Μεταγλωττιστές
Βελτιστοποίηση

Νίκος Παπασπύρου
nickie@softlab.ntua.gr

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχ. Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού
Πολυτεχνειούπολη, 15780 Ζωγράφου

Βελτιστοποίηση (i)

- Μετασχηματισμός που βελτιώνει τον κώδικα
  - σε ταχύτητα εκτέλεσης
  - σε απαιτήσεις μνήμης
  - και στα δύο
- Παραπλανητικός όρος!
- Θεωρητικά, η λύση του προβλήματος είναι αδύνατη

Βελτιστοποίηση (ii)

- Κριτήρια εφαρμογής μετασχηματισμών βελτιστοποίησης
  - διατηρείται η σημασία των προγραμμάτων
  - κατά μέσο όρο, ο κώδικας βελτιώνεται
  - αξίζει τον κόπο!
  - ασύμφοροι μετασχηματισμοί για τα στάδια ανάπτυξης του προγράμματος

Βελτιστοποίηση (iii)

- Βελτίωση στο επίπεδο του αρχικού κώδικα
  - $2.02n^2$ μsec για insertion sort
  - $12n \log(2n)$ μsec για quicksort
  - για n=100,000 η διαφορά είναι 3 τάξεις μεγέθους!
  - δυστυχώς, ο μεταγλωττιστής γενικά δεν μπορεί να αλλάζει τον αλγόριθμο
- Ο μεταγλωττιστής περιορίζεται σε βελτιώσεις στο επίπεδο του ενδιάμεσου και του τελικού κώδικα

Βελτιστοποίηση (iv)

- Γιατί βελτιστοποίηση στο επίπεδο του ενδιάμεσου κώδικα;
- Βελτιστοποιητής ενδιάμεσου κώδικα
  - ανάλυση ροής ελέγχου
  - ανάλυση ροής δεδομένων
  - βελτιστοποιητικοί μετασχηματισμοί

Παράδειγμα: quicksort

```
procedure quicksort (var a : array of integer; m, n : integer);
   var i, j, temp : integer;
begin
  if n <= m then return;
  i := m; j := n;
  while i <= j do begin
    while a[i] < a[(m+n) div 2] do i := i+1;
    while a[j] > a[(m+n) div 2] do j := j-1;
    if i <= j then begin temp := a[i];
                         a[i] := a[j];
                         a[j] := temp;
                         i := i+1; j := j-1
                   end
  end;
  quicksort(a, m, j);
  quicksort(a, i, n)
end;
```

Ενδιάμεσος κώδικας

```
i := m; j := n;

while i <= j do
begin
  while a[i] < a[(m+n) div 2] do
    i := i+1;
  while a[j] > a[(m+n) div 2] do
    j := j-1;
  if i <= j then
  begin
    temp := a[i];
    a[i] := a[j];
    a[j] := temp;
    i := i+1;
    j := j-1
  end
end;
```

```
 5: :=, m, -, i
 6: :=, n, -, j
 7: <=, i, j, 9
 8: jump, -, -, 41
 9: array, a, i, $1
10: +, m, n, $2
11: /, $2, 2, $3
12: array, a, $3, $4
13: <, [$1], [$4], 15
14: jump, -, -, 18
15: +, i, 1, $5
16: :=, $5, -, i
17: jump, -, -, 9
18: array, a, j, $6
19: +, m, n, $7
20: /, $7, 2, $8
21: array, a, $8, $9
22: >, [$6], [$9], 24
23: jump, -, -, 27
24: -, j, 1, $10
25: :=, $10, -, j
26: jump, -, -, 18
```

```
27: <=, i, j, 29
28: jump, -, -, 7
29: array, a, i, $11
30: :=, [$11], -, temp
31: array, a, i, $12
32: array, a, j, $13
33: :=, [$13], -, [$12]
34: array, a, j, $14
35: :=, temp, -, [$14]
36: +, i, 1, $15
37: :=, $15, -, i
38: -, j, 1, $16
39: :=, $16, -, j
40: jump, -, -, 7
```

Ανάλυση ροής ελέγχου (i)

