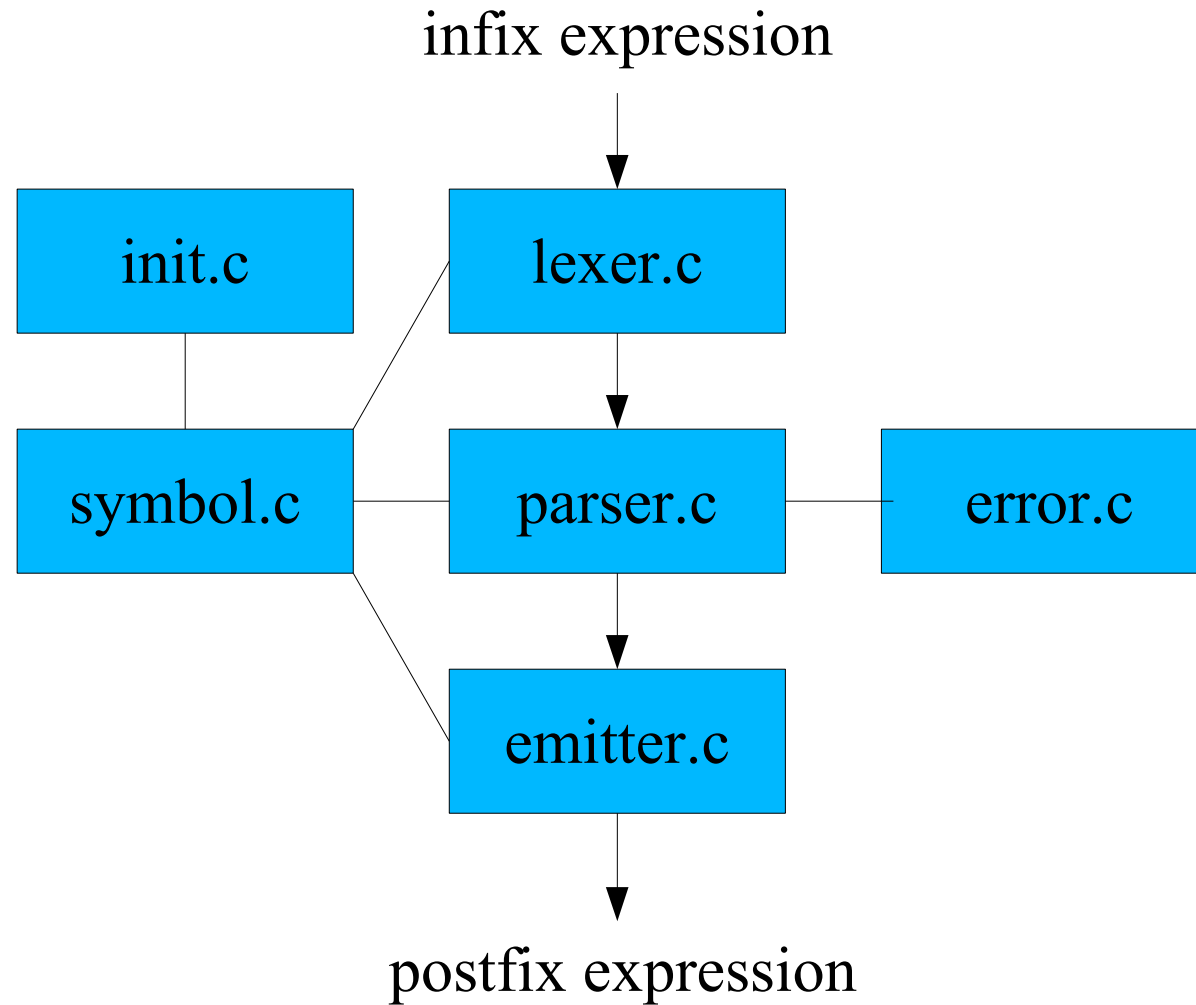
Program analizatora leksykalnego z tablicą symboli

```
int lineno = 1;
int tokenval = NONE;
int lexan () {
    int t;
    while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t');
        else if (t == '\n')
            lineno++;
        else if (isdigit (t)) {
            ungetc (t, stdin);
            scanf ("%d", &tokenval);
            return NUM;
        } else if (isalpha (t)) {
            int p, b = 0;
            while (isalnum (t)) {
                lexbuf[b] = t;
                t = getchar ();
                b++;
                if (b >= BSIZE) error ("compiler error");
            }
            lexbuf[b] = EOS;
            if (t != EOF) ungetc (t, stdin);
            p = lookup (lexbuf);
            if (p == 0) p = insert (lexbuf, ID);
            tokenval = p;
            return symtable[p].token;
        } else if (t == EOF)    return DONE;
        else {
            tokenval = NONE;
            return t;
        }
    }
}
```

Gramatyka translatora

```
start -> list eof
list  -> expr ; list
      | ε
expr  -> expr + term    { print('+') }
      | expr - term    { print('-') }
      | term
term  -> term * factor  { print('*') }
      | term / factor  { print('/') }
      | term div factor { print('DIV') }
      | term mod factor { print('MOD') }
      | factor
factor -> ( expr )
      | id              { print(id.lexeme) }
      | num             { print(num.value) }
```

Struktura translatora



Analizator leksykalny

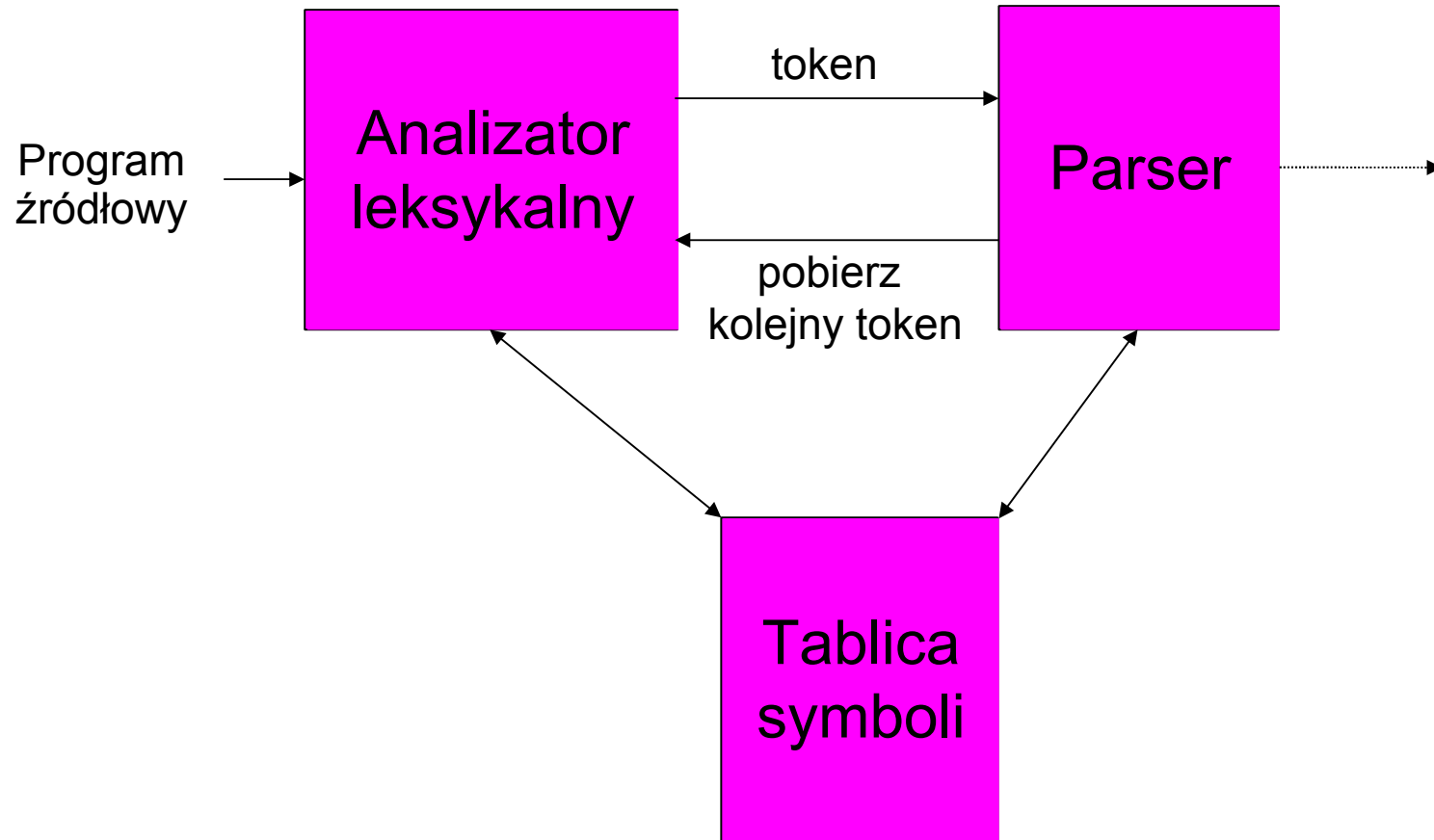
TOKENY: '+' '-' '*' '/' DIV MOD '(' ')' ID NUM DONE

Lexeme	Token	Attribute value
white space		
sequence of digits	NUM	numeric value of sequence
div	DIV	
mod	MOD	
other sequences of a letter then letters and digits	ID	index into symtable
end-of-file character	DONE	
any other character	that character	

Zmodyfikowana gramatyka translatora

```
start      -> list eof
list      -> expr ; list
           |  $\epsilon$ 
expr      -> term moreterms
moreterms -> + term { print('+') } moreterms
           | - term { print('-') } moreterms
           |  $\epsilon$ 
term      -> factor morefactors
morefactors -> * factor { print('*') } morefactors
           | / factor { print('/') } morefactors
           | div factor { print('DIV') } morefactors
           | mod factor { print('MOD') } morefactors
           |  $\epsilon$ 
factor    -> ( expr )
           | id           { print(id.lexeme) }
           | num          { print(num.value) }
```

Analizator leksykalny i jego rola w kompilatorze



Analiza leksykalna i składniowa

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

if → if

then → then

else → else

relop → < | <= | = | <> | > | >=

id → letter (letter | digit)*

num → digit⁺(.digit⁺)?(E(+|-)? digit⁺)?

Plik wejściowy do programu (f)lex

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

/* regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

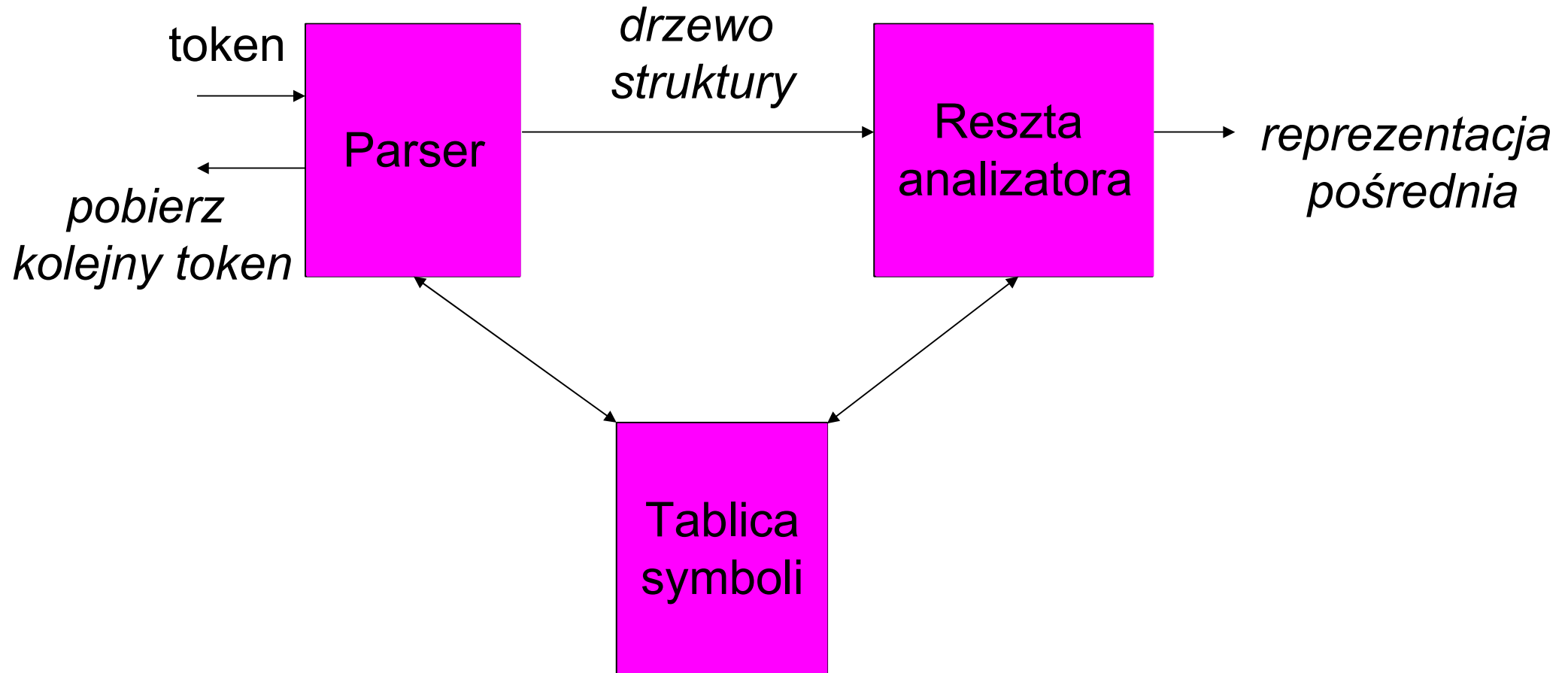
%%
{ws}     { /* no action and no return */}
if       {return(IF);}
then     {return(THEN);}
else     {return(ELSE);}
{id}     {yylval = install_id(); return(ID);}
{number} {yylval = install_num(); return(NUMBER);}
"<"     {yylval = LT; return(RELOP);}
"<="    {yylval = LE; return(RELOP);}
"="      {yylval = EQ; return(RELOP);}
"<>"    {yylval = NE; return(RELOP);}
">"     {yylval = GT; return(RELOP);}
">="    {yylval = GE; return(RELOP);}

%%

int install_id() {
    /* procedure to install the lexeme, whose first character is pointed by yytext
    and whose length is yyleng, into the symbol table and return a pointer thereto */
}

int install_num() {
    /* similar procedure to install a lexeme that is a number */
}
```

Rola parsera w kompilatorze



Obsługa błędów

Typy błędów

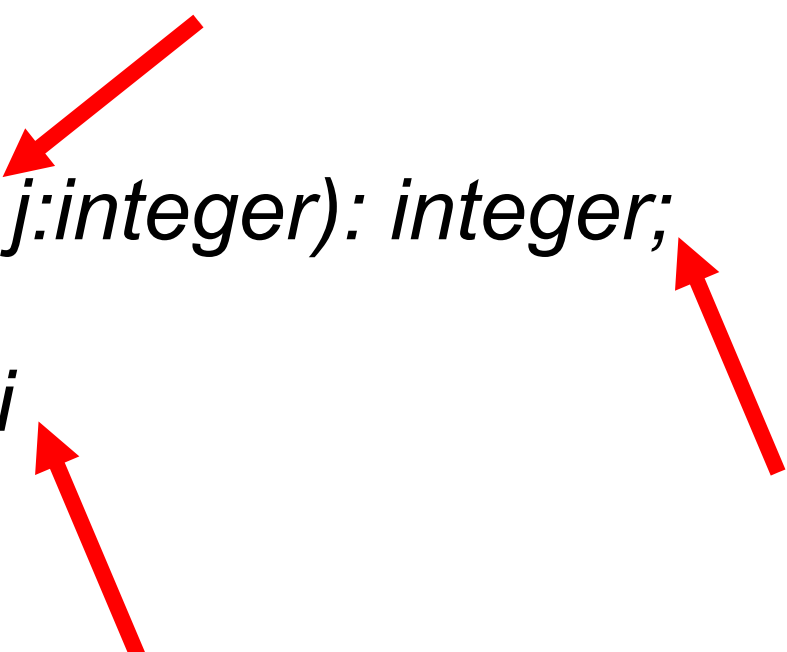
- Leksykalne
- Syntaktyczne
- Semantyczne
- Logiczne

Program obsługi błędów

- Powinien informować o błędach jasno i precyzyjnie
- Wydobywać się z błędów w celu wykrycia kolejnych
- Nie powinien spowalniać znacząco przetwarzania poprawnych programów

Typowe błędy w języku Pascal

```
program prmax(input,output);  
var  
    x,y: integer;  
function max(i:integer; j:integer): integer;  
begin  
    if I > j then max:=i  
    else max :=j  
end;  
  
begin  
    readln (x,y);  
    writeln(max(x,y))  
end.
```



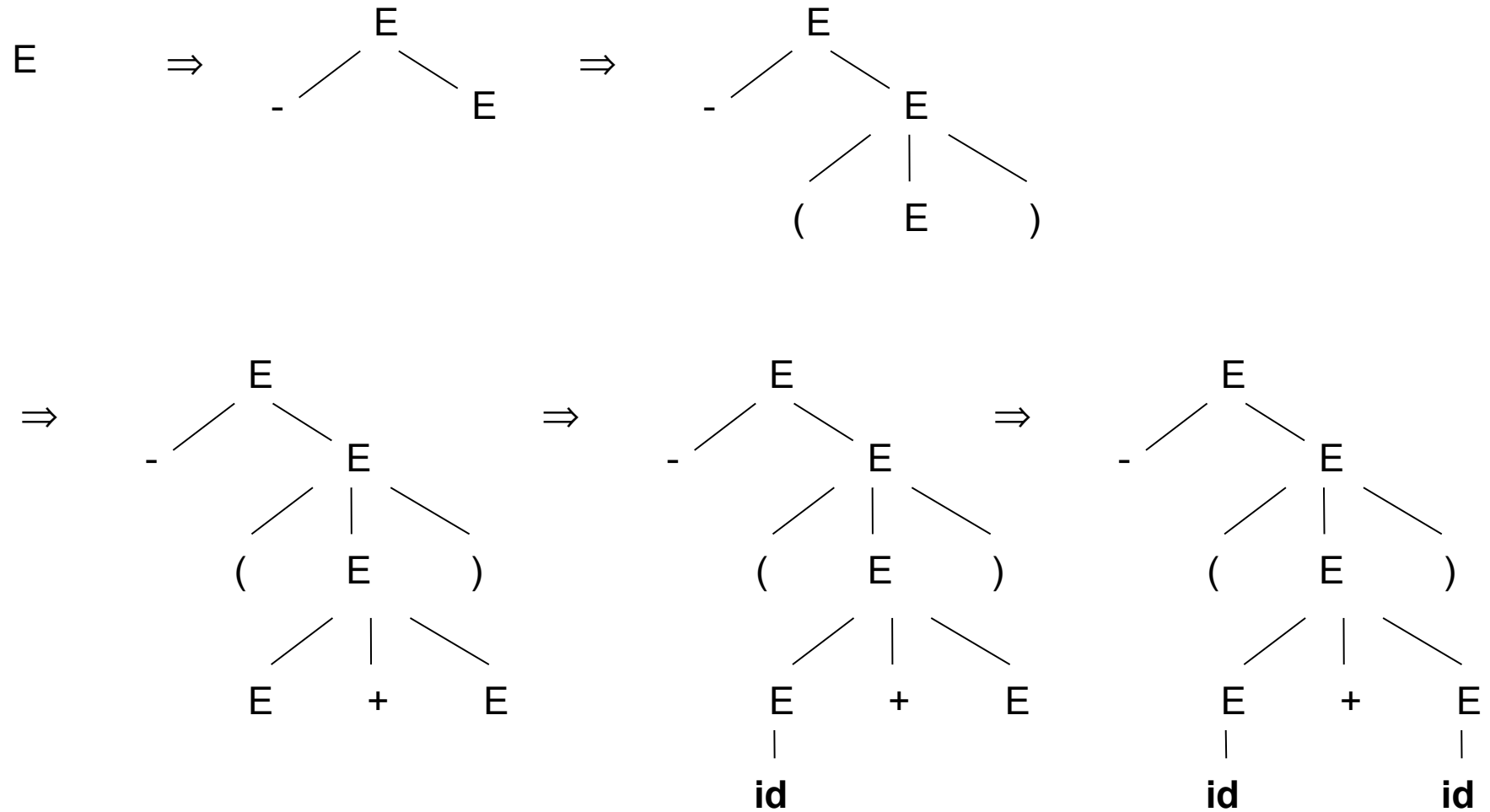
Strategie wydobywania się z błędów

- Panika - pomijanie symboli z wejścia aż do napotkania tokenu synchronizującego
- Na poziomie zdania - lokalna korekcja błędów
- Produkcje uwzględniające błędy
- Globalna korekcja

Wyprowadzenia

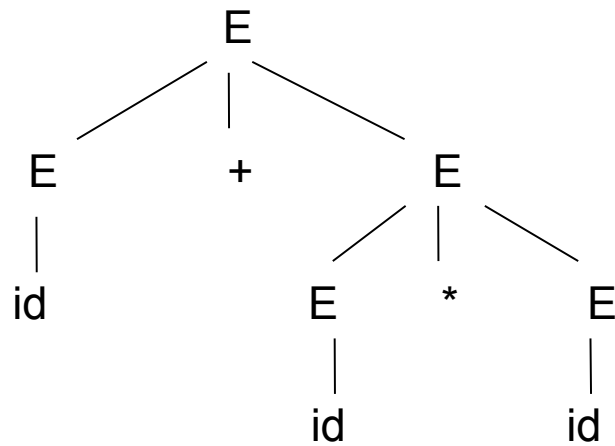
$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$
$$E \Rightarrow -E$$
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$
$$E \Rightarrow -(\mathbf{id})$$
$$E \underset{\text{Im}}{\Rightarrow} -E \underset{\text{Im}}{\Rightarrow} -(E) \underset{\text{Im}}{\Rightarrow} -(E+E) \underset{\text{Im}}{\Rightarrow} -(\mathbf{id}+E) \underset{\text{Im}}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Wyprowadzenia

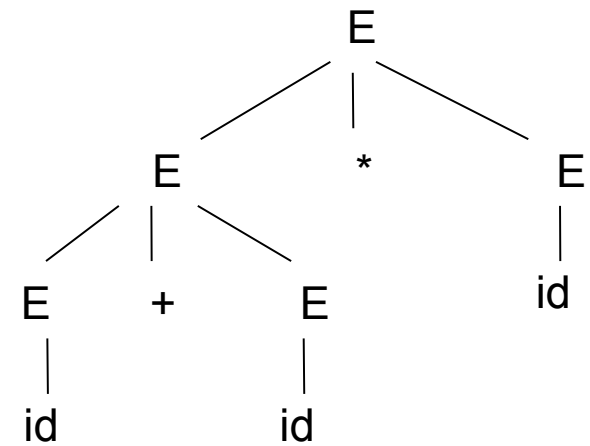


Niejednoznaczność

$E \Rightarrow E + E$
 $\Rightarrow id + E$
 $\Rightarrow id + E * E$
 $\Rightarrow id + id * E$
 $\Rightarrow id + id * id$



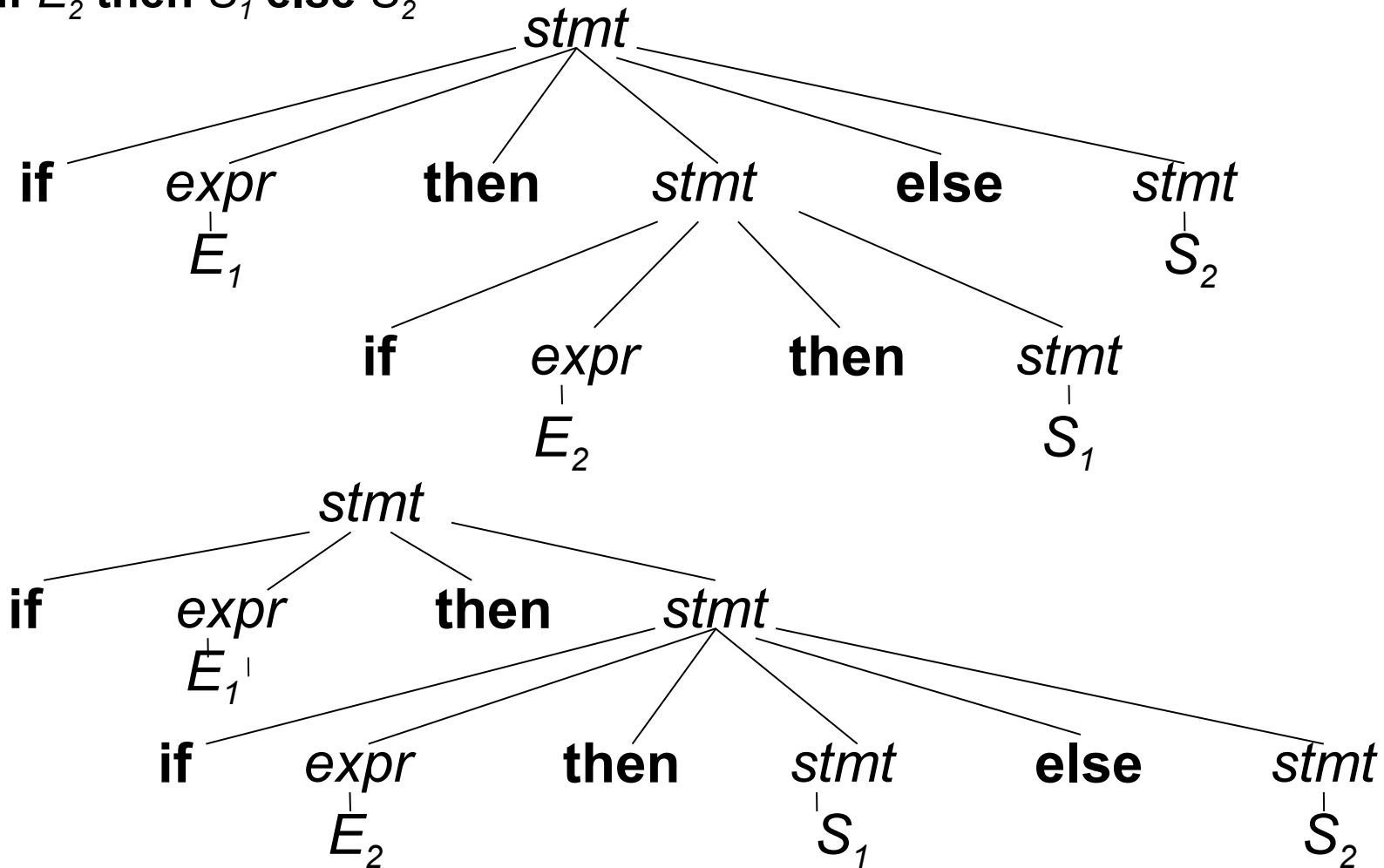
$E \Rightarrow E * E$
 $\Rightarrow E + E * E$
 $\Rightarrow id + E * E$
 $\Rightarrow id + id * E$
 $\Rightarrow id + id * id$



Problem “dangling else”

stmt →
| **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

if E_1 then if E_2 then S_1 else S_2



Problem “dangling else” - ścisłe rozwiązanie