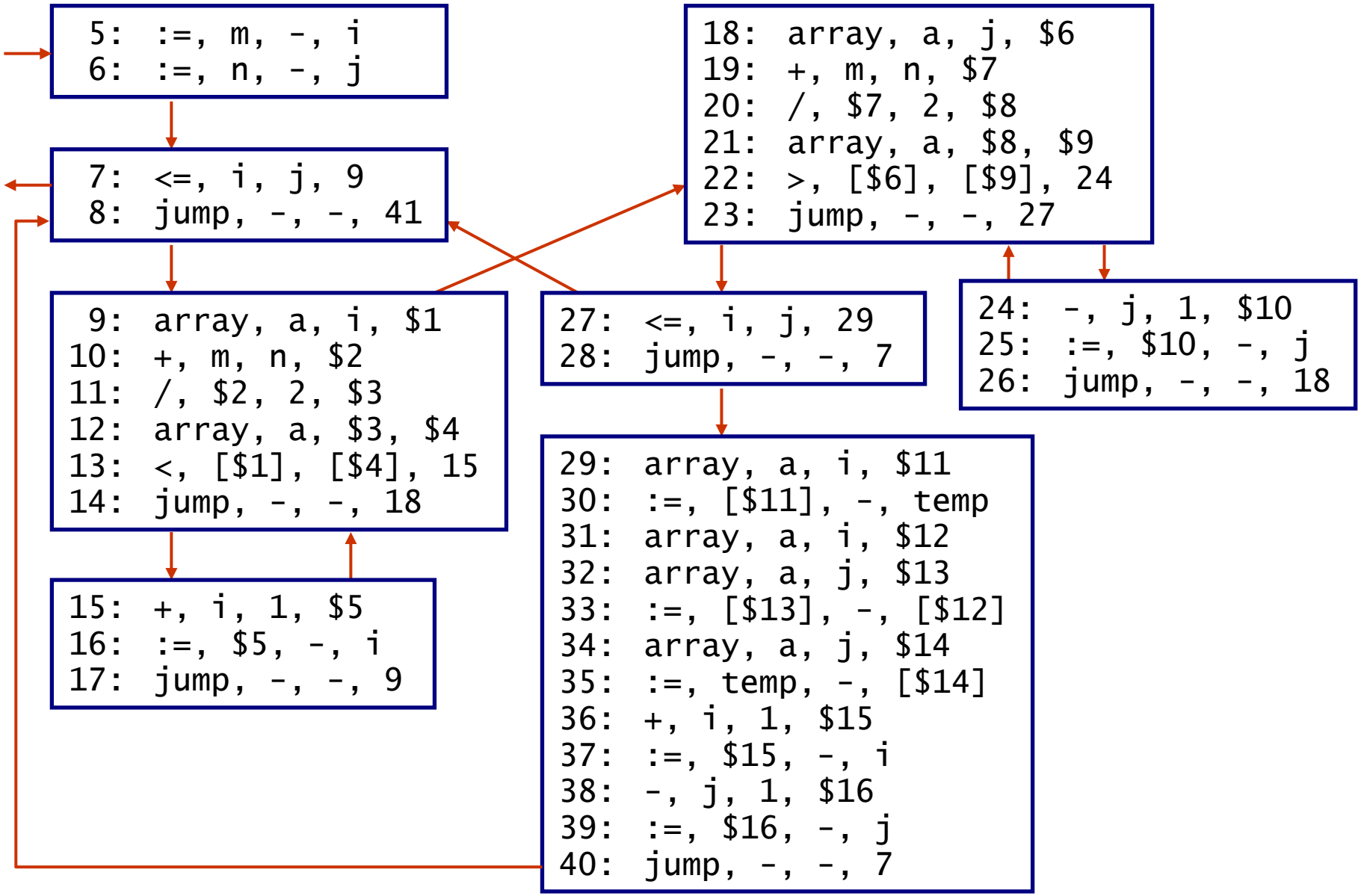
- **Βασική ενότητα**: ακολουθία διαδοχικών τετράδων με τις εξής ιδιότητες
  - αν υπάρχουν τετράδες άλματος (υπό συνθήκη ή χωρίς συνθήκη), αυτές βρίσκονται στο τέλος
  - δεν υπάρχει καμία τετράδα (σε ολόκληρο τον ενδιάμεσο κώδικα) που να κάνει άλμα στο εσωτερικό της βασικής ενότητας
- **Γράφος ροής δεδομένων**
  - οι κόμβοι είναι οι βασικές ενότητες
  - οι ακμές δείχνουν τα άλματα (και τη διαδοχή) μεταξύ βασικών ενοτήτων

Ανάλυση ροής ελέγχου (ii)