```
stmt → matched_stmt  
      | unmatched_stmt  
matched_stmt → if expr then matched_stmt else matched_stmt  
              | other  
unmatched_stmt → if expr then stmt  
                 | if expr then matched_stmt else unmatched_stmt
```

Parser predykcyjny

- Eliminacja lewostronnej rekursji

$expr \rightarrow expr + term \mid term$

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

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$

- Lewostronna faktoryzacja

$stmt \rightarrow \mathbf{if\ expr\ then\ stmt}$
 $\quad \mid \mathbf{if\ expr\ then\ stmt\ else\ stmt}$

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

$A \rightarrow \alpha A'$

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

Przekształcenia dla gramatyki wyrażeń arytmetycznych

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

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

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

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

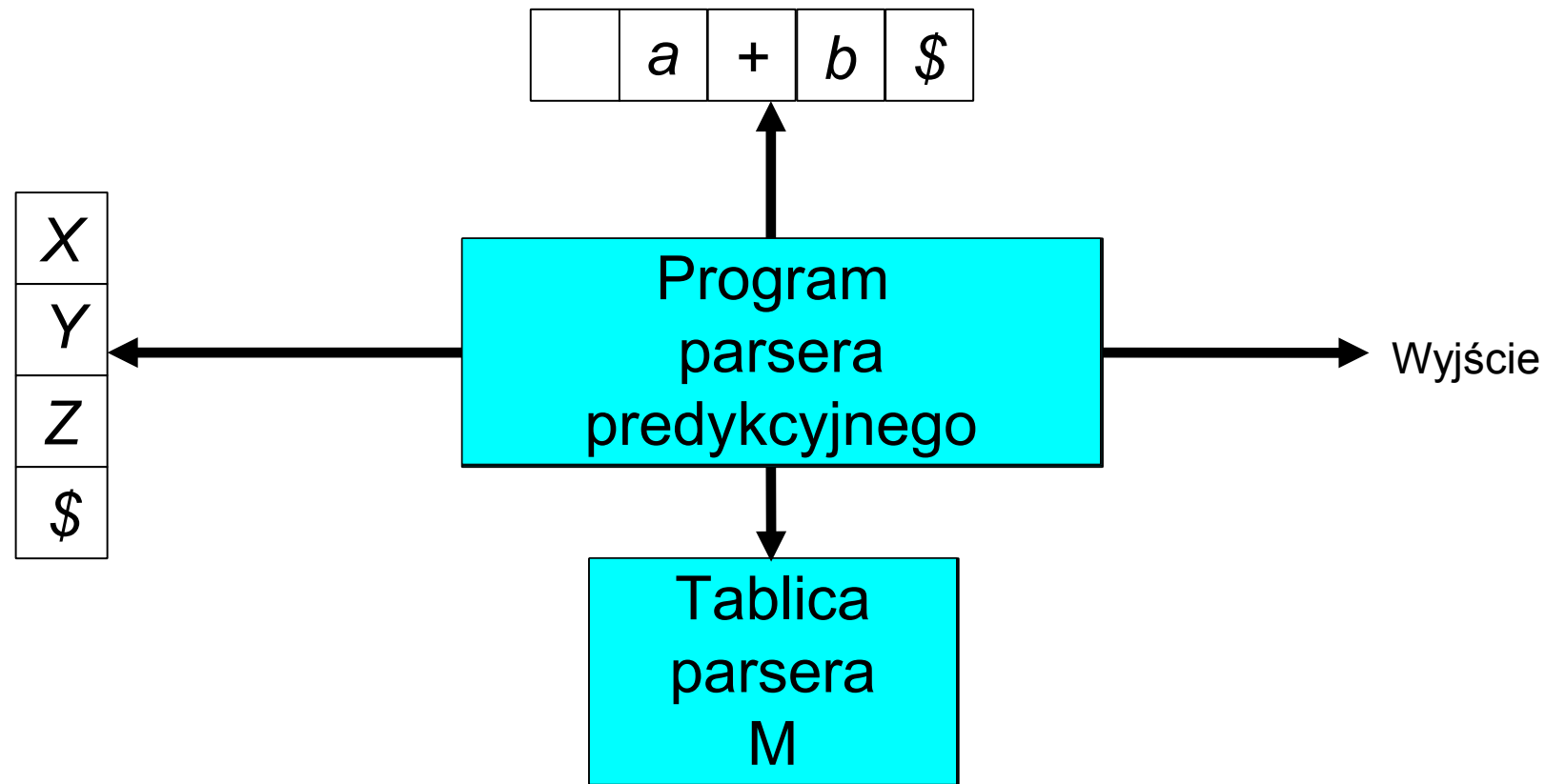
$$T' \rightarrow *FT' \mid \epsilon$$

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

Jakich konstrukcji nie można opisać za pomocą gramatyk bezkontekstowych

- Deklarowanie identyfikatorów przed ich użyciem
- Wywołanie funkcji z właściwą liczbą parametrów

Parser predykcyjny bez rekursji



1. Jeżeli $X=a=\$$ to zakończ pracę
2. Jeżeli $X=a\neq\ \$$ podnieś X ze stosu i przesunij wskaźnik wejściowy do przodu
3. Jeżeli X jest symbolem nieterminalnym zastosuj produkcję z tablicy $M[X,a]$

Tablica M dla wyrażeń arytmetycznych

Symbol nieterminalny	Symbol wejściowy					
	Id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Określenie zbioru FIRST(X)

1. Jeżeli X jest symbolem terminalnym, $\text{FIRST}(X) = \{X\}$

2. Jeżeli istnieje produkcja

$$X \rightarrow \epsilon,$$

dodaj ϵ do $\text{FIRST}(X)$

3. Jeżeli X jest symbolem nieterminalnym i istnieje produkcja

$$X \rightarrow Y_1 Y_2 \dots Y_k$$

umieść a w $\text{FIRST}(X)$ jeżeli dla jakiegoś i , a należy do $\text{FIRST}(Y_i)$ oraz ϵ należy do $\text{FIRST}(Y_1), \dots,$

$\text{FIRST}(Y_{i-1})$

Określenie zbioru FOLLOW(X)

1. Umieść \$ w FOLLOW(S), gdzie S jest symbolem startowym
2. Jeżeli istnieje produkcja $A \rightarrow \alpha B \beta$,
umieść wszystkie elementy FIRST(β) z
wyjątkiem ϵ w FOLLOW(B)
3. Jeżeli istnieje produkcja $A \rightarrow \alpha B$
albo produkcja $A \rightarrow \alpha B \beta$ gdzie FIRST(β) zawiera
 ϵ , umieść wszystkie elementy FOLLOW(A) w
FOLLOW(B)

Tworzenie tablicy M

1. Dla każdej produkcji $A \rightarrow \alpha$ wykonaj kroki 2 i 3
2. Dla każdego symbolu terminalnego a należącego do $\text{FIRST}(\alpha)$ umieść $A \rightarrow \alpha$ w $M[A, a]$
3. Jeżeli $\text{FIRST}(\alpha)$ zawiera ϵ , umieść $A \rightarrow \alpha$ w $M[A, b]$ dla każdego b ze zbioru $\text{FOLLOW}(A)$

Parser predykcyjny - wydobywanie się z błędów

- **Tokeny synchronizujące**
 - FOLLOW(A)
 - Słowa kluczowe
 - FIRST(A)
 - Pusta produkcja (jeżeli istnieje) jako domyślna w wypadku błędu
 - Wstawienie tokenu, jeżeli jest na wierzchu stosu
- **Lokalna korekcja błędów**

Tablica M z tokenami synchronizacyjnymi

Symbol nieterminalny	Symbol wejściowy					
	Id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Analiza wstępująca (bottom-up)

$S \rightarrow aABe$

abbcde

$A \rightarrow Abc \mid b$

aAbcde

$B \rightarrow d$

aAde

aABe

S

$S \xRightarrow{rm} aABe \xRightarrow{rm} aAde \xRightarrow{rm} aAbcde \xRightarrow{rm} abbcde$

Uchwyty

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$E \Rightarrow \underline{E + E}$$

$$\Rightarrow E + \underline{E * E}$$

$$\Rightarrow E + E * \underline{\text{id}_3}$$

$$\Rightarrow E + \underline{\text{id}_2} * \text{id}_3$$

$$\Rightarrow \underline{\text{id}_1} + \text{id}_2 * \text{id}_3$$

$$E \Rightarrow \underline{E * E}$$

$$\Rightarrow E * \underline{\text{id}_3}$$

$$\Rightarrow \underline{E + E} * \text{id}_3$$

$$\Rightarrow E + \underline{\text{id}_2} * \text{id}_3$$

$$\Rightarrow \underline{\text{id}_1} + \text{id}_2 * \text{id}_3$$

Uchwyty

Prawostronna forma zdaniowa	Uchwyt	Redukująca produkcja
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Analiza składniowa metodą wstępującą

Stos	Wejście	Działanie
\$	$\mathbf{id_1 + id_2 * id_3}$ \$	przesuń
\$ $\mathbf{id_1}$	$\mathbf{+ id_2 * id_3}$ \$	zredukuj przez $E \rightarrow \mathbf{id}$
\$ \mathbf{E}	$\mathbf{+ id_2 * id_3}$ \$	przesuń
\$ $\mathbf{E +}$	$\mathbf{id_2 * id_3}$ \$	przesuń
\$ $\mathbf{E + id_2}$	$\mathbf{* id_3}$ \$	zredukuj przez $E \rightarrow \mathbf{id}$
\$ $\mathbf{E + E}$	$\mathbf{* id_3}$ \$	przesuń
\$ $\mathbf{E + E *}$	$\mathbf{id_3}$ \$	przesuń
\$ $\mathbf{E + E * id_3}$		\$ zredukuj przez $E \rightarrow \mathbf{id}$
\$ $\mathbf{E + E * E}$		\$ zredukuj przez $E \rightarrow \mathbf{E * E}$
\$ $\mathbf{E + E}$		\$ zredukuj przez $E \rightarrow \mathbf{E + E}$
\$ \mathbf{E}		\$ akceptuj

yacc / bison - generatory parserów

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line:    expr '\n'    { printf("%d\n", $1); }
        ;
expr:    expr '+' term    { $$ = $1 + $3; }
        | term
        ;
term:    term '*' factor  { $$ = $1 * $3; }
        | factor
        ;
factor  :    '{' expr ')' { $$ = $2; }
        | DIGIT
        ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

Wykorzystanie priorytetu operatorów

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:    lines expr '\n'    { printf("%g\n", $2); }
        |    lines '\n'
        |    /* empty */
        ;
expr :    expr '+' expr { $$ = $1 + $3; }
        |    expr '-' expr { $$ = $1 - $3; }
        |    expr '*' expr { $$ = $1 * $3; }
        |    expr '/' expr { $$ = $1 / $3; }
        |    '(' expr ')' { $$ = $2; }
        |    '-' expr %prec UMINUS { $$ = -$2; }
        |    NUMBER
        ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ');
    if ( c == '.' || isdigit(c) ) {
        ungetc(c, stdin);
        scanf("%lf",&yylval);
        return NUMBER;
    }
    return c;
}
```

yacc / bison - rozwiązywanie konfliktów

1. Reduce/reduce - wybierana pierwsza produkcja wymieniona w pliku wejściowym
2. Shift/reduce - wybierany shift

Symbolom terminalnym można przypisać priorytety i łączność w części deklarycyjnej.

Priorytet produkcji to normalnie priorytet prawego skrajnego symbolu terminalnego.

Dla konfliktu: reduce $A \rightarrow \alpha$ i shift a

reduce - jeżeli priorytet produkcji większy od priorytetu a lub są one równe i produkcja jest lewostronnie łączna

Obsługa błędów

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%

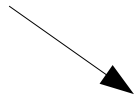
lines:  lines expr '\n'  { printf("%g\n", $2); }
      |  lines '\n'
      |  /* empty */
      |  error '\n' { yyerror("reenter last line:"); yyerrok; }
      ;

expr:   expr '+' expr   { $$ = $1 + $3; }
      |  expr '-' expr   { $$ = $1 - $3; }
      |  expr '*' expr   { $$ = $1 * $3; }
      |  expr '/' expr   { $$ = $1 / $3; }
      |  '(' expr ')' { $$ = $2; }
      |  '-' expr %prec UMINUS { $$ = -$2; }
      |  NUMBER
      ;

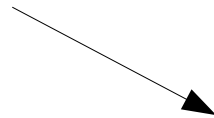
%%
```

Translacja sterowana składnią

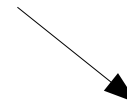
łańcuch
wejściowy



drzewo
składniowe



graf
zależności



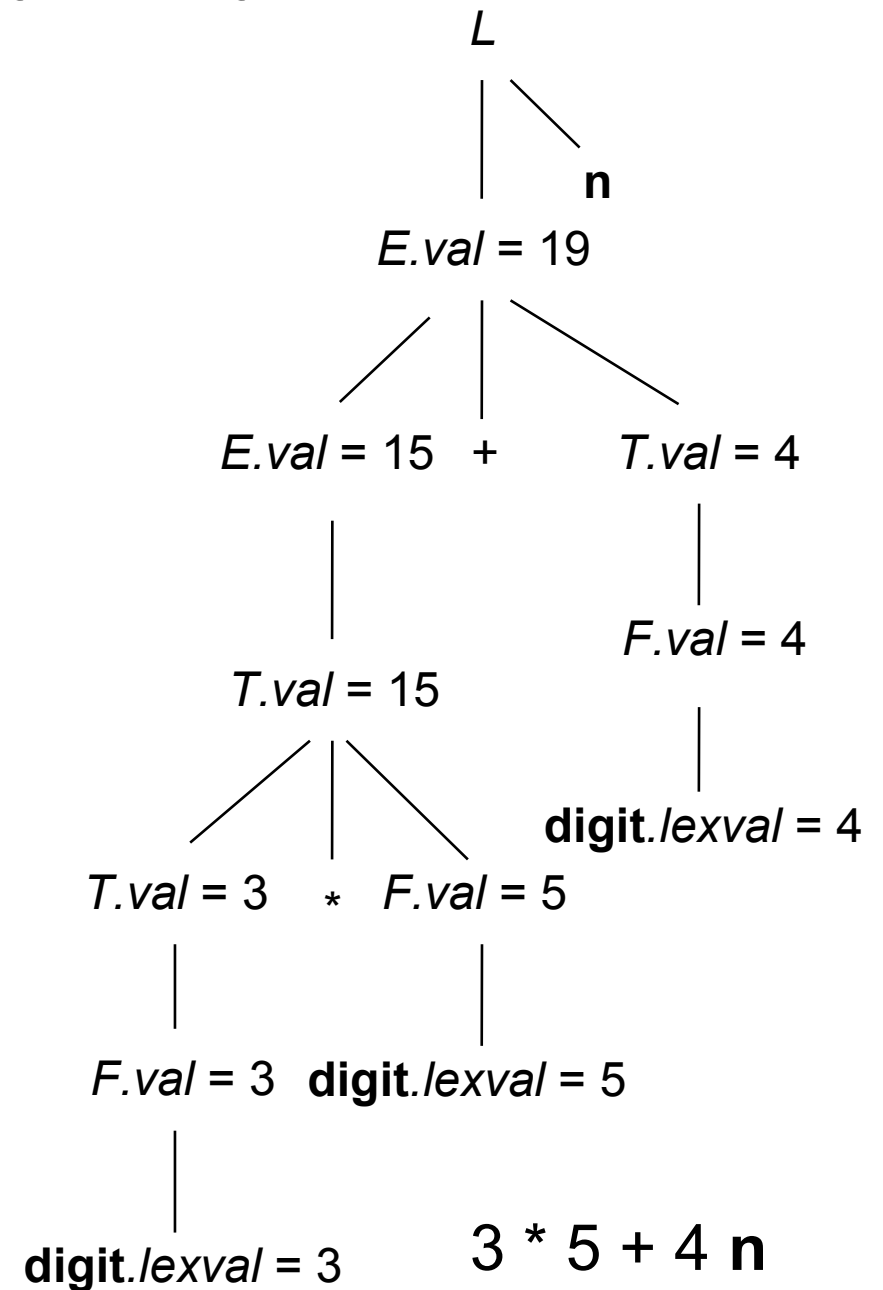
kolejność obliczania
reguł semantycznych

Atrybuty syntezowane i dziedziczone

- W definicji sterowanej składnią z każdą produkcją gramatyki $A \rightarrow \alpha$ jest związany zbiór reguł semantycznych o postaci $b := f(c_1, c_2, \dots, c_k)$, gdzie f jest funkcją, oraz
 - b jest atrybutem syntezowanym symbolu A , a c_1, c_2, \dots, c_k są atrybutami symboli produkcji
 - b jest atrybutem dziedziczonym jednego z symboli gramatycznych po prawej stronie produkcji, a c_1, c_2, \dots, c_k są atrybutami symboli produkcji

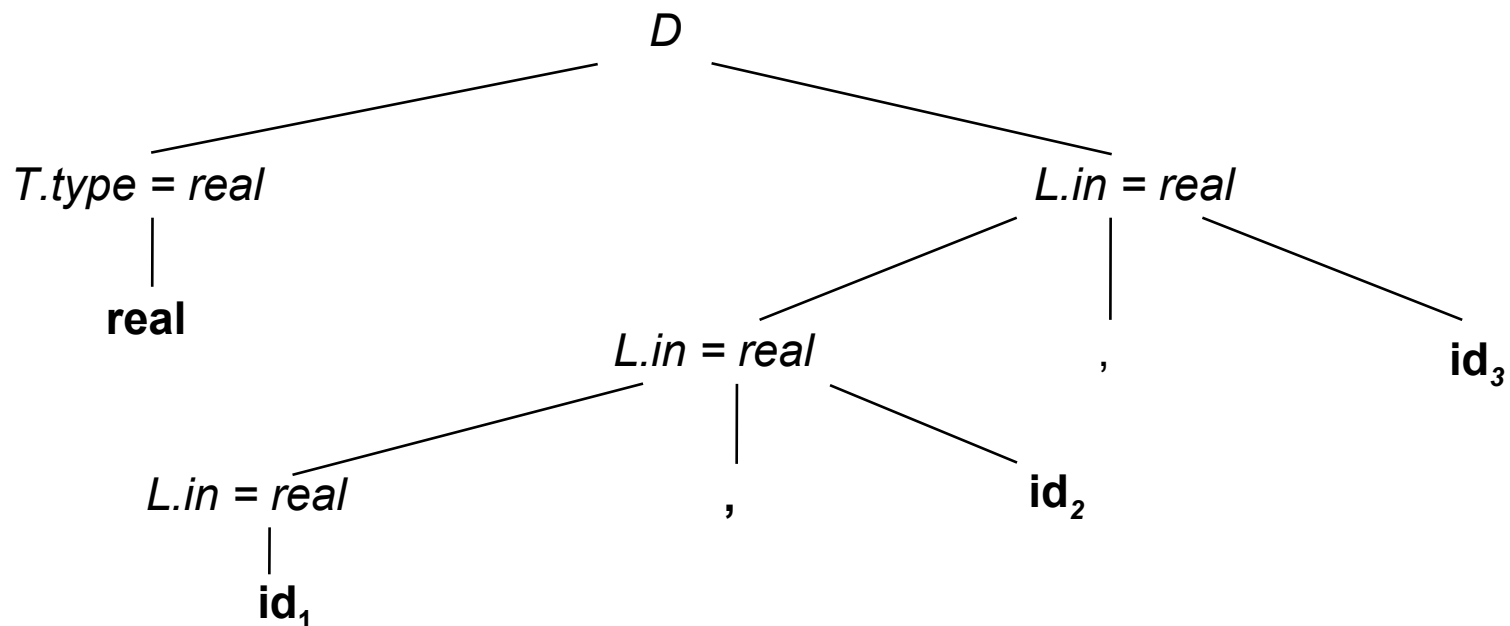
Gramatyka S-atrybutywna

Produkcja	Reguły semantyczne
$L \rightarrow E n$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$



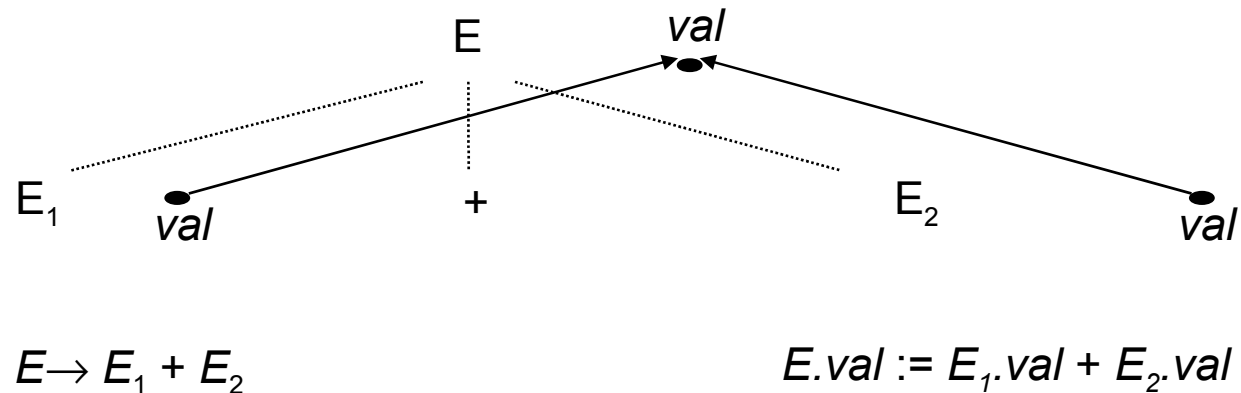
Gramatyka nie S-atrybutywna

Produkcja	Reguły semantyczne
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \mathbf{int}$	$T.type := integer$
$T \rightarrow \mathbf{real}$	$T.type := real$
$L \rightarrow L_1, \mathbf{id}$	$L_1.in := L.in$ $addtype(\mathbf{id}.entry, L.in)$
$L \rightarrow \mathbf{id}$	$addtype(\mathbf{id}.entry, L.in)$

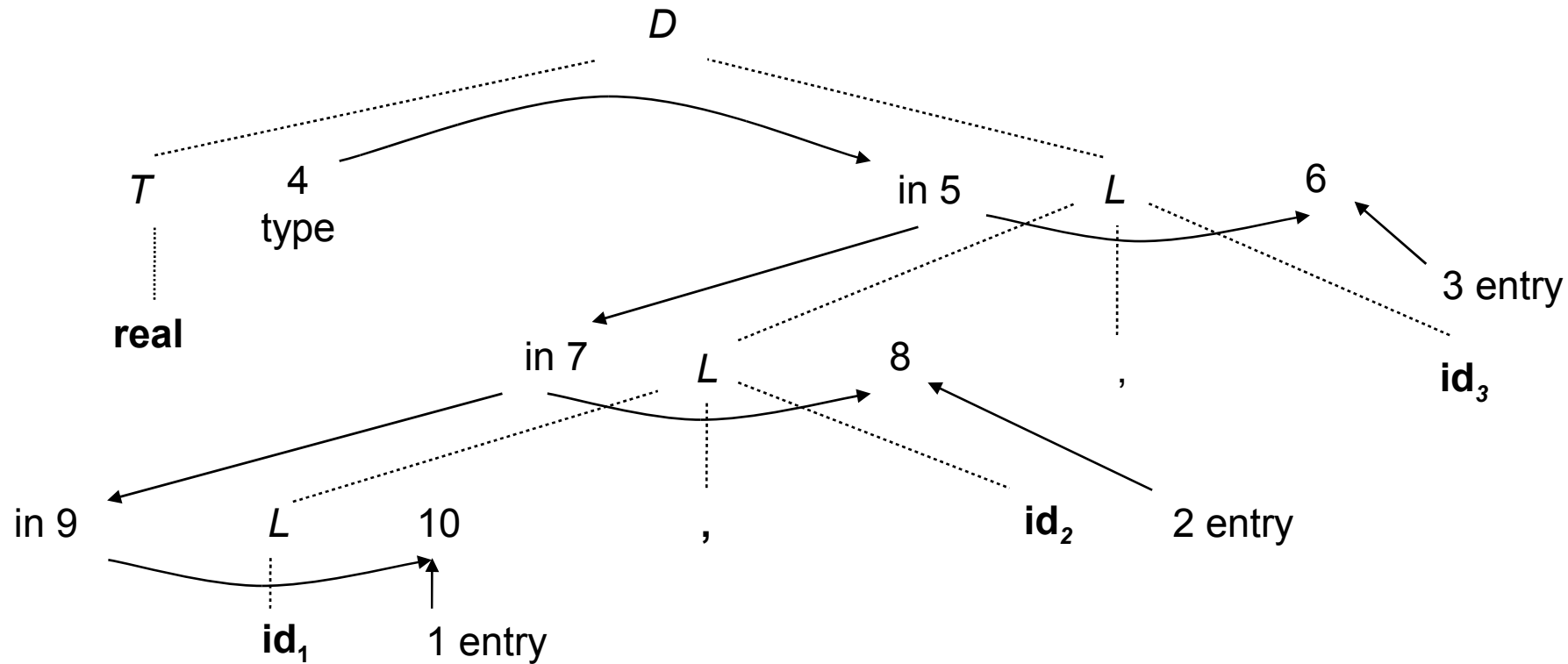


Graf zależności

```
for each node  $n$  in the parse tree do
  for each attribute  $a$  of the grammar symbol at  $n$  do
    construct a node in the dependency graph for  $a$ ;
for each node  $n$  in the parse tree do
  for each semantic rule  $b := f(c_1, c_2, \dots, c_k)$  associated with the production used at  $n$  do
    for  $i := 1$  to  $k$  do
      construct an edge from the node for  $c_i$  to the node for  $b_i$ 
```



Graf zależności



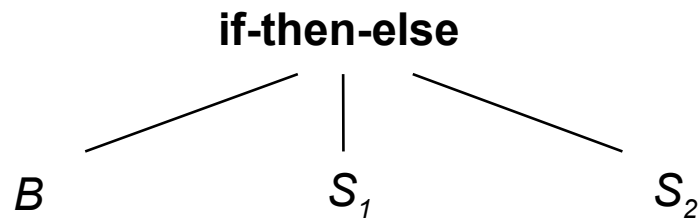
Produkcja	Reguły semantyczne
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow int$	$T.type := integer$
$T \rightarrow real$	$T.type := real$
$L \rightarrow L_1 , id$	$L_1.in := L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

```

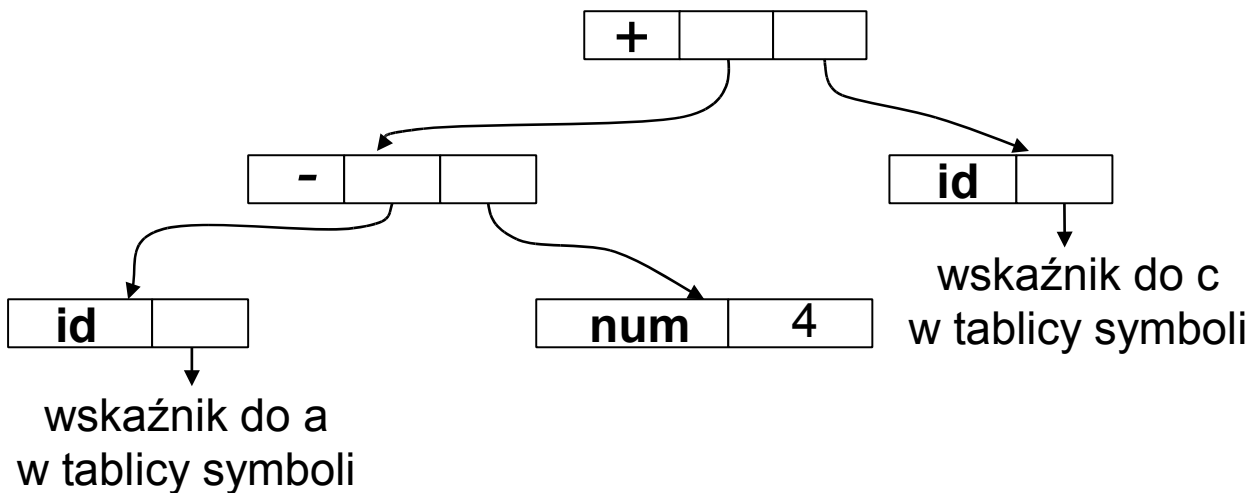
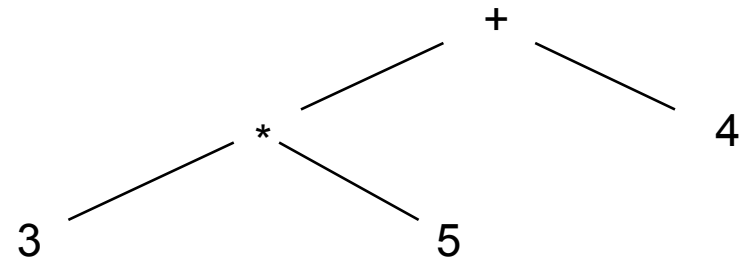
a4 := real;
a5 := a4;
addtype(id3.entry, a5);
a7 := a5;
addtype(id2.entry, a7);
a9 := a7;
addtype(id1.entry, a9);
    
```

Konstrukcja abstrakcyjnych drzew składniowych

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$



$3 * 5 + 4$

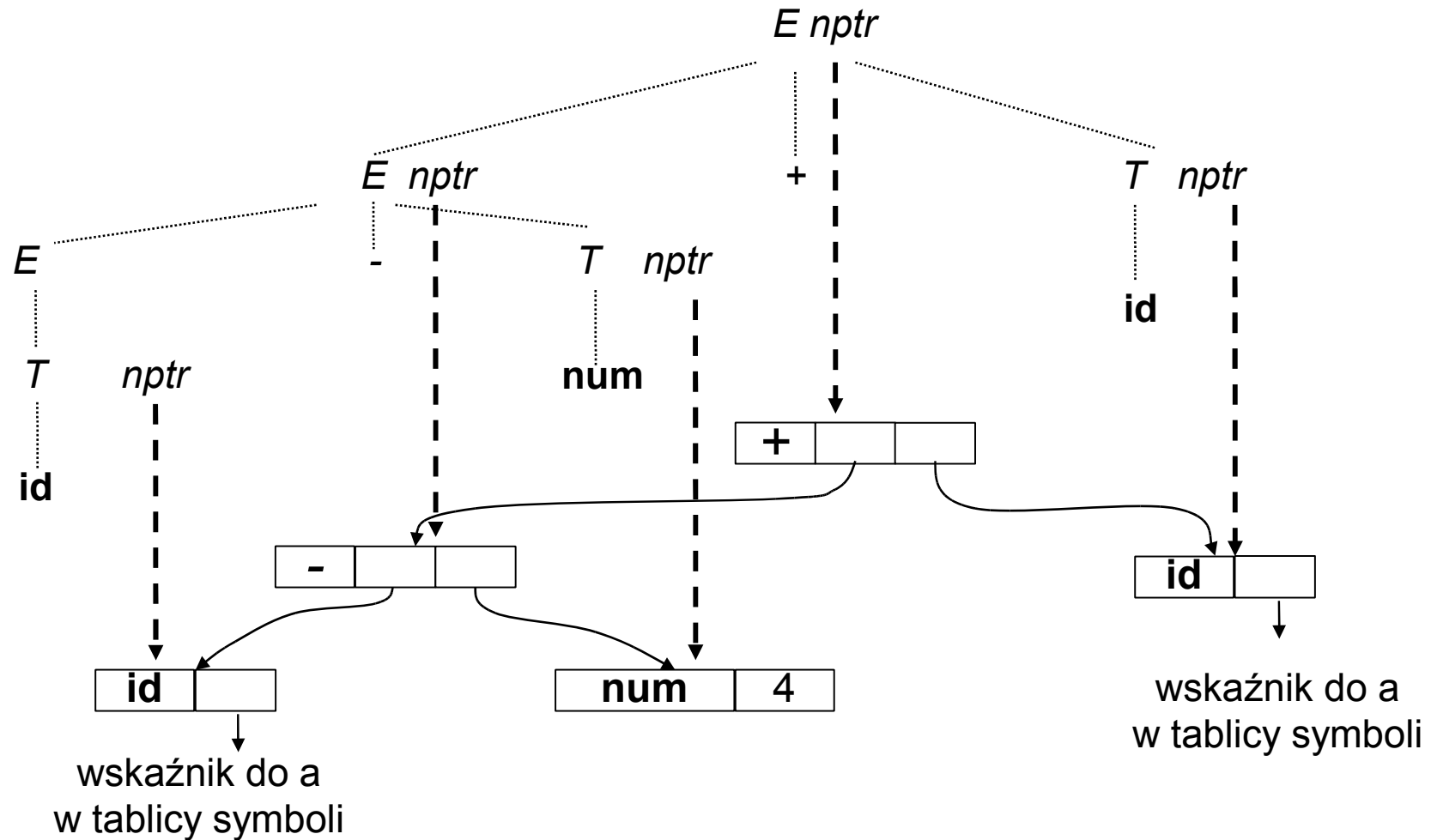


Konstrukcja abstrakcyjnych drzew składniowych

mknnode (*op*, *left*, *right*)
mkleaf (**id**, *entry*)
mkleaf(**num**, *val*)

Produkcja	Reguły semantyczne
$E \rightarrow E_1 + T$	$E.nptr := mknnode ('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknnode ('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow \mathbf{id}$	$T.nptr := mkleaf (\mathbf{id}, \mathbf{id}.entry)$
$T \rightarrow \mathbf{num}$	$T.nptr := mkleaf (\mathbf{num}, \mathbf{num}.val)$

Konstrukcja abstrakcyjnych drzew składniowych



$a - 4 + c$

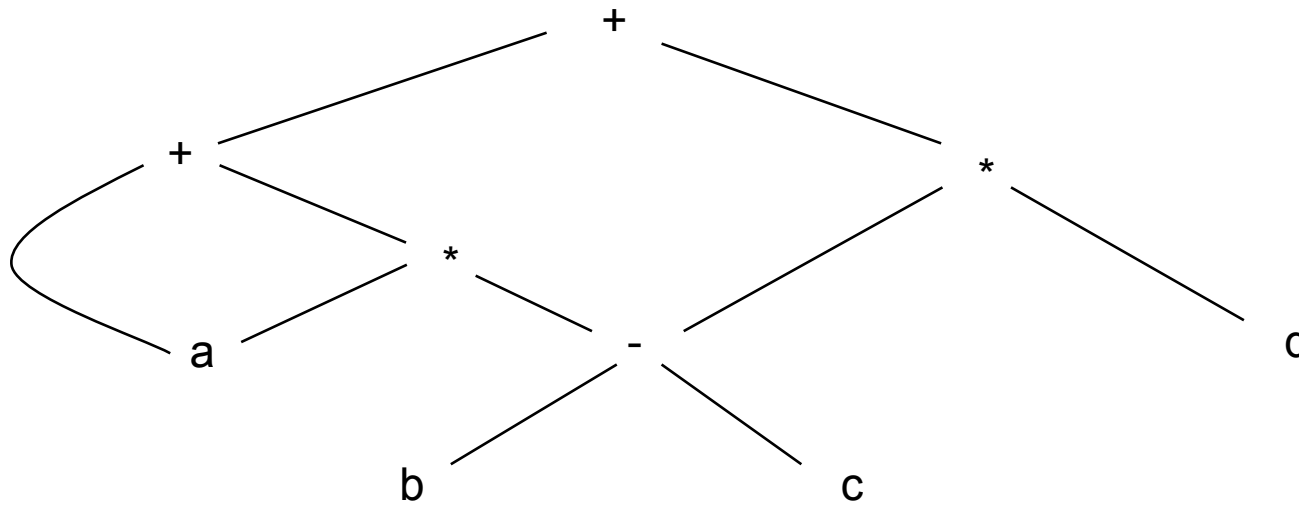
```

p1 := mkleaf (id, entrya);
p2 := mkleaf (num, 4);
p3 := mknnode ('-', p1, p2);
p4 := mkleaf (id, entryc);
p5 := mknnode ('+', p3, p4);
    
```

Skierowany graf acykliczny

DAG – Directed Acyclic Graph

$$a + a * (b - c) + (b - c) * d$$



Atrybuty na stosie parsera

$A \rightarrow XYZ$

top \rightarrow

Symbol	Atrybut
...	...
X	$X.x$
Y	$Y.y$
Z	$Z.z$
...	...

Produkcja	Fragment programu
$L \rightarrow E n$	$print(val[top])$
$E \rightarrow E_1 + T$	$val[ntop] := val[top-2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntop] := val[top-2] * val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntop] := val[top-1]$
$F \rightarrow \mathbf{digit}$	

Atrybuty na stosie parsera

Wejście	Stos	Atrybuty	Produkcje
3 * 5 + 4 n			
* 5 + 4 n	3	3	
* 5 + 4 n	<i>F</i>	3	<i>F</i> → digit
* 5 + 4 n	<i>T</i>	3	<i>T</i> → <i>F</i>
5 + 4 n	<i>T</i> *	3 -	
+ 4 n	<i>T</i> * 5	3 - 5	
+ 4 n	<i>T</i> * <i>F</i>	3 - 5	<i>F</i> → digit
+ 4 n	<i>T</i>	15	<i>T</i> → <i>T</i> * <i>F</i>
+ 4 n	<i>E</i>	15	<i>E</i> → <i>T</i>
4 n	<i>E</i> +	15 -	
n	<i>E</i> + 4	15 - 4	
n	<i>E</i> + <i>F</i>	15 - 4	<i>F</i> → digit
n	<i>E</i> + <i>T</i>	15 - 4	<i>T</i> → <i>F</i>
n	<i>E</i>	19	<i>E</i> → <i>E</i> + <i>T</i>
	<i>E</i> n	19 -	
	<i>L</i>	19	<i>L</i> → <i>E</i> n

Gramatyka L-atrybutywna

- Gramatykę atrybutywną nazywamy L-atrybutywną, jeżeli każdy atrybut dziedziczony symbolu X_j , $1 \leq j \leq n$, po prawej stronie produkcji $A \rightarrow X_1 X_2 \dots X_n$ zależy jedynie od:
 - atrybutów symboli X_1, X_2, \dots, X_{j-1} znajdujących się po lewej stronie X_j w produkcji oraz
 - dziedziczonych atrybutów A

Gramatyka L-atrybutywna i kolejność obliczania atrybutów

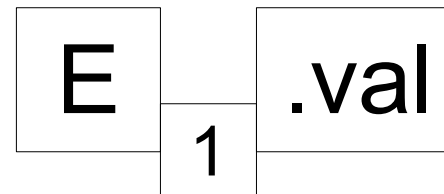
- W gramatyce L-atrybutywnej wszystkie atrybuty można wyliczyć, posługując się następującym algorytmem:

```
procedure dfvisit(n: node);  
begin  
  dla każdego potomka m węzła n,  
  od lewej do prawej do begin  
    oblicz atrybuty dziedziczone m;  
    dfvisit(m)  
  end;  
  oblicz atrybuty syntezowane n  
end
```

Przykład gramatyki L-atrybutywnej

Produkcja	Reguły semantyczne
$S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$
$B \rightarrow B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps := B.ps$ $B_2.ps := \text{shrink}(B.ps)$ $B.ht := \text{disp}(B_1.ht, B_2.ht)$
$B \rightarrow \text{text}$	$B.ht := \text{text.h} * B.ps$

E sub 1 .val



Schemat translacji

$S \rightarrow$ B $\{ B.ps := 10 \}$
 $\{ S.ht := B.ht \}$

$B \rightarrow$ B_1 $\{ B_1.ps := B.ps \}$
 B_2 $\{ B_2.ps := B.ps \}$
 B_2 $\{ B.ht := \max(B_1.ht, B_2.ht) \}$

$B \rightarrow$ B_1 $\{ B_1.ps := B.ps \}$
 B_1 $\{ B_1.ps := B.ps \}$
sub $\{ B_2.ps := \text{shrink}(B.ps) \}$
 B_2 $\{ B.ht := \text{disp}(B_1.ht, B_2.ht) \}$

$B \rightarrow$ **text** $\{ B.ht := \text{text.h} * B.ps \}$

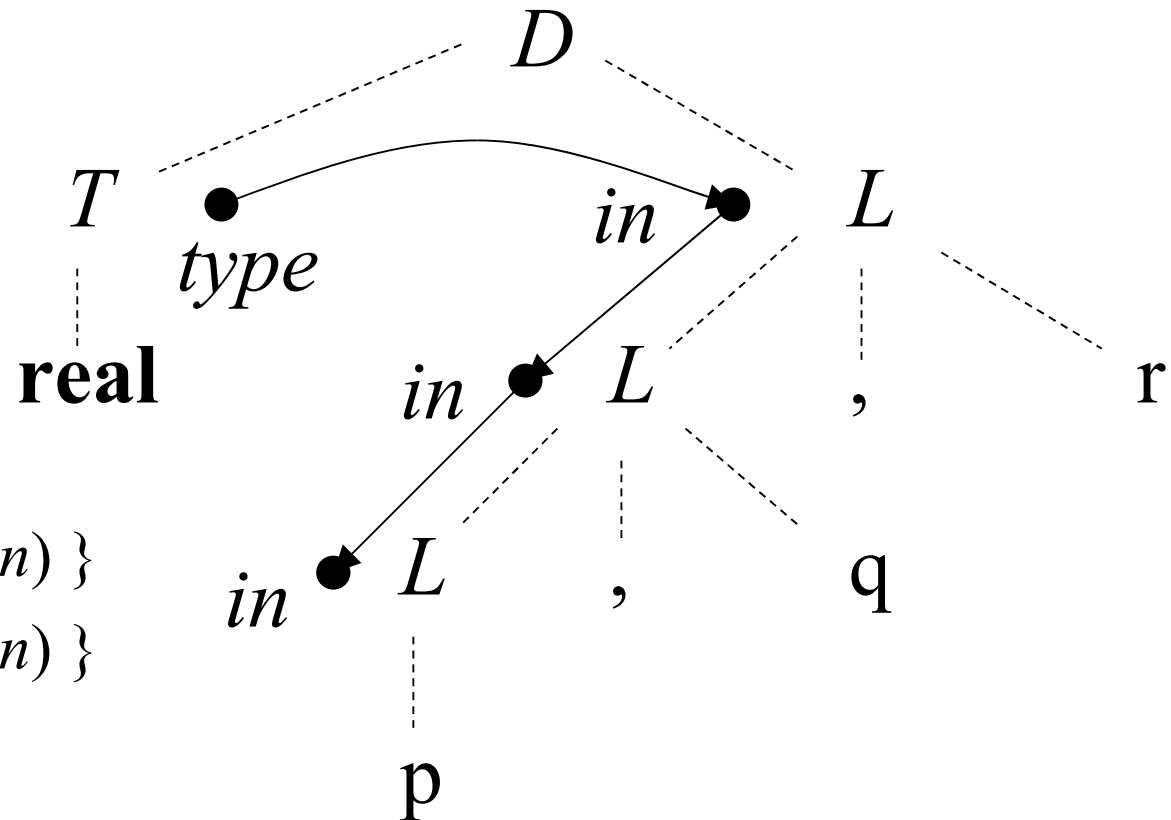
Usuwanie wbudowanych akcji w metodzie wstępującej

$$\begin{aligned} E &\rightarrow T R \\ R &\rightarrow + T \{ \text{print}('+') \} R \mid - T \{ \text{print}('-') \} R \mid \in \\ T &\rightarrow \text{num} \{ \text{print}(\text{num.val}) \} \end{aligned}$$
$$\begin{aligned} E &\rightarrow T R \\ R &\rightarrow + T M R \mid - T N R \mid \in \\ T &\rightarrow \text{num} \{ \text{print}(\text{num.val}) \} \\ M &\rightarrow \in \{ \text{print}('+') \} \\ N &\rightarrow \in \{ \text{print}('-') \} \end{aligned}$$

Atrybuty dziedziczone i metoda wstępująca

Dla tej gramatyki zawsze $L.in = T.type$

$D \rightarrow T \quad \{ L.in := T.type \}$
 L
 $T \rightarrow \mathbf{int} \quad \{ T.type := integer \}$
 $T \rightarrow \mathbf{real} \quad \{ T.type := real \}$
 $L \rightarrow \quad \{ L_1.in := L.in \}$
 $L_1, \mathbf{id} \quad \{ addtype(id.entry, L.in) \}$
 $L \rightarrow \mathbf{id} \quad \{ addtype(id.entry, L.in) \}$



Atrybuty dziedziczone i metoda wstępująca

$D \rightarrow T \quad \{ L.in := T.type \}$
 L
 $T \rightarrow \mathbf{int} \quad \{ T.type := integer \}$
 $T \rightarrow \mathbf{real} \quad \{ T.type := real \}$
 $L \rightarrow \quad \{ L_1.in := L.in \}$
 $L_1, \mathbf{id} \quad \{ addtype(id.entry, L.in) \}$
 $L \rightarrow \mathbf{id} \quad \{ addtype(id.entry, L.in) \}$

Stan	Wejście	Produkcja
-	real p, q, r	
real	p, q, r	
T	p, q, r	$T \rightarrow \mathbf{real}$
T p	, q, r	
T L	, q, r	$L \rightarrow \mathbf{id}$
T L,	q, r	
T L, q	, r	
T L	, r	$L \rightarrow L, \mathbf{id}$
T L,	r	
T L, r		
T L		$L \rightarrow L, \mathbf{id}$
D		$D \rightarrow TL$

T jest zawsze pod L na stosie parsera

Atrybuty dziedziczone i metoda wstępująca

$$\begin{aligned}
 D \rightarrow T L & \quad \{ L.in := T.type \} \\
 T \rightarrow \mathbf{int} & \quad \{ T.type := integer \} \\
 T \rightarrow \mathbf{real} & \quad \{ T.type := real \} \\
 L \rightarrow L_1, \mathbf{id} & \quad \{ addtype(id.entry, L.in) \} \\
 L \rightarrow \mathbf{id} & \quad \{ addtype(id.entry, L.in) \}
 \end{aligned}$$

Produkcja	Fragment programu
$D \rightarrow T L ;$	
$T \rightarrow \mathbf{int}$	$val[ntop] := integer$
$T \rightarrow \mathbf{real}$	$val[ntop] := real$
$L \rightarrow L , \mathbf{id}$	$addtype(val[top], val[top-3])$
$L \rightarrow \mathbf{id}$	$addtype(val[top], val[top-1])$

Atrybuty dziedziczone – zastosowanie markera

$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABC$	$C.i := A.s$
$C \rightarrow c$	$C.s := g(C.i)$

$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABMC$	$M.i := A.s; C.i := M.s$
$C \rightarrow c$	$C.s := g(C.i)$
$M \rightarrow \epsilon$	$M.s := M.i$

Zastosowanie markera – reguła inna, niż reguła kopiowania

$S \rightarrow aAC \quad C.i := f(A.s)$

$S \rightarrow aANC \quad N.i := A.s; C.i := N.s$
 $N \rightarrow \epsilon \quad N.s := f(N.i)$

Przetwarzanie wstępujące gramatyki L-atrybutywnej

$S \rightarrow$	$L B$	$B.ps := L.s$ $S.ht := B.ht$
$L \rightarrow$	ϵ	$L.s := 10$
$B \rightarrow$	$B_1 M B_2$	$B_1.ps := B.ps$ $M.i := B.ps$ $B_2.ps := M.s$ $B.ht := \max(B_1.ht, B_2.ht)$
$B \rightarrow$	$B_1 \mathbf{sub} N B_2$	$B_1.ps := B.ps$ $N.i := B.ps$ $B_2.ps := N.s$ $B.ht := \mathit{disp}(B_1.ht, B_2.ht)$
$B \rightarrow$	\mathbf{text}	$B.ht := \mathbf{text.h} * B.ps$
$M \rightarrow$	ϵ	$M.s := M.i$
$N \rightarrow$	ϵ	$N.s := \mathit{shrink}(N.i)$

Analiza semantyczna

- Kontrola typów
- Właściwy przepływ sterowania
- Sprawdzenie niepowtarzalności nazwy
- Różne inne testy związane z nazwami

Wyrażenia określające typ

- Typ podstawowy: *boolean, char, integer, real*
- Typ nazwany (typedef)
- Typ konstruowany:

- Tablice: *array(I, T)*

```
var A: array[1..10] of integer;  
array(1..10, integer)
```

- Iloczyn kartezjański

$T_1 \times T_2$

- Rekord

```
type row = record  
  address: integer;  
  lexeme: array [1..15] of char  
end;  
var table: array [1..101] of row;
```

record((address \times integer) \times (lexeme \times array(1..15, char)))

- Wskaźniki

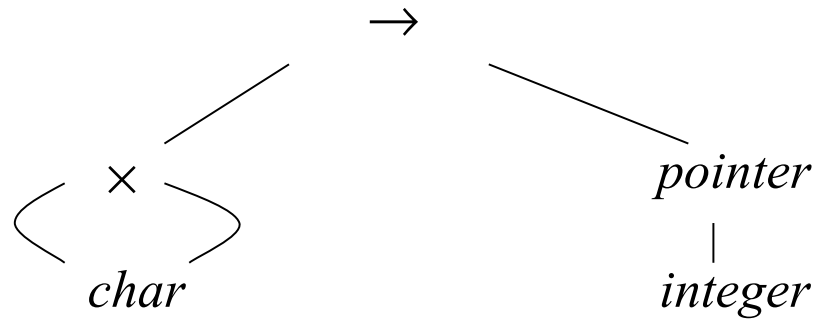
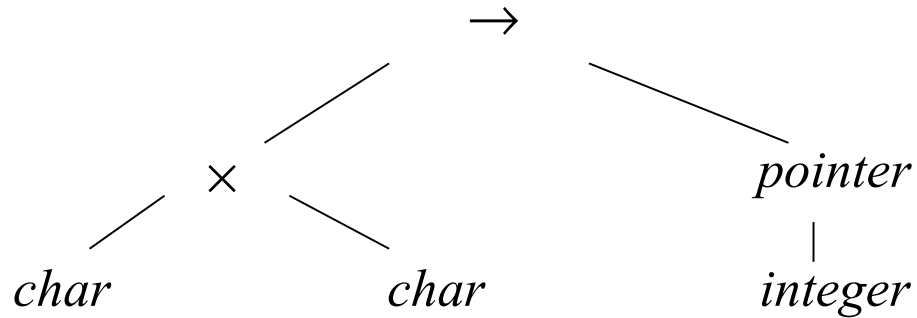
```
var p: ^row  
pointer(row)
```

- Funkcje

```
function f(a, b: char) : ^integer;  
char  $\times$  char  $\rightarrow$  pointer(integer)
```

- Zmienne

Reprezentacja typu w postaci drzewa i DAGu



Prosty system typów

$P \rightarrow D ; E$
 $D \rightarrow D ; D \mid \mathbf{id} : T$
 $T \rightarrow \mathbf{char} \mid \mathbf{integer} \mid \mathbf{array} [\mathbf{num}] \mathbf{of} T \mid \wedge T$
 $E \rightarrow \mathbf{literal} \mid \mathbf{num} \mid \mathbf{id} \mid E \mathbf{mod} E \mid E [E] \mid E \wedge$

$P \rightarrow D ; E$
 $D \rightarrow D ; D$
 $D \rightarrow \mathbf{id} : T \quad \{ \mathit{addtype}(\mathbf{id.entry}, T.type) \}$
 $T \rightarrow \mathbf{char} \quad \{ T.type := \mathit{char} \}$
 $T \rightarrow \mathbf{integer} \quad \{ T.type := \mathit{integer} \}$
 $T \rightarrow \wedge T_1 \quad \{ T.type := \mathit{pointer}(T_1.type) \}$
 $T \rightarrow \mathbf{array} [\mathbf{num}] \mathbf{of} T_1 \quad \{ T.type := \mathit{array}(1..\mathbf{num.val}, T_1.type) \}$

Kontrola typów

$E \rightarrow$ **literal** { $E.type := char$ }

$E \rightarrow$ **num** { $E.type := integer$ }

$E \rightarrow$ **id** { $E.type := lookup(id.entry)$ }

$E \rightarrow$ E_1 **mod** E_2 { $E.type :=$ **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then**
integer **else** *type_error* }

$E \rightarrow$ E_1 [E_2] { $E.type :=$ **if** $E_2.type = integer$ **and** $E_1.type = array(s,t)$ **then**
t **else** *type_error* }

$E \rightarrow$ E_1 **^** { $E.type :=$ **if** $E_1.type = pointer(t)$ **then** *t* **else** *type_error* }

Kontrola typów dla instrukcji

```
P → D ; S  
S → id := E { S.type := if id.type = E.t then void else type_error }  
S → if E then S1 { S.type := if E.type = boolean then S1.type else type_error }  
S → while E do S1 { S.type := if E.type = boolean then S1.type else type_error }  
S → S1 ; S2 { S.type := if S1.type = void and S2.type = void then void else type_error }
```

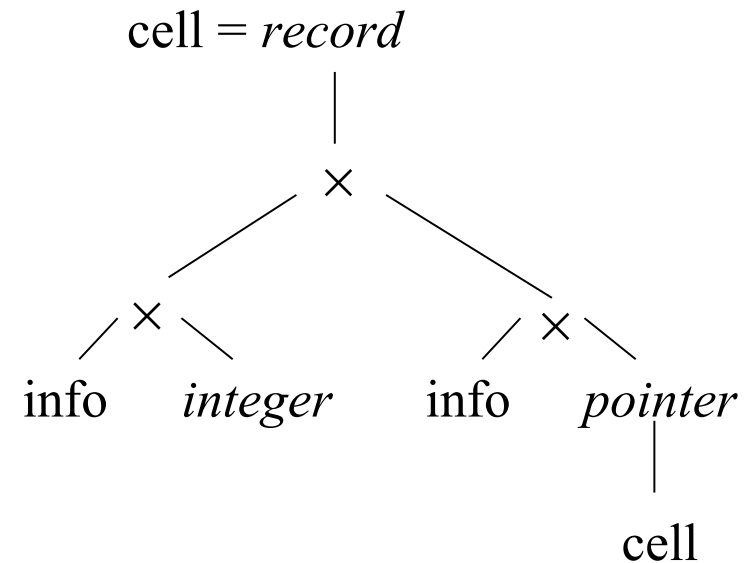
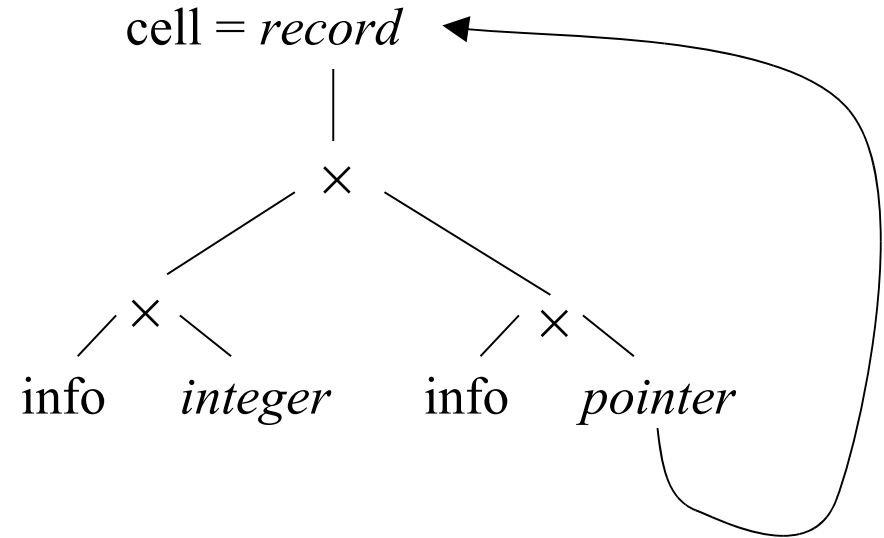
Kontrola typów dla funkcji

```
E → E ( E )  
T → T1 '→ ' T2 { T.type := T1.type → T2.type }  
E → E1 ( E2 ) { E.type := if E2.type = s and E1.type = s → t then t else type_error }
```

Równoważność strukturalna i równoważność nazw

```
type link = ^ cell;  
var next : link;  
    last : link;  
    p    : ^ cell;  
    q, r : ^ cell;  
  
cell = record  
    info : integer;  
    next : link  
end;
```

```
struct cell {  
    int info;  
    struct cell * next;  
};
```



Konwersja typów

x + i

Notacja postfiksowa:

x i inttoreal real+

$E \rightarrow$ **num** { $E.type := integer$ }

$E \rightarrow$ **num.num** { $E.type := real$ }

$E \rightarrow$ **id** { $E.type := lookup(id.entry)$ }

$E \rightarrow$ E_1 **op** E_2 { $E.type :=$

if $E_1.type = integer$ and $E_2.type = integer$ then integer else

if $E_1.type = integer$ and $E_2.type = real$ then real else

if $E_1.type = real$ and $E_2.type = integer$ then real else

if $E_1.type = real$ and $E_2.type = real$ then real

else type_error }

Środowisko czasu wykonania

```
program sort(input, output);
  var a : array [0..10] of integer;
  procedure readarray;
  var i : integer;
  begin
    for i := 1 to 9 read(a[i])
  endl

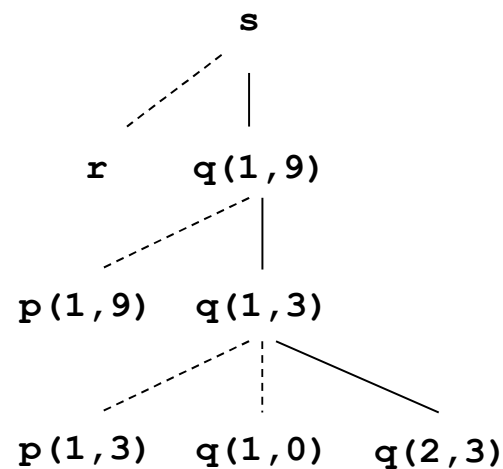
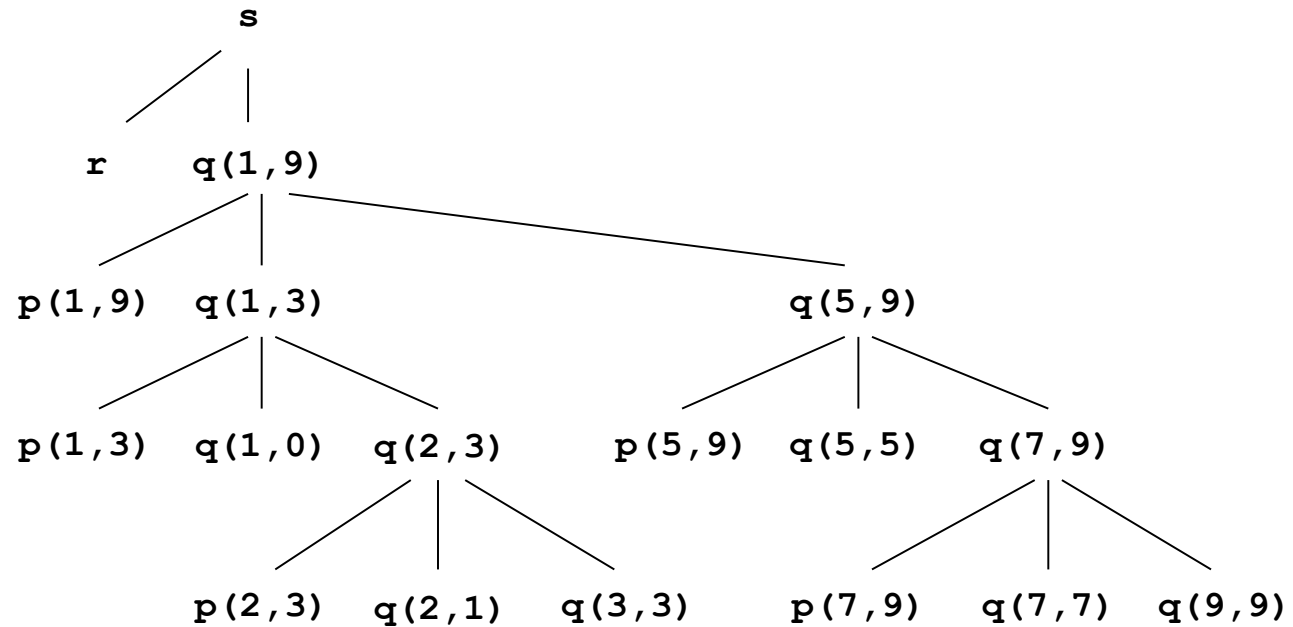
  function partition(y, z: integer) : integer;
  var i, j, x, v: integer;
  begin ...
  end;

  procedure quicksort(m, n integer);
  var i : integer;
  begin
    if ( n > m ) then begin
      i := partition(m,n);
      quicksort(m,i-1);
      quicksort(i+1,n);
    end
  end;

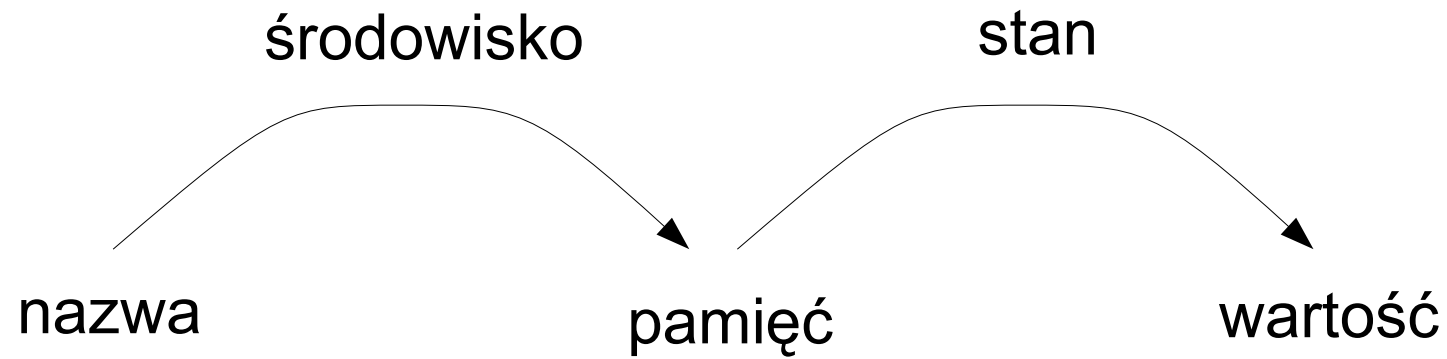
begin
  a[0] := -9999; a[10] := 999;
  readarray;
  quicksort(1,9);
end.
```

Drzewo aktywacji

Execution begins...
 enter readarray
 leave readarray
 enter quicksort(1,9)
 enter partition(1,9)
 leave partition(1,9)
 enter quicksort(1,3)
 ...
 leave quicksort(1,3)
 enter quicksort(5,9)
 ...
 leave quicksort(5,9)
 leave quicksort(1,9)
 Execution terminated.



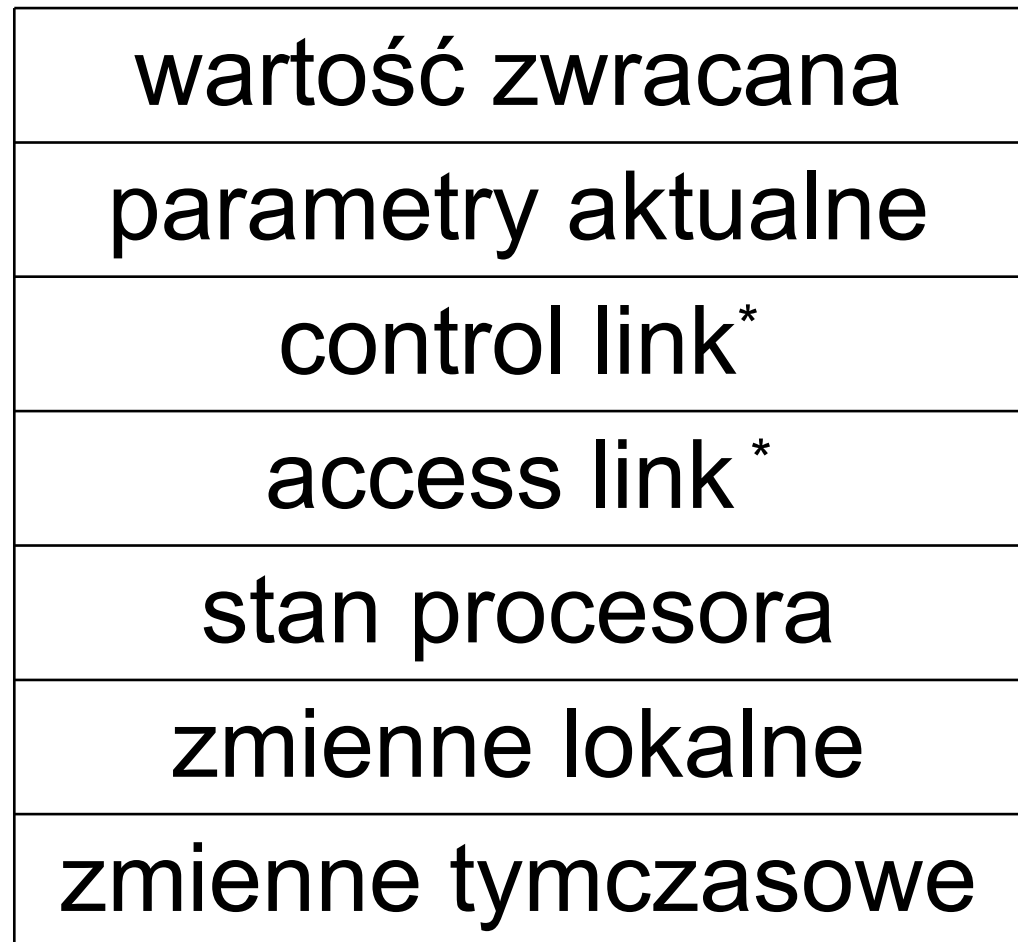
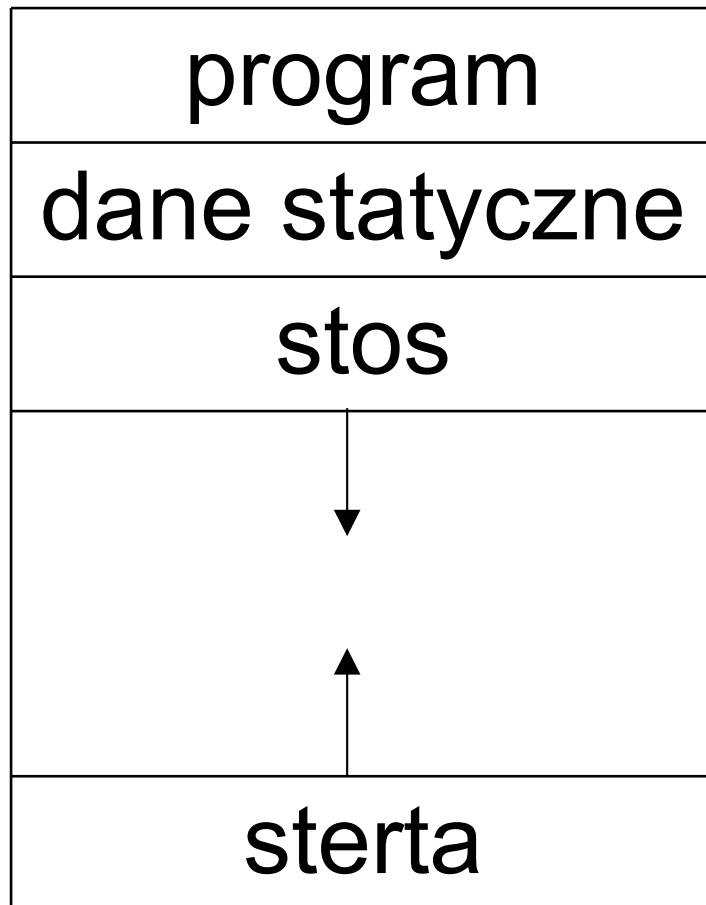
Dowiązanie nazw



Czynniki wpływające na organizację środowiska czasu wykonania

- Czy procedury mogą być rekursywnie wywoływane
- Co dzieje się ze zmiennymi lokalnymi po wyjściu z procedury
- Czy procedura ma dostęp do zmiennych nielokalnych
- Jak są przekazywane parametry do procedur
- Czy procedury mogą być przekazywane jako parametry
- Czy procedury mogą być zwracane jako wyniki
- Czy jest możliwa dynamiczna alokacja pamięci pod kontrolą programisty
- Czy pamięć musi być jawnie zwalniana

Organizacja pamięci, rekord aktywacji



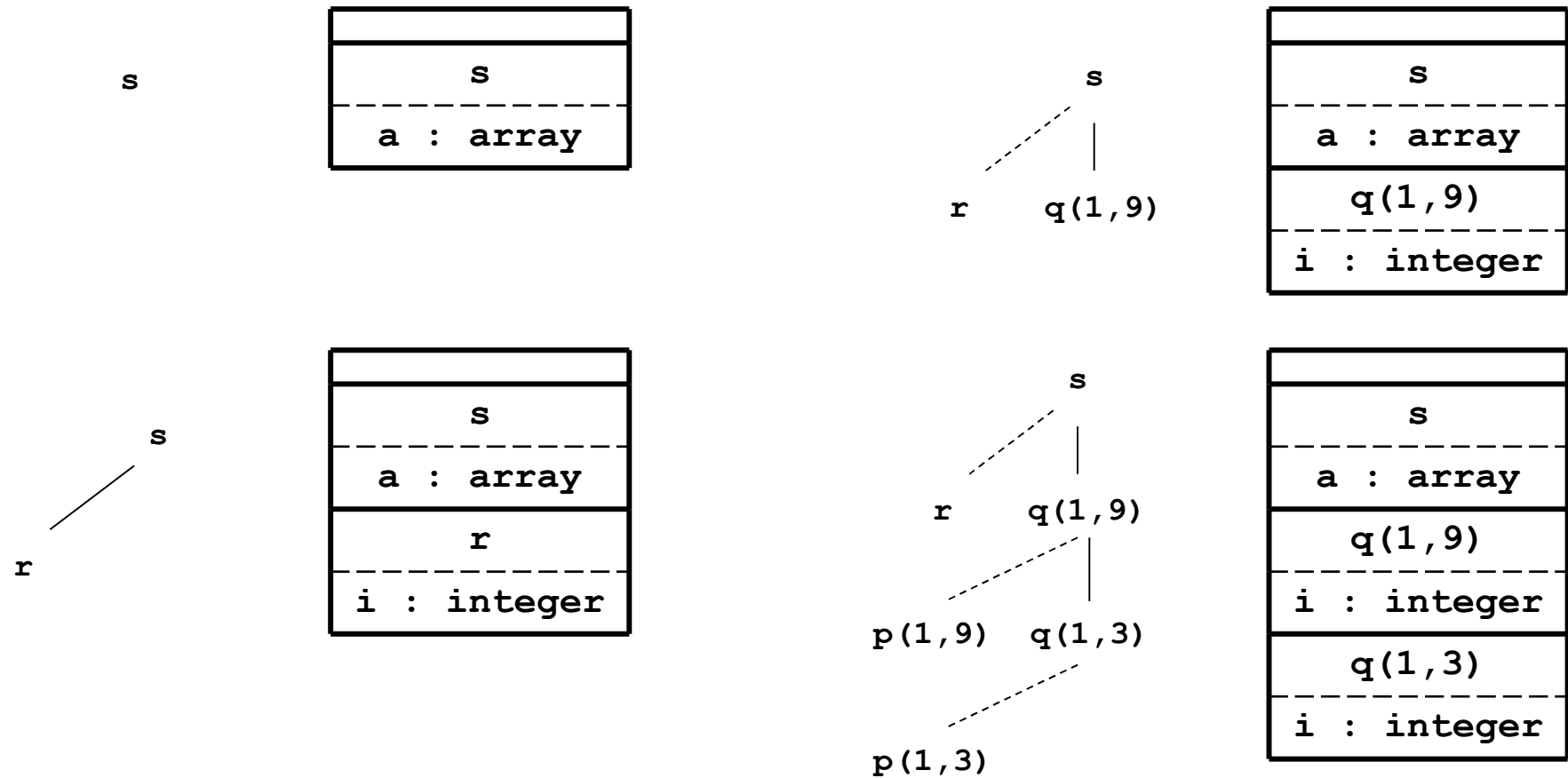
*opcjonalnie

Rozmieszczenie danych w różnych procesorach

Typ	Rozmiar (bity)		Wyrównanie (bity)	
	Procesor 1	Procesor 2	Procesor 1	Procesor 2
char	8	8	8	64*
short	16	24	16	64
int	32	48	32	64
long	32	64	32	64
float	32	64	32	64
double	64	128	32	64
char*	32	30	32	64
inne wsk.	32	24	32	64
struktury	≥ 8	≥ 64	32	64

***8 bitów w przypadku tablic znaków**

Stos rekordów aktywacji



Zmienne lokalne instrukcji złożonych

```
int main()
{
    int a = 0;
    int b = 0;
    {
        int b = 1;
        {
            int a = 2;
            printf("%d %d\n", a, b);
        }
        {
            int b = 3;
            printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
};
```

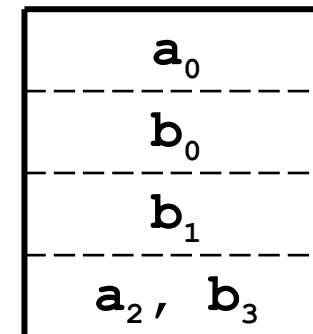
B_0

B_1

B_2

B_3

Deklaracja	Zasięg
<code>int a = 0;</code>	$B_0 - B_2$
<code>int b = 0;</code>	$B_0 - B_1$
<code>int b = 1;</code>	$B_1 - B_3$
<code>int a = 2;</code>	B_2
<code>int b = 3;</code>	B_3



Dostęp do zmiennych nielokalnych

```
program sort(input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
    var i : integer;
    begin ... a ... end { readarray } ;

  procedure exchange( i, j: integer);
    begin
      x := a[i]; a[i] := a[j]; a[j] := x
    end { exchange } ;

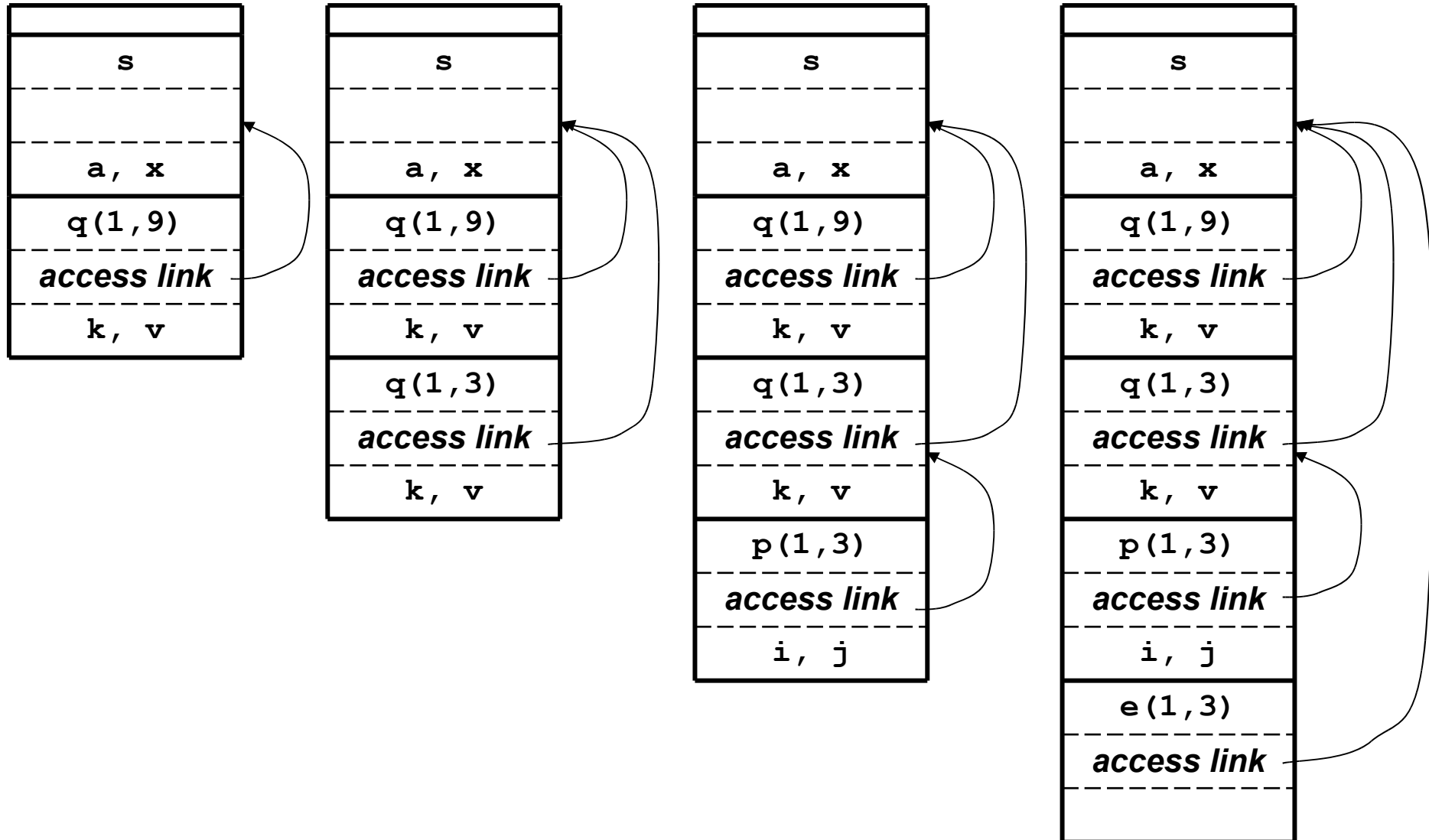
  procedure quicksort( m, n: integer);
    var k, v : integer;

    function partition( y, z: integer): integer;
      var i, j: integer;
      begin ... a ...
            ... v ...
            ... exchange(i,j); ...
      end { partition }

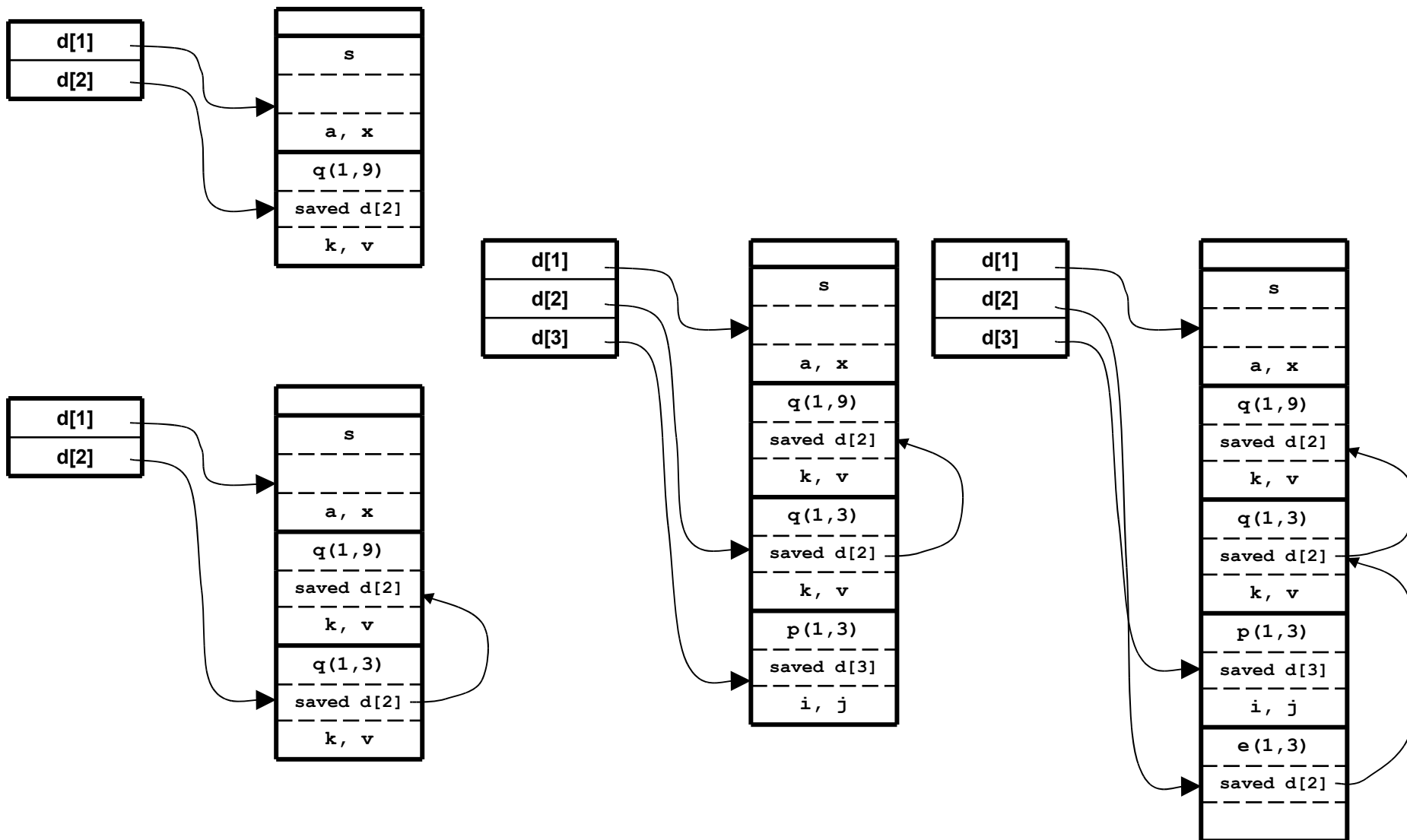
    begin ... end { quicksort };

begin ... end { sort };
```

Dostęp do zmiennych nielokalnych



Dostęp do zmiennych nielokalnych - display



Procedury jako parametry

```
Program param(input, output);  
  
    procedure b(function h(n:integer): integer);  
    begin writeln(h(2)) end;  
  
    procedure c;  
    var m : integer;  
  
    function f(n : integer): integer;  
        begin f := m + n end ;  
  
    begin m := 0; b(f) end ;  
  
begin  
c  
end;
```

Aby przekazywanie procedur jako argumentów działało poprawnie, wraz z procedurą należy przekazać jej access link.

Zasięg dynamiczny

```
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.
```

Przekazywanie parametrów

Referencja

```
program reference(input, output);
var a,b : integer;
procedure swap (var 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); writeln('b=',b)
end.
```

Nazwa

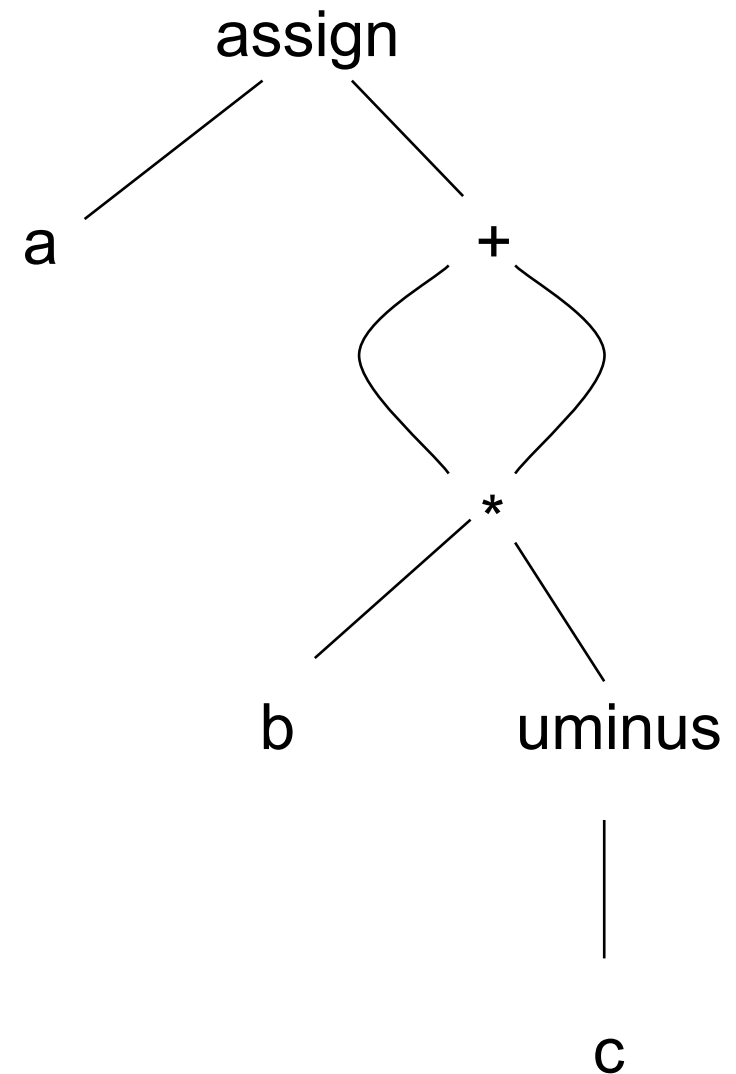
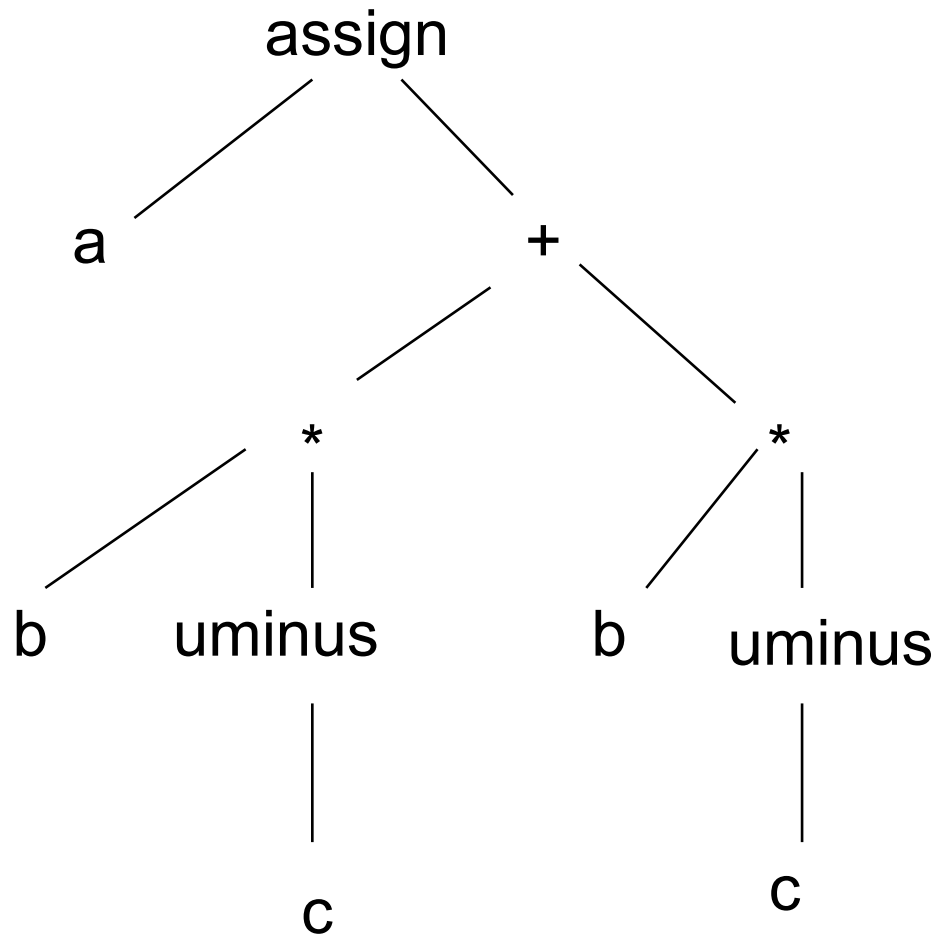
```
swap(i,a[i])
```

```
temp:=i;
i:=a[i];
a[i]=temp;
```

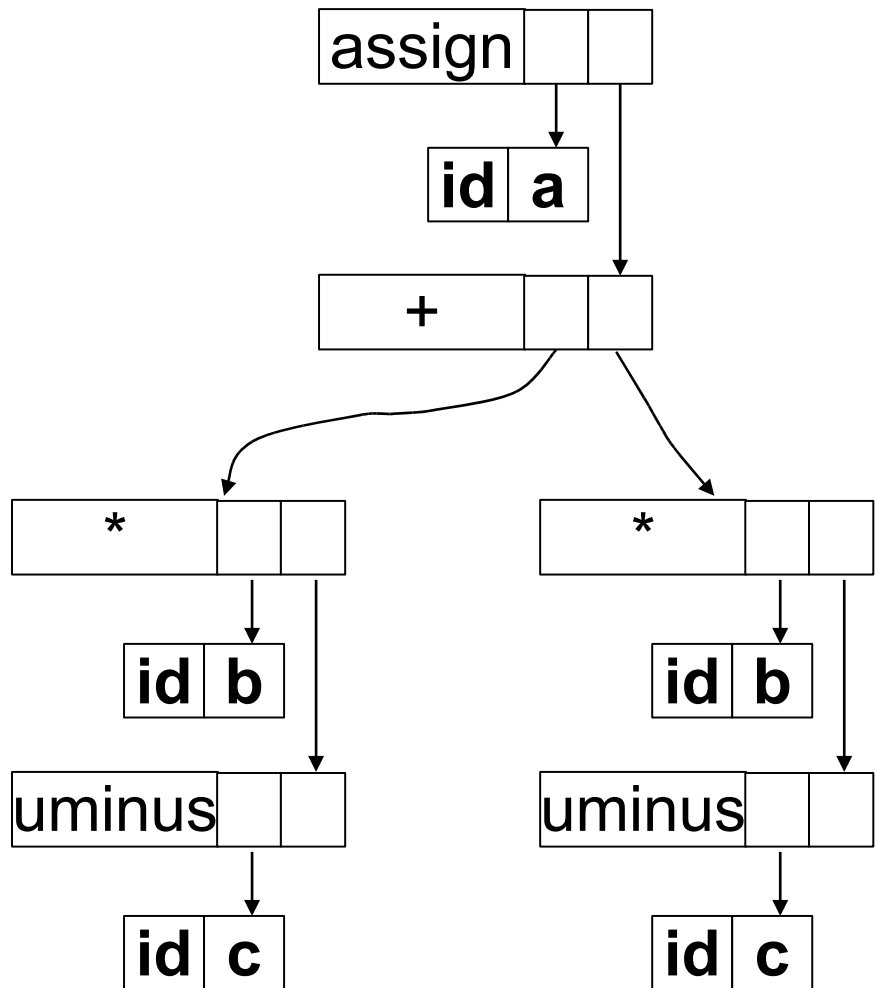
Copy-restore

```
program copyout(input, output);
var a: integer;
    procedure unsafe(var x: integer);
    begin x:=2; a:=0 end;
begin
    a:=1; unsafe(a); writeln(a)
end.
```

Kod pośredni w postaci drzewa



Dwie reprezentacje drzewa składniowego



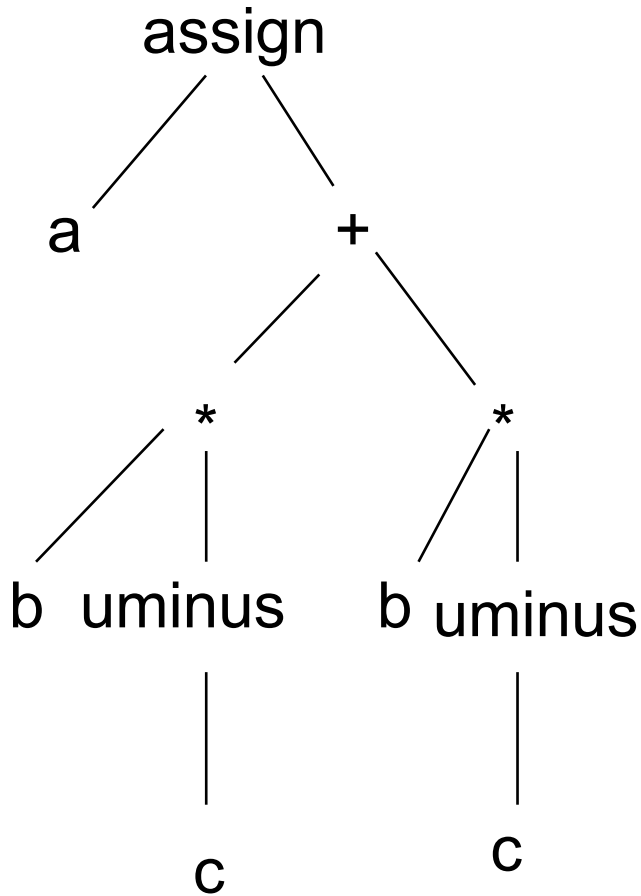
0	id	b		
1	id	c		
2	uminus		1	
3	*		0	2
4	id	b		
5	id	c		
6	uminus		5	
7	*		4	6
8	+		3	7
9	id	a		
10	assign		9	8
11	...			

Kod trzyadresowy

`x := y op z`

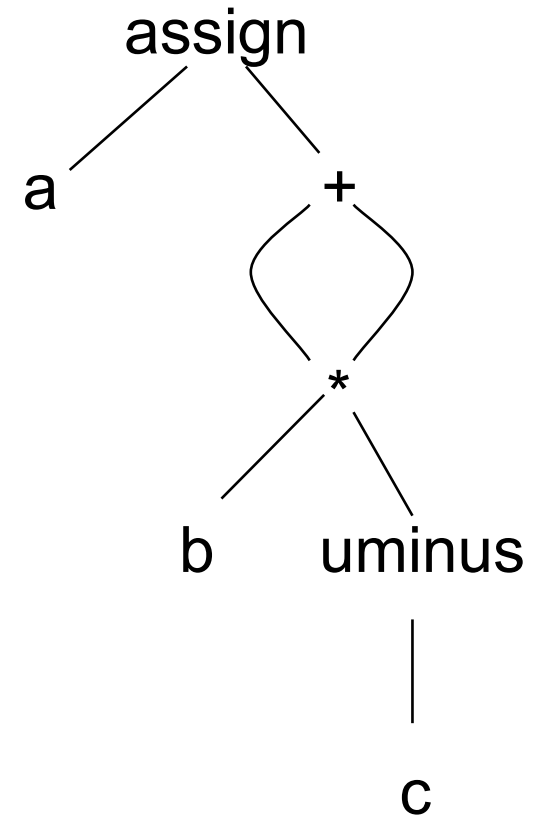
Drzewo

Dag



```
t1 := -c  
t2 := b * t1  
t3 := -c  
t4 := b * t3  
t5 := t2 + t4  
a := t5
```

```
t1 := -c  
t2 := b * t1  
t5 := t2 + t2  
a := t5
```



Typy instrukcji kodu trzyadresowego

- Instrukcje $x:=y \text{ op } z$
- Instrukcje $x:=\text{op } y$
- Instrukcje kopiowania $x:=y$
- Skoki warunkowe
if x rel op y goto L
- wywołanie procedur
param x_1
param x_2
...
param x_n
call p,n
- Instrukcje $x:=y[i], x[i]:=y$
- Operacje na wskaźnikach
 $x:=\&y, x:=*y$

Generacja kodu dla wyrażeń

Produkcja	Reguły semantyczne
$S \rightarrow \mathbf{id} := E$	$S.code := E.code \text{gen}(\mathbf{id.place} := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code E_2.code \text{gen}(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code E_2.code \text{gen}(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := \text{newtemp};$ $E.code := E_1.code \text{gen}(E.place := \text{'uminus'} E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \mathbf{id}$	$E.place := \mathbf{id.place}$ $E.code := ''$

Generacja kodu dla instrukcji *while*

Produkcja	Reguły semantyczne
$S \rightarrow \text{while } E \text{ do } S_1$	<pre> S.begin:=newlabel; S.after:=newlabel; S.code:=gen(S.begin ':') E.code gen('if' E.place '=' '0' 'goto' S.after) S₁.code gen('goto' S.begin) gen(S.after ':') </pre>

S.begin :

<i>E.code</i>
if <i>E.place</i> = 0 goto <i>S.after</i>
<i>S₁.code</i>
goto <i>S.begin</i>

S.after :

Reprezentacja kodu trzyadresowego w postaci czwórek i trójek

	op	arg1	arg2	result
(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

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

$x[i] := y$

	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

$x := y[i]$

Reprezentacja kodu trzyadresowego w postaci trójek pośrednich

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

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Deklaracje

$P \rightarrow \{ \textit{offset} := 0 \} D$

$D \rightarrow D ; D$

$D \rightarrow \mathbf{id} : T \quad \{ \textit{enter}(\mathbf{id.name}, T.type, \textit{offset});$
 $\quad \textit{offset} := \textit{offset} + T.width \}$

$T \rightarrow \mathbf{char} \quad \{ T.type := \textit{char}; T.width := 1 \}$

$T \rightarrow \mathbf{integer} \quad \{ T.type := \textit{integer}; T.width := 4 \}$

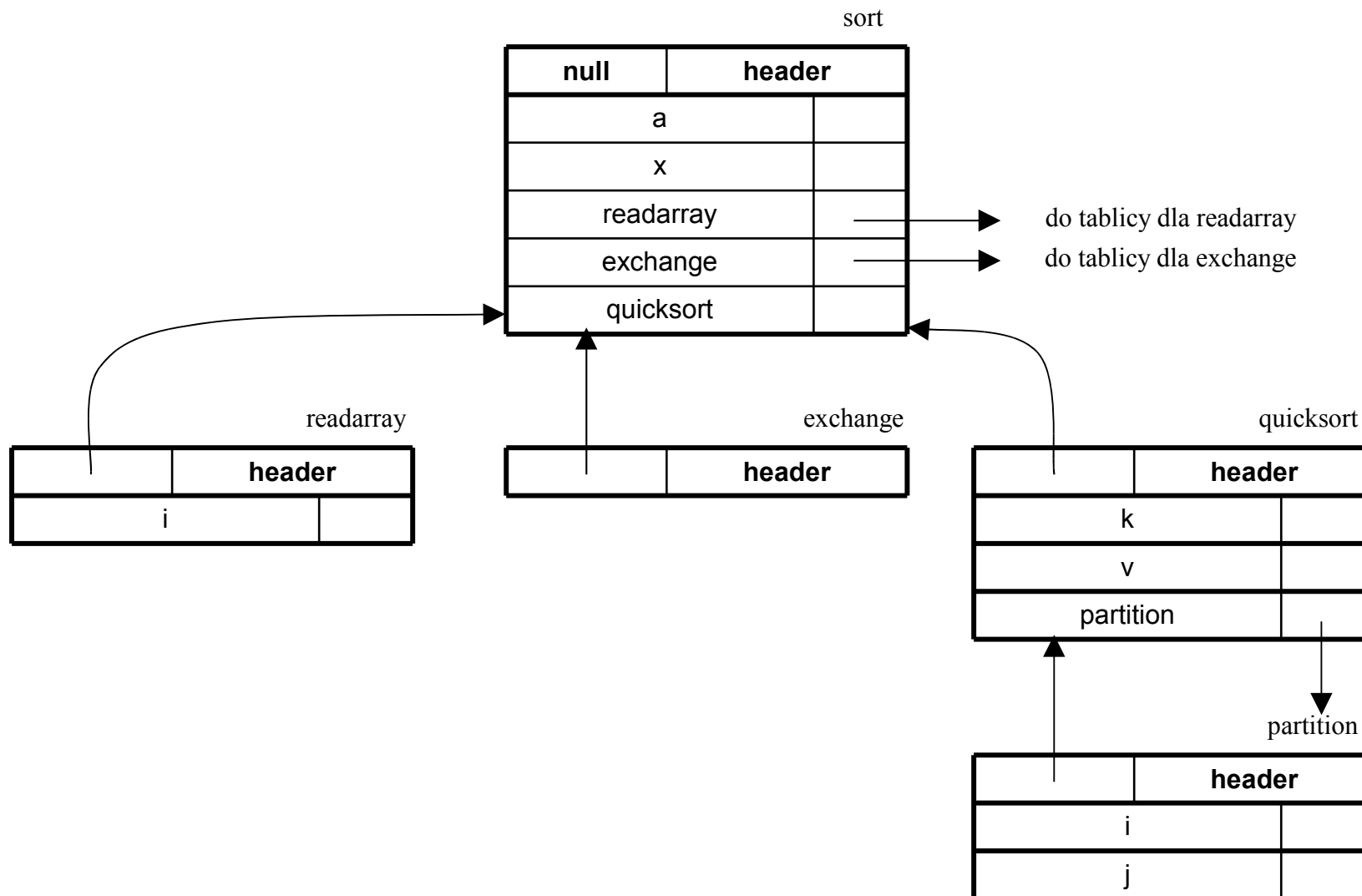
$T \rightarrow \mathbf{real} \quad \{ T.type := \textit{real}; T.width := 8 \}$

$T \rightarrow \wedge T_1 \quad \{ T.type := \textit{pointer}(T_1.type); T.width := 4 \}$

$T \rightarrow \mathbf{array} [\mathbf{num}] \mathbf{of} T_1$
 $\quad \{ T.type := \textit{array}(1..\mathbf{num.val}, T_1.type);$
 $\quad \quad T.width := \mathbf{num.val} * T_1.width \}$

name	type	offset
a	<i>char</i>	0
b	<i>pointer(real)</i>	1
c	<i>real</i>	5

Tablica symboli dla zagnieżdżonych procedur



Tablica symboli dla zagnieżdżonych procedur

Produkcja	Reguły semantyczne
$P \rightarrow MD$	addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset)
$M \rightarrow \epsilon$	t:=mhtable(nil); push(t, tblptr); push(0,offset)
$D \rightarrow D_1; D_2$	
$D \rightarrow \mathbf{proc\ id}; ND_1; S$	t:=top(tblptr); addwidth(t,top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.name , t)
$D \rightarrow \mathbf{id}: T$	enter(top(tblptr), id.name , T.type, top(offset)); top(offset) := top(offset)+T.width
$N \rightarrow \epsilon$	t:=mhtable(top(tblptr)); push(t,tblptr); push(0,offset)

Generacja kodu - wyrażenia

Produkcja	Reguły semantyczne
$S \rightarrow \text{id} := E$	<pre>p:=lookup(id.name); if p<> nil then emit(p ':=' E.place) else error</pre>
$E \rightarrow E_1 + E_2$	<pre>E.place:=newtemp; emit(E.place ':=' E₁.place '+' E₂.place)</pre>
$E \rightarrow E_1 * E_2$	<pre>E.place:=newtemp; emit(E.place ':=' E₁.place '*' E₂.place)</pre>
$E \rightarrow - E_1$	<pre>E.place:=newtemp; emit(E.place ':=' 'uminus' E₁.place)</pre>
$E \rightarrow (E_1)$	<pre>E.place := E₁.place</pre>
$E \rightarrow \text{id}$	<pre>p:= lookup(id.name); if p<> nil then E.place:=p else error</pre>

Adresowanie elementów tablic

$$A[\text{low}..\text{high}] \quad A[i] \quad \text{base} + (i - \text{low}) * w$$

$$i * w + (\text{base} - \text{low} * w)$$

$$A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2] \quad A[i_1, i_2] \quad n_2 = \text{high}_2 - \text{low}_2 + 1$$

$$\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

$$((i_1 * n_2 + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w))$$

Konwersja typów

```
E → E1 + E2 { E.place := newtemp;  
  if E1.type = integer and E2.type = integer then begin  
    emit (E.place := E1.place 'int+' E2.place);  
    E.type := integer;  
  end  
  else if E1.type = real and E2.type = real then begin  
    emit (E.place := E1.place 'real+' E2.place);  
    E.type := real;  
  end  
  else if E1.type = integer and E2.type = real then begin  
    u := newtemp;  
    emit (u := 'inttoreal' E1.place);  
    emit (E.place := u 'real+' E2.place);  
    E.type := real;  
  end  
  else if E1.type = real and E2.type = integer then begin  
    u := newtemp;  
    emit (u := 'inttoreal' E2.place);  
    emit (E.place := E1.place 'real+' u);  
    E.type := real;  
  end  
  else  
    E.type := type_error  
}
```

```
x := y + i * j
```

```
t1 := i int* j
```

```
t3 := inttoreal t1
```

```
t2 := y real+ t3
```

```
x := t2
```

Wyrażenia logiczne

$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$

a or b and not c

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

a < b \Rightarrow if a < b then 1 else 0

100: if a < b goto 103

101: t := 0

102: goto 104

103: t := 1

104:

Wyrażenia logiczne

$E \rightarrow E_1 \text{ or } E_2$ { $E.place := newtemp; emit (E.place := E_1.place \text{ 'or' } E_2.place)$ }
 $E \rightarrow E_1 \text{ and } E_2$ { $E.place := newtemp; emit (E.place := E_1.place \text{ 'and' } E_2.place)$ }
 $E \rightarrow \text{not } E_1$ { $E.place := newtemp; emit (E.place := \text{ 'not' } E_1.place)$ }
 $E \rightarrow (E_1)$ { $E.place := E_1.place$ }
 $E \rightarrow id_1 \text{ relop } id_2$ { $E.place := newtemp;$
 $emit (\text{ 'if' } id_1.place \text{ relop.op } id_2.place \text{ 'goto' } nextstat + 3);$
 $emit (E.place := '0')$
 $emit (\text{ 'goto' } nextstat + 2)$
 $emit (E.place := '1')$
 }

$E \rightarrow \text{true}$ { $E.place := newtemp; emit (E.place := '1')$ }
 $E \rightarrow \text{false}$ { $E.place := newtemp; emit (E.place := '0')$ }

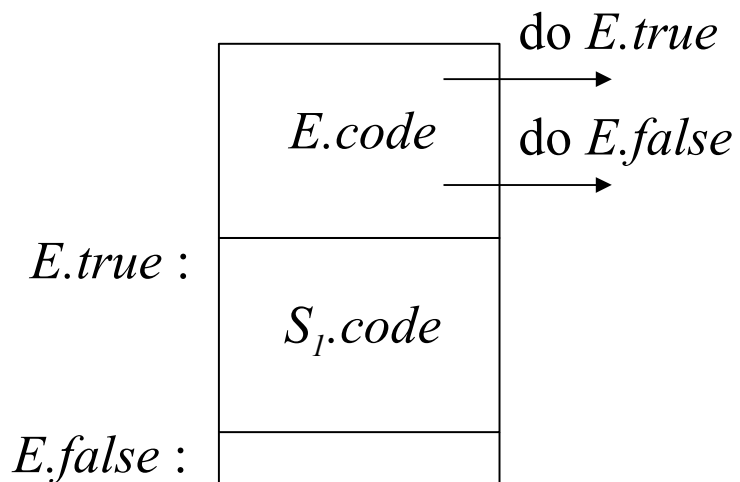
$a < b \text{ or } c < d \text{ and } e < f$

```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
104: if c < d goto 107
105: t2 := 0
106: goto 108
```

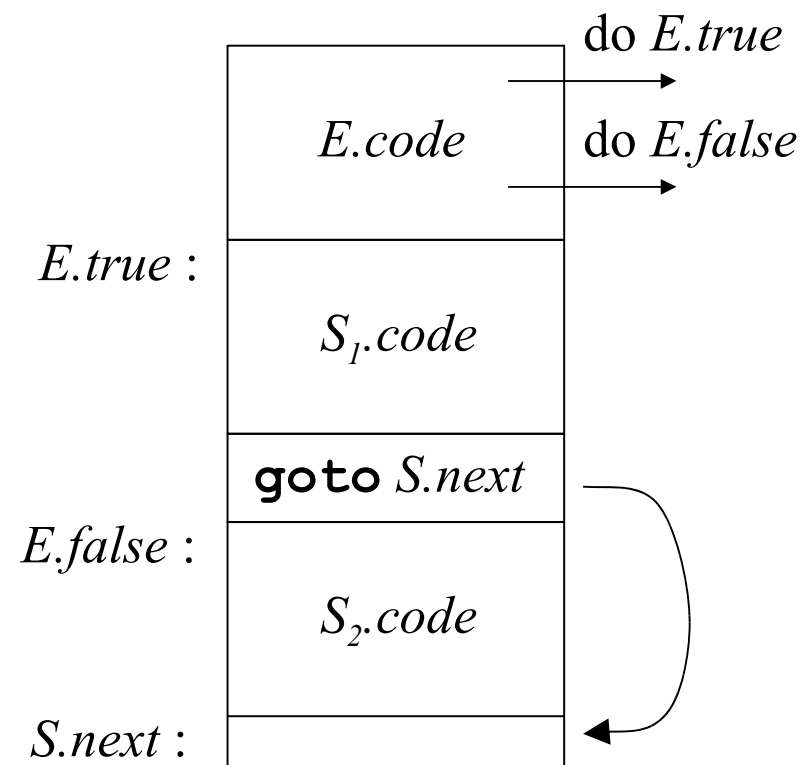
```
107: t2 := 1
108: if e < f goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
113: t5 := t1 or t4
```

Instrukcje sterujace

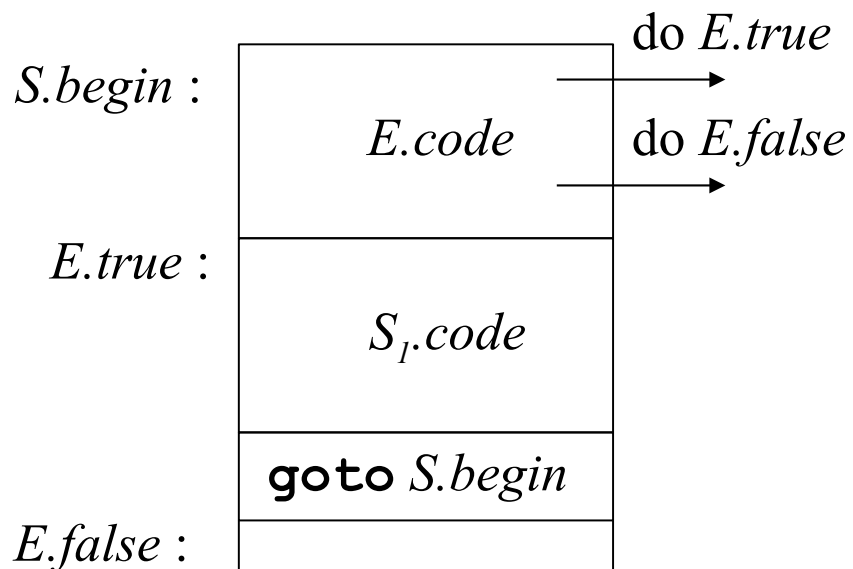
$S \rightarrow \text{if } E \text{ then } S_1$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$



$S \rightarrow \text{while } E \text{ do } S_1$



Instrukcje sterujace - gramatyka

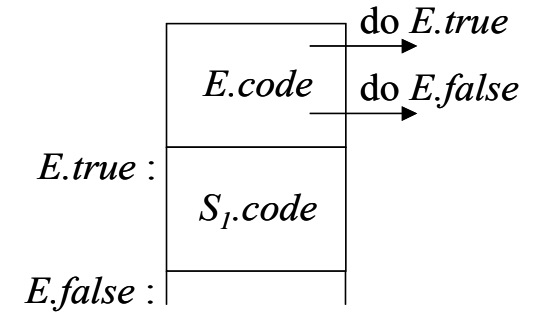
$S \rightarrow \text{if } E \text{ then } S_1$

 $E.true := \text{newlabel};$

 $E.false := S.next;$

 $S_1.next := S.next;$

 $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

 $E.true := \text{newlabel};$

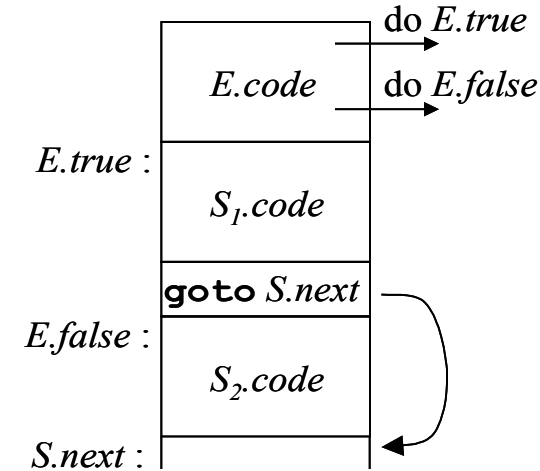
 $E.false := \text{newlabel};$

 $S_1.next := S.next;$

 $S_2.next := S.next;$

 $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code \parallel$

 $\text{gen}(\text{'goto' } S.next) \parallel \text{gen}(E.false ':') \parallel S_2.code$



$S \rightarrow \text{while } E \text{ do } S_1$

 $S.begin := \text{newlabel};$

 $E.true := \text{newlabel};$

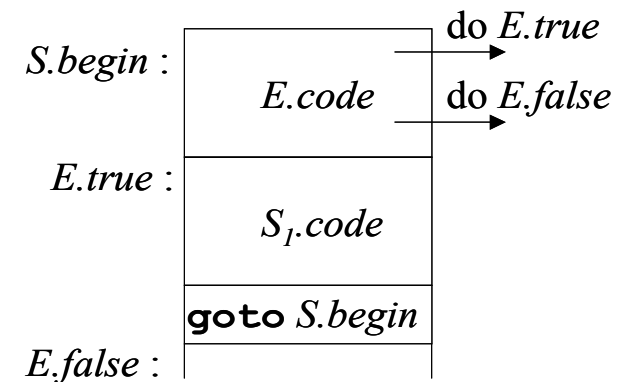
 $E.false := S.next$

 $S_1.next := S.begin;$

 $S.code := \text{gen}(S.begin ':') \parallel E.code \parallel$

 $\text{gen}(E.true ':') \parallel S_1.code \parallel$

 $\text{gen}(\text{'goto' } S.begin)$



Wyrażenia logiczne - translacja "short-circuit"

$E \rightarrow E_1 \text{ or } E_2$

```
E1.true := E.true;  
E1.false := newlabel;  
E2.true := E.true;  
E2.false := E.false;  
E.code := E1.code || gen(E1.false ':') || E2.code
```

$E \rightarrow E_1 \text{ and } E_2$

```
E1.true := newlabel;  
E1.false := E.false;  
E2.true := E.true;  
E2.false := E.false;  
E.code := E1.code || gen(E1.true ':') || E2.code
```

$E \rightarrow \text{not } E_1$

```
E1.true := E.false;  
E1.false := E.true;  
E.code := E1.code;
```

$E \rightarrow (E_1)$

```
E1.true := E.true;  
E1.false := E.false;  
E.code := E1.code;
```

$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$

```
E.code := gen( 'if' id1.place relop.op id2.place 'goto' E.true ) ||  
gen( 'goto' E.false )
```

$E \rightarrow \text{true}$

```
E.code := gen( 'goto' E.true )
```

$E \rightarrow \text{false}$

```
E.code := gen( 'goto' E.false )
```

Kod dla $E = a < b$:

```
if a < b goto E.true  
goto E.false
```

Wyrażenia logiczne - translacja "short-circuit"

a < b or c < d and e < f

```
    if a < b goto Ltrue
    goto L1
L1:  if c < d goto L2
    goto Lfalse
L2:  if e < f goto Ltrue
    goto Lfalse
```

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

```
L1:  if a < b goto L2
    goto Lnext
L2:  if c < d goto L3
    goto L4
L3:  t1 := y + z
    x := t1
    goto L1
L4:  t2 := y - z
    x := t2
    goto L1
Lnext:
```

Wyrażenia logiczne - tryb mieszany

$$(a + b) < c$$
$$(a < b) + (b < a)$$

$E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid \text{id}$

$E \rightarrow E_1 + E_2$ $E.type := arith;$
if $E_1.type = arith$ **and** $E_2.type = arith$ **then begin**
 /* dodawanie arytmetyczne */
 $E.place := newtemp;$
 $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
end
else if $E_1.type = arith$ **and** $E_2.type = bool$ **then begin**
 $E.place := newtemp;$
 $E_2.true := newlabel;$
 $E_2.false := newlabel;$
 $E.code := E_1.code \parallel E_2.code \parallel gen(E_2.true := E.place := E_1.place + 1) \parallel$
 $gen('goto' nextstat + 1) \parallel$
 $gen(E_2.false := E.place := E_1.place)$
else if ...

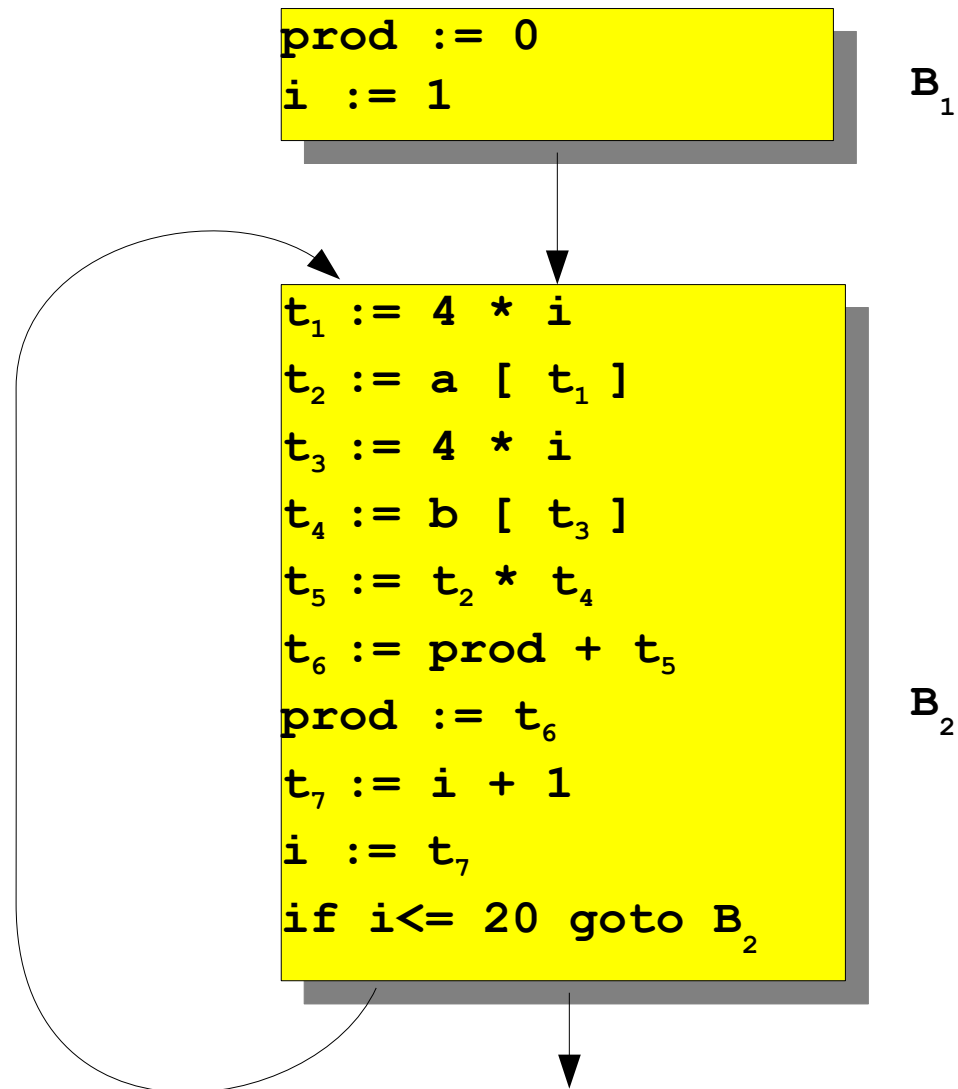
$E_2.true : E.place := E_1.place + 1$
 goto $nextstat + 1$
 $E_2.false : E.place := E_1.place$

Bloki podstawowe

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i := i + 1
  end
  while i <= 20
end
```

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Graf przepływu



Określenie następnych odwołań

$i : x := y \text{ op } z$

- Dołącz do instrukcji i informację z tablicy symboli dotyczącą następnych odwołań i żywotności x , y i z .
- W tablicy symboli ustaw atrybuty x jako “martwa” i “niewykorzystana”
- W tablicy symboli ustaw y i z jako żywe i następne użycia y i z w instrukcji i

Algorytm generacji kodu

Dla każdej instrukcji postaci $x:=y$ op z wykonaj poniższe operacje:

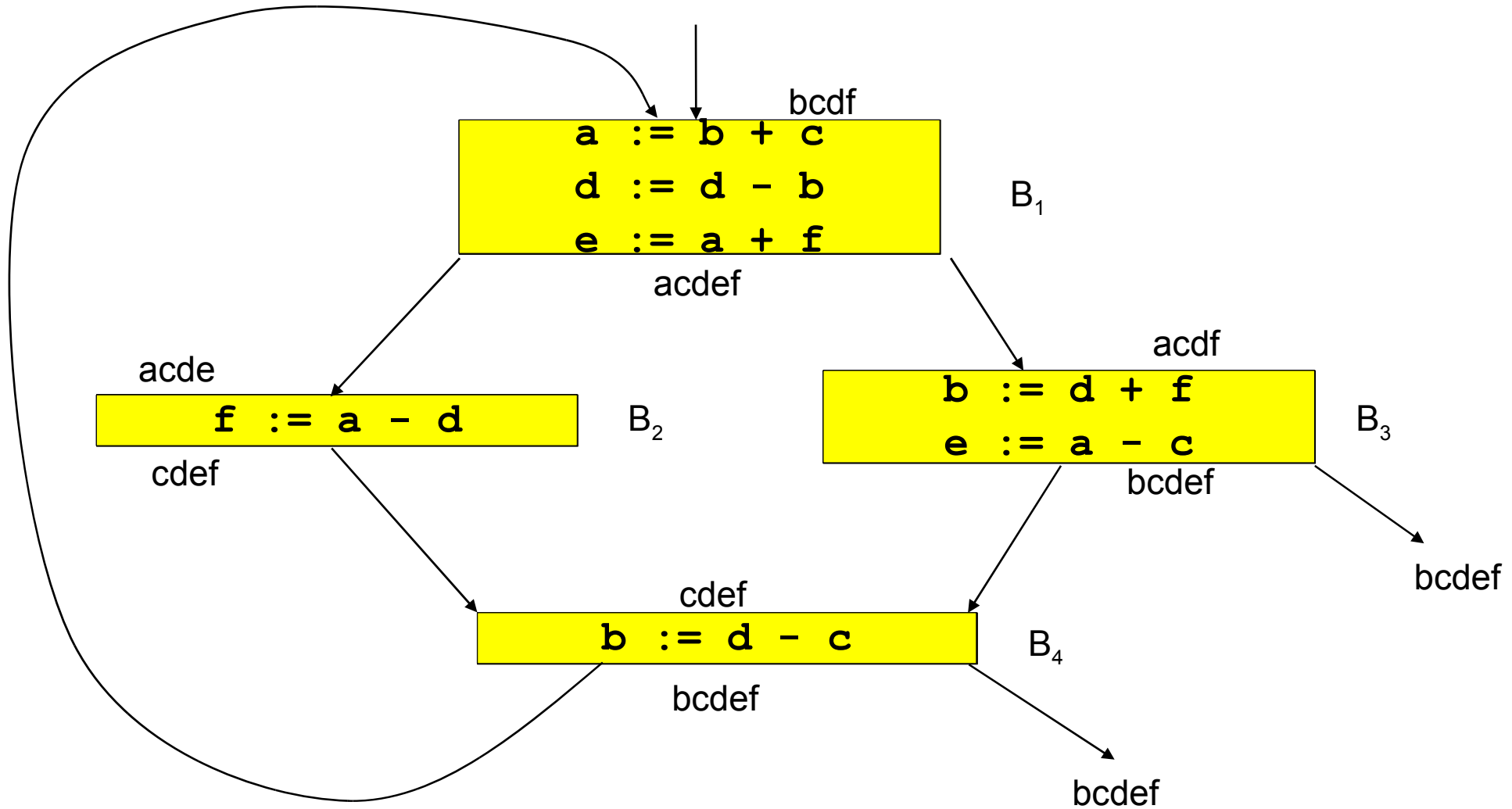
1. Wywołaj funkcję `getreg` w celu określenia lokalizacji L w której będzie przechowany wynik operacji. L zazwyczaj jest rejestrem, ale może być także adresem komórki pamięci.
2. Z deskryptora adresowego y pobierz y' , (jedną z) bieżących lokalizacji y . Preferuj rejestr jeżeli y jest obecnie w pamięci i rejestrze. Jeżeli wartość y nie jest obecnie w L , generuj instrukcję `MOV y' .L`.
3. Generuj instrukcję `OP z' ,L`, gdzie z' jest bieżącą lokalizacją z . Preferuj rejestr jeżeli z jest w rejestrze i w pamięci. Uaktualnij deskryptor adresowy x wskazując, że x jest w L . Jeżeli L jest rejestrem, uaktualnij jego deskryptor wskazując, że zawiera x i usuń x ze wszystkich innych deskryptorów rejestrów
4. Jeżeli bieżące wartości y i z nie mają następnych użyć, są martwe przy wyjściu z bloku podstawowego i są w rejestrach, zmień deskryptory rejestrów wskazując, że te rejestry nie zawierają już odpowiednio y oraz z .

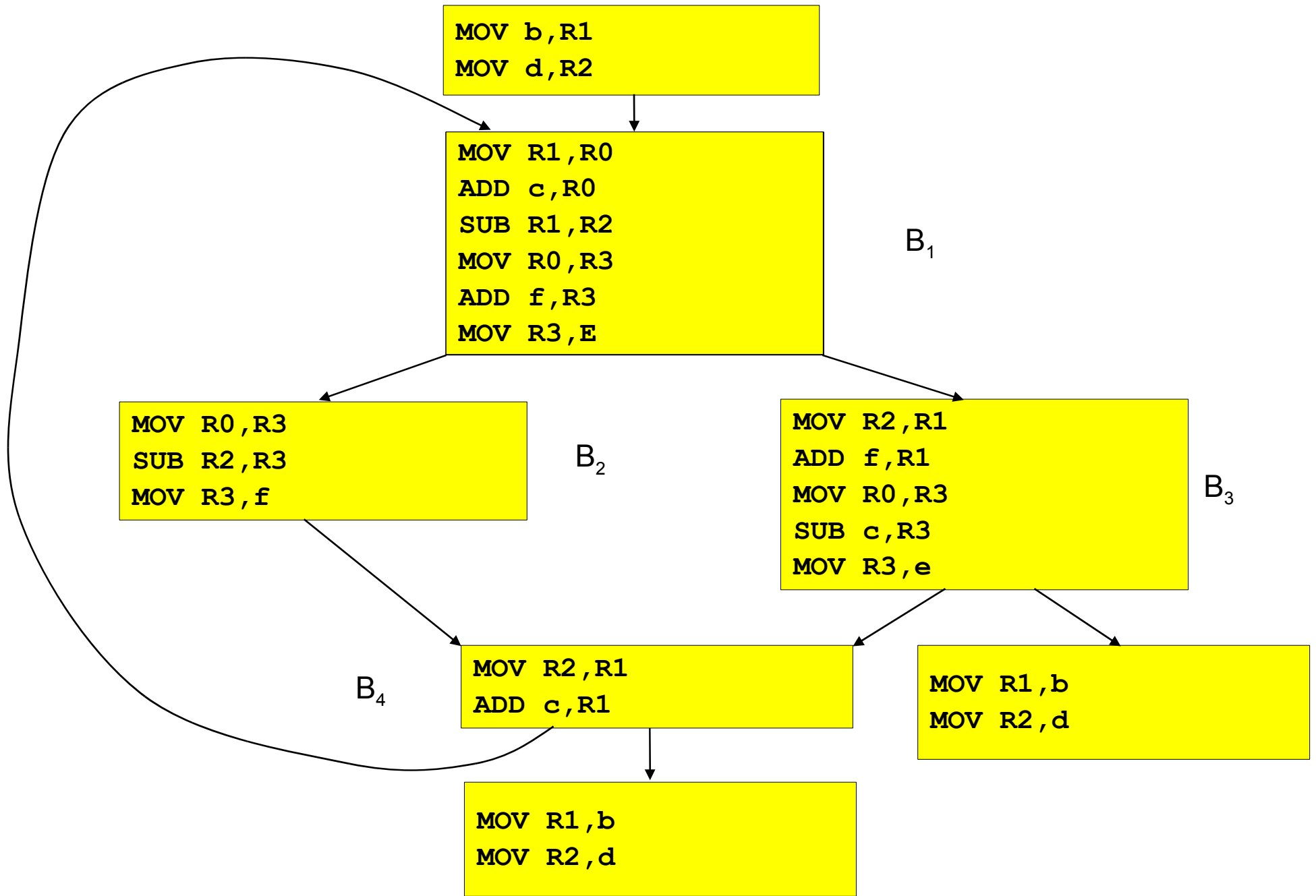
Funkcja `getreg`

Dana jest instrukcja postaci `x:=y op z`

1. Jeżeli `y` znajduje się w rejestrze nie zawierającym innych zmiennych, `y` jest martwe i nie ma następnego użycia po wykonaniu operacji `x:=y op z`, zwróć rejestr w którym jest `y` jako `L`. Uaktualnij deskryptor adresu dla `y` wskazując, że `y` już nie znajduje się w `L`.
2. W przeciwnym przypadku, zwróć pusty rejestr (jeżeli istnieje) jako lokalizację `L`.
3. W przeciwnym wypadku, jeżeli `x` ma następne użycie w bloku, albo `op` jest operatorem, który wymaga rejestru (np. indeksowanie) znajdź zajęty rejestr `R`. Przechowaj wartość `R` w lokalizacji w pamięci (`MOV R,M`) jeżeli nie znajduje się we właściwej komórce pamięci, uaktualnij deskryptor adresu dla `M` i zwróć `R`. Jeżeli `R` przechowuje kilka zmiennych, instrukcja `MOV` musi być wygenerowana dla każdej z nich.
4. Jeżeli `x` nie ma następnego użycia w bloku, albo nie można znaleźć odpowiedniego zajętego rejestru, wybierz komórkę pamięci jako `L`.

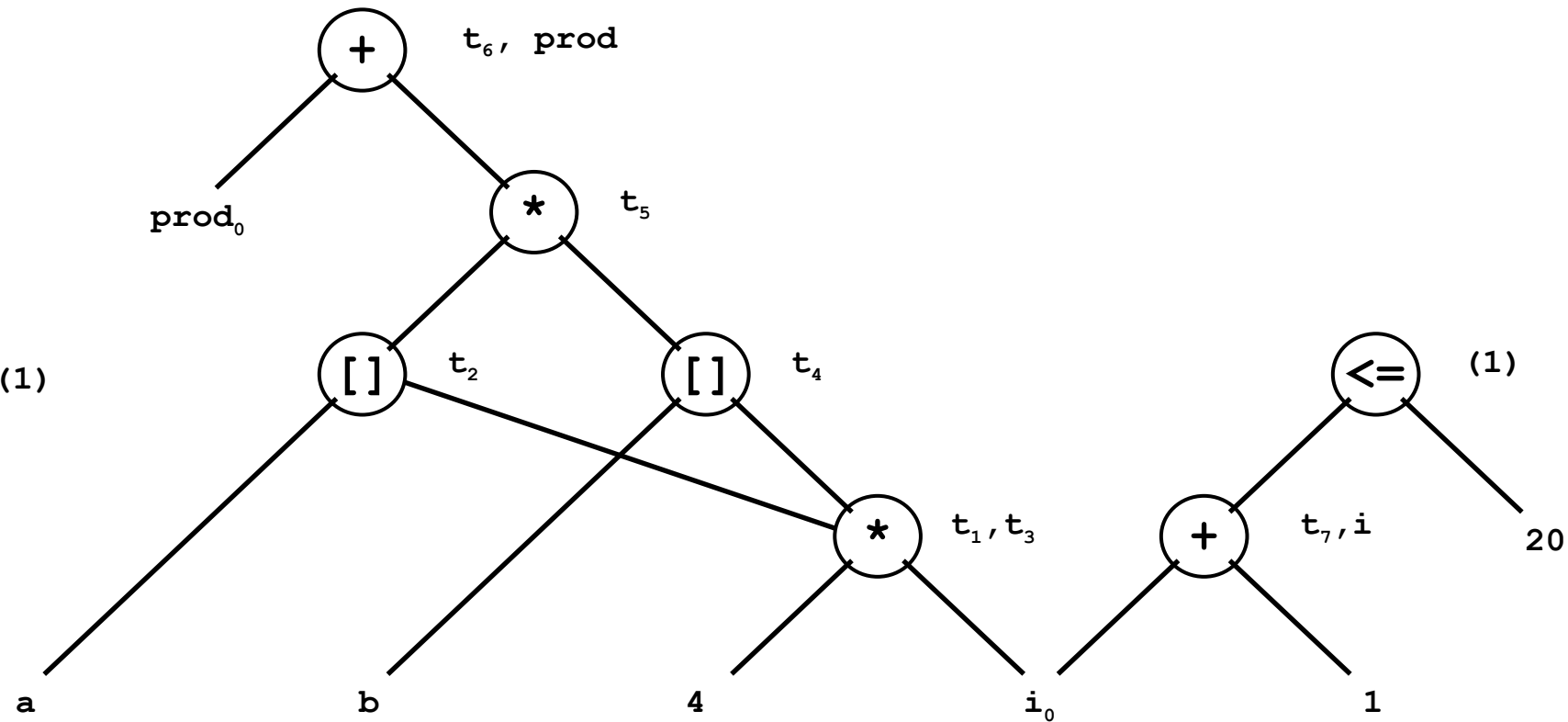
Alokacja rejestrów





Reprezentacja bloków podstawowych w postaci skierowanych grafów acyklicznych

- (1) $t_1 := 4 * i$
- (2) $t_2 := a [t_1]$
- (3) $t_3 := 4 * i$
- (4) $t_4 := b [t_3]$
- (5) $t_5 := t_2 * t_4$
- (6) $t_6 := \text{prod} + t_5$
- (7) $\text{prod} := t_6$
- (8) $t_7 := i + 1$
- (9) $i := t_7$
- (10) $\text{if } i \leq 20 \text{ goto } (1)$



Reprezentacja bloków podstawowych w postaci skierowanych grafów acyklicznych

Dodajmy do tablicy symboli dla identyfikatora x pole $node(x)$, wskazujące węzeł reprezentujący x w bieżącym momencie tworzenia grafu. Początkowo wartość $node(x)$ jest nieokreślona. Rozważmy trzy postacie instrukcji:

- (i) $x := y \ op \ z$ (instrukcja dwuargumentowa)
- (ii) $x := op \ y$ (instrukcja jednoargumentowa)
- (iii) $x := y$ (instrukcja kopiowania)

Dla każdej instrukcji w bloku po kolei wykonaj po kolei następujące czynności:

1. Jeżeli $node(y)$ jest nieokreślone, stwórz węzeł y i niech $node(y)$ wskazuje na ten węzeł. W przypadku (i) wykonaj podobne czynności dla z .
2. W przypadku (i) znajdź węzeł oznaczony op , którego lewym potomkiem jest $node(y)$, a prawym $node(z)$. Jeżeli taki węzeł nie istnieje, utwórz go. Niech n oznacza węzeł właśnie utworzony lub znaleziony. W przypadku (ii) znajdź lub stwórz węzeł oznaczony op , który posiada potomka $node(y)$. W przypadku (iii) n jest węzłem wskazywanym przez $node(y)$, utworzonym jeżeli jeszcze nie istnieje.
3. Usun x z listy identyfikatorów dołączonych do $node(x)$. Dodaj x do listy identyfikatorów dla węzła n i niech $node(x)$ wskazuje na węzeł n .

Tablice, wskaźniki i wywołania procedur

$x := a[i]$	$x := a[i]$
$a[j] := y$	$z := x$
$z := a[i]$	$a[j] := y$

Powyższa "optymalizacja" da błędny rezultat dla $i = j$ i $y \neq a[i]$.

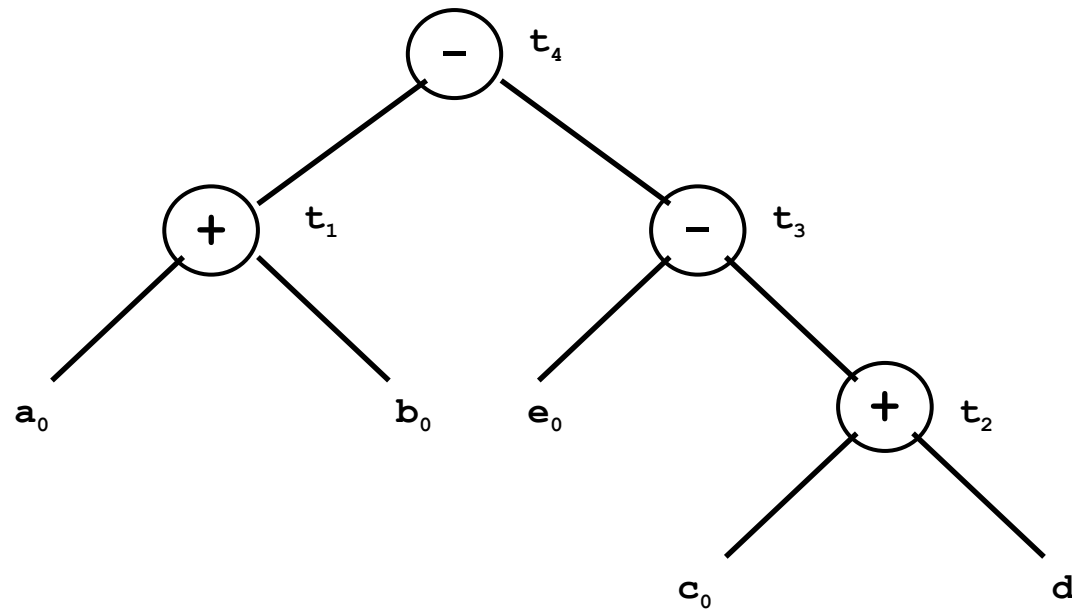
Przy przypisaniu do elementu tablicy a musimy "zabić" węzły oznaczone [], którego lewym argumentem jest a powiększony lub pomniejszony o jakąś wartość. Wywołanie procedury lub przypisanie za pośrednictwem wskaźnika "zabija" wszystkie węzły.

Jeżeli chcemy odtworzyć blok z grafu w innej kolejności, musimy w grafie zaznaczyć kolejność obliczania węzłów, wprowadzając dodatkowe krawędzie $m \rightarrow n$, oznaczające, że m należy obliczyć przed n .

1. Użycie elementu tablicy a lub przypisanie do elementu tablicy a musi być poprzedzone przez wszelkie poprzednie przypisania do elementów a .
2. Przypisanie do elementu tablicy a musi być poprzedzone przez wszystkie uprzednie użycia elementów a .
3. Użycie identyfikatora musi być poprzedzone przez wszystkie wywołania procedur i przypisania za pośrednictwem wskaźnika
4. Wywołanie procedury lub przypisanie za pośrednictwem wskaźnika musi być poprzedzone przez uprzednie obliczenie wszystkich identyfikatorów

Kolejność instrukcji w kodzie pośrednim

$t_1 := a + b$
 $t_2 := c + d$
 $t_3 := e - t_2$
 $t_4 := t_1 - t_3$

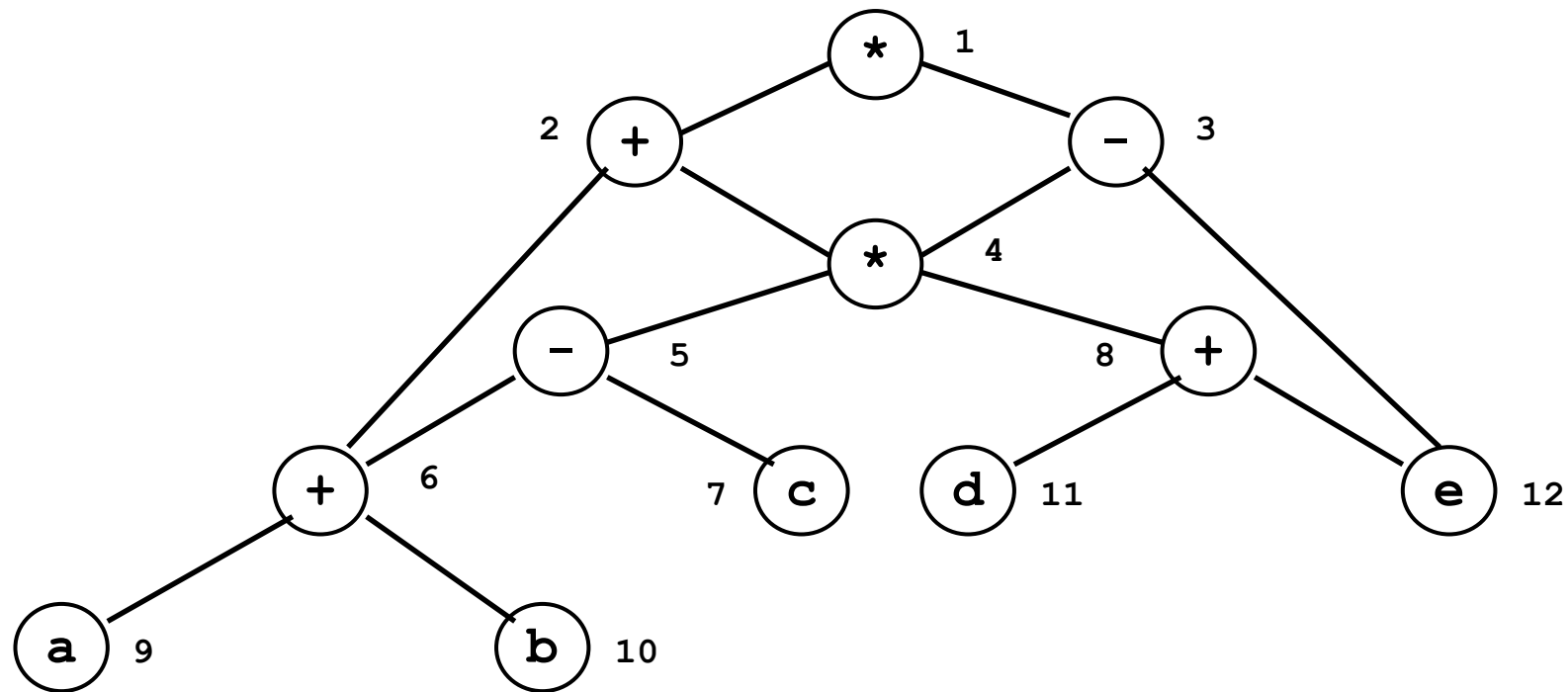


$t_2 := c + d$
 $t_3 := e - t_2$
 $t_1 := a + b$
 $t_4 := t_1 - t_3$

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

Heurystyczne uporządkowanie grafu



```
while pozostały węzły wewnętrzne nie dodane do listy do begin  
  wybierz węzeł  $n$  spoza listy, którego wszystkie węzły rodzicielskie są na liście;  
  dołącz  $n$  do listy;  
  while skrajny lewy potomek  $m$  węzła  $n$  nie ma rodziców poza listą i nie jest liściem do  
    begin  
      dołącz  $m$  do listy;  
       $n := m$   
    end  
  end  
end  
end
```

Peephole optimization

Eliminacja instrukcji nadmiarowych

```
MOV R0, a
MOV a, R0
```

Optymalizacja przepływu sterowania

```
goto L1
...
L1:goto L2
    ↓
    goto L2
...
L1:goto L2
```

```
if a < b goto L1
```

```
...
```

```
L1:goto L2
```

↓

```
if a < b goto L2
```

```
...
```

```
L1:goto L2
```

```
goto L1
```

```
...
```

```
L1:if a<b goto L2
```

```
L3:
```

↓

```
if a<b goto L2
```

```
goto L3
```

```
...
```

```
L3:
```

Peephole optimization

Uproszczenia algebraiczne

`x := x + 0`

`x := x * 1`

Redukcja mocy

`x := x * x`

Użycie instrukcji specyficznych dla procesora

Autoinkrementacja

Autodekrementacja

Złożone tryby adresowania

Eliminacja nieosiągalnego kodu

```
#define debug 0
```

```
...
```

```
if ( debug ) {
```

```
    ...
```

```
}
```

```
    if debug = 1 goto L1
```

```
    goto L2
```

```
L1:    ...
```

```
L2:
```

```
    if debug ≠ 1 goto L2
```

```
    ...
```

```
L2:
```

```
    if 0 ≠ 1 goto L2
```

```
    ...
```

```
L2:
```

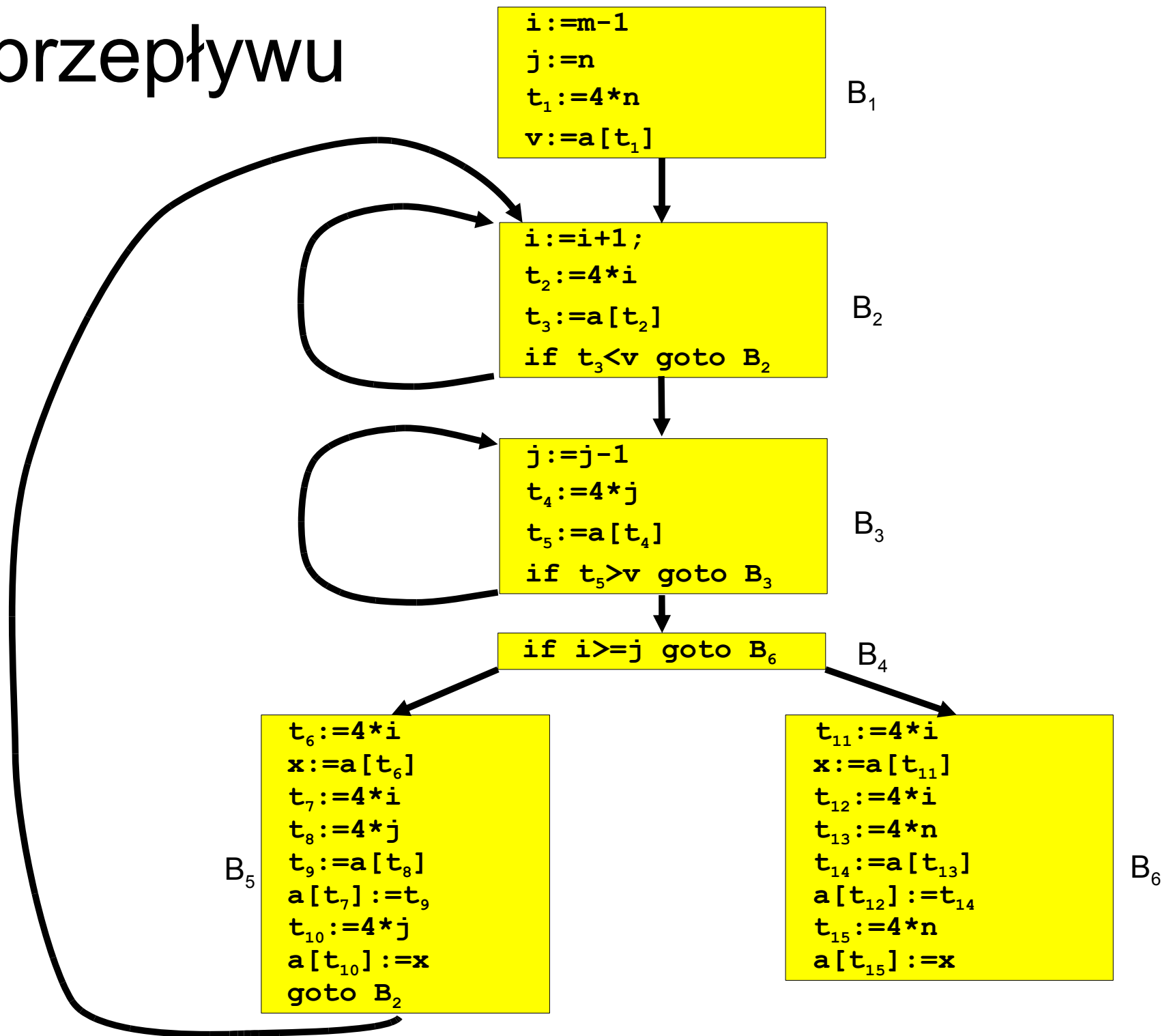
Fragment programu w C

```
void quicksort(m,n)
int m,n
{
    int i,j;
    int v,x;
    if(n<=m) return;
    /*-----*/
    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;
    /*-----*/
    quicksort(m,j);
    quicksort(i+1,n);
}
```

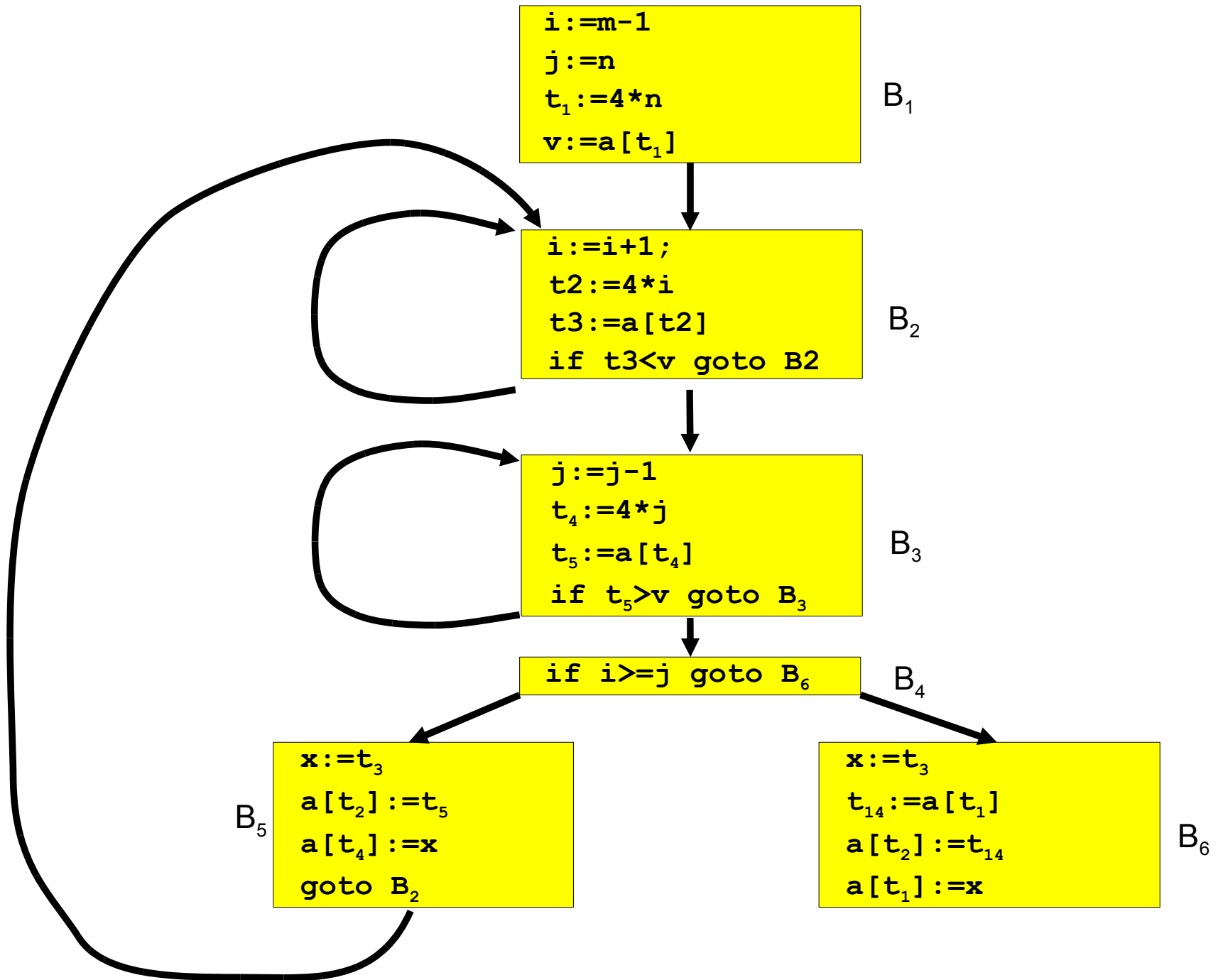
Wygenerowany kod pośredni

```
01 i := m - 1
02 j := n
03 t1 := 4 * n
04 v := a[t1]
05 i := i + 1;
06 t2 := 4 * i
07 t3 := a[t2]
08 if t3 < v goto 5
09 j := j - 1
10 t4 := 4 * j
11 t5 := a[t4]
12 if t5 > v goto 9
13 if i >= j goto 23
14 t6 := 4 * i
15 x := a[t6]
16 t7 := 4 * i
17 t8 := 4 * j
18 t9 := a[t8]
19 a[t7] := t9
20 t10 := 4 * j
21 a[t10] := x
22 goto 5
23 t11 := 4 * i
24 x := a[t11]
25 t12 := 4 * i
26 t13 := 4 * n
27 t14 := a[t13]
28 a[t12] := t14
29 t15 := 4 * n
30 a[t15] := x
```

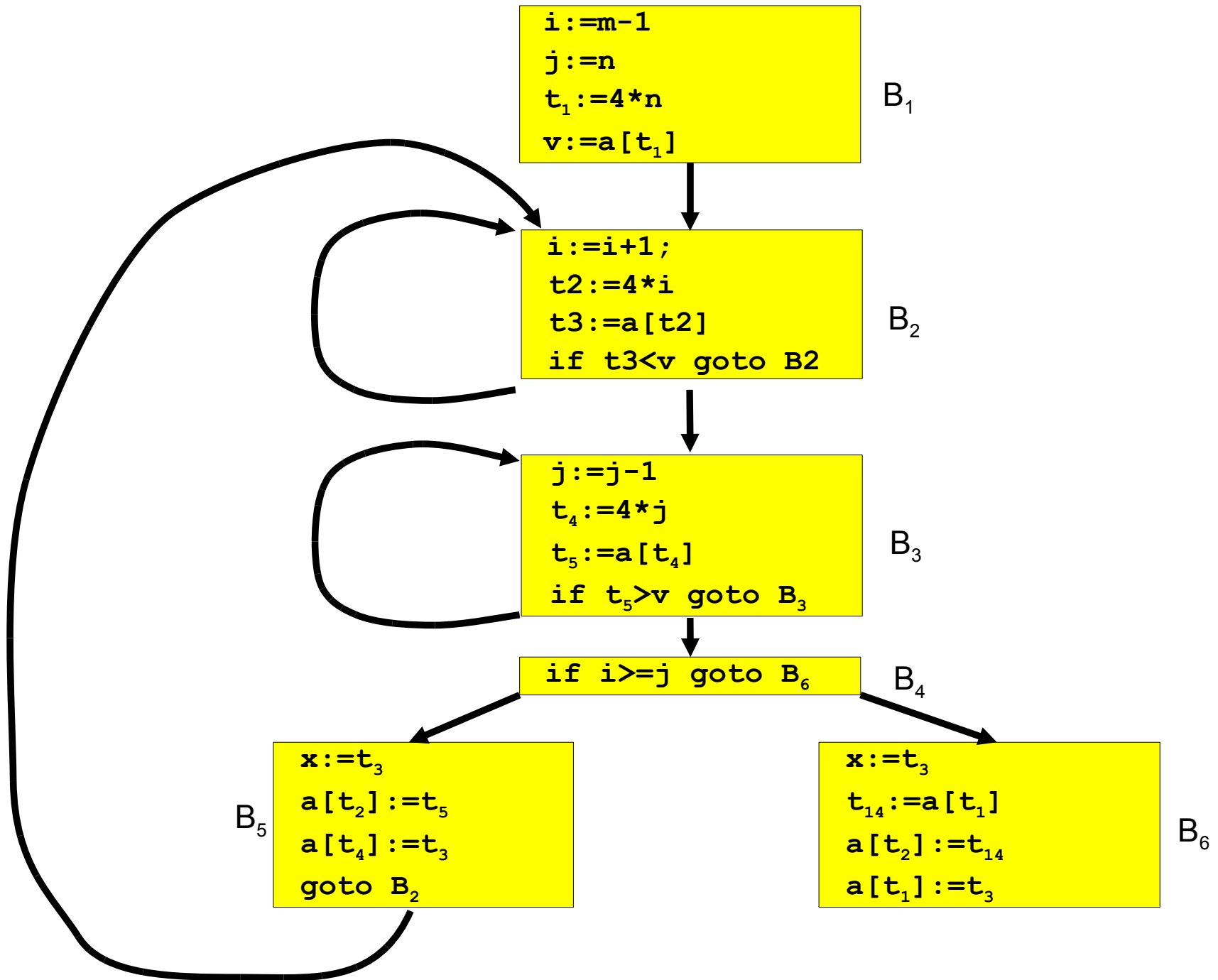
Graf przepływu



Eliminacja wspólnych podwyrażeń



Propagacja kopii



Eliminacja martwego kodu i zmiennych indukcyjnych