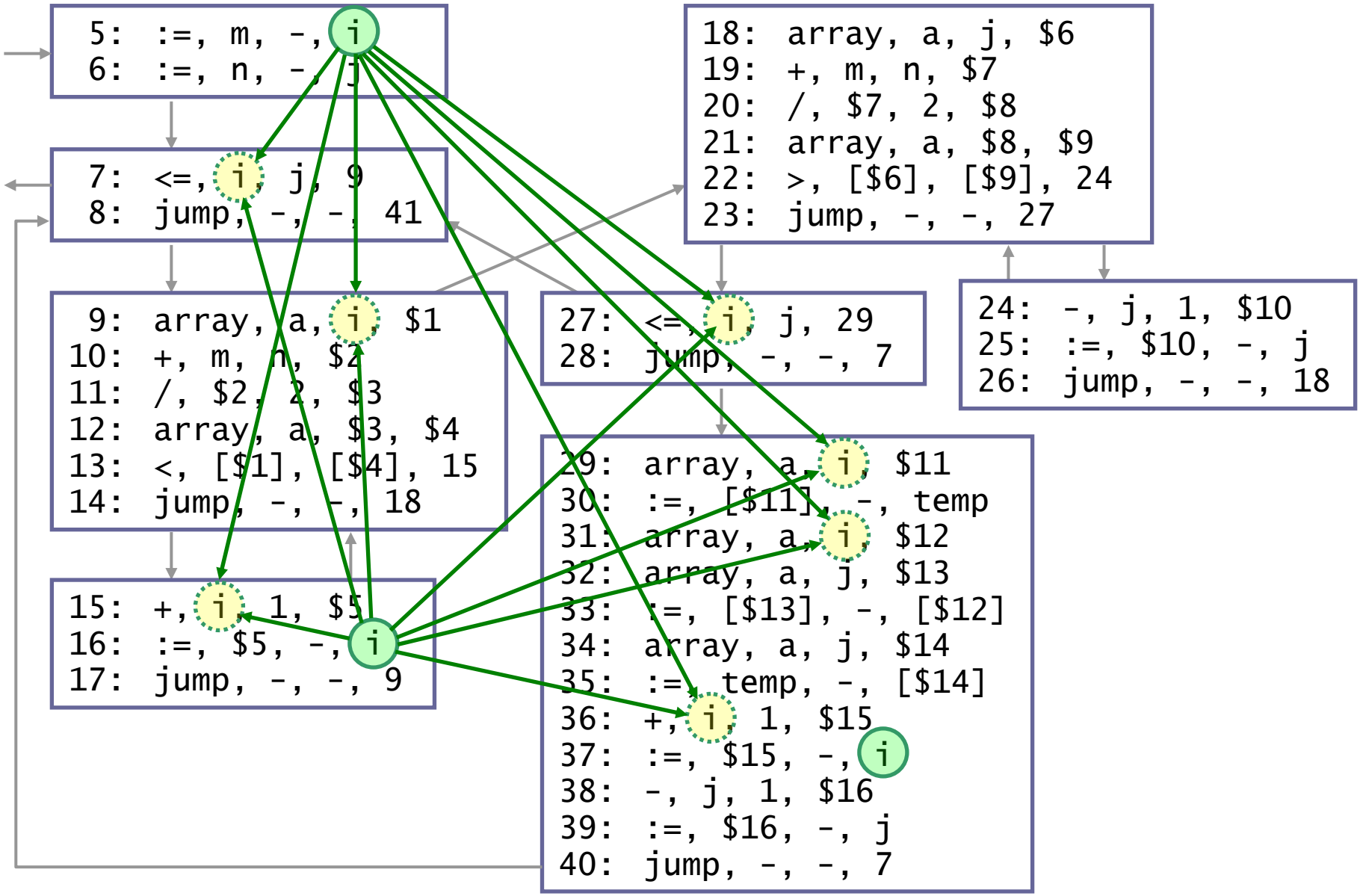
```
 5: :=, m, -, i
 6: :=, n, -, j
 7: <=, i, j, 9
 8: jump, -, -, 41
 9: array, a, i, $1
10: +, m, n, $2
11: /, $2, 2, $3
12: array, a, $3, $4
13: <, [$1], [$4], 15
14: jump, -, -, 18
15: +, i, 1, $5
16: :=, $5, -, i
17: jump, -, -, 9
18: array, a, j, $6
19: +, m, n, $7
20: /, $7, 2, $8
21: array, a, $8, $9
22: >, [$6], [$9], 24
23: jump, -, -, 27
```

```
24: -, j, 1, $10
25: :=, $10, -, j
26: jump, -, -, 18
27: <=, i, j, 29
28: jump, -, -, 7
29: array, a, i, $11
30: :=, [$11], -, temp
31: array, a, i, $12
32: array, a, j, $13
33: :=, [$13], -, [$12]
34: array, a, j, $14
35: :=, temp, -, [$14]
36: +, i, 1, $15
37: :=, $15, -, i
38: -, j, 1, $16
39: :=, $16, -, j
40: jump, -, -, 7
```

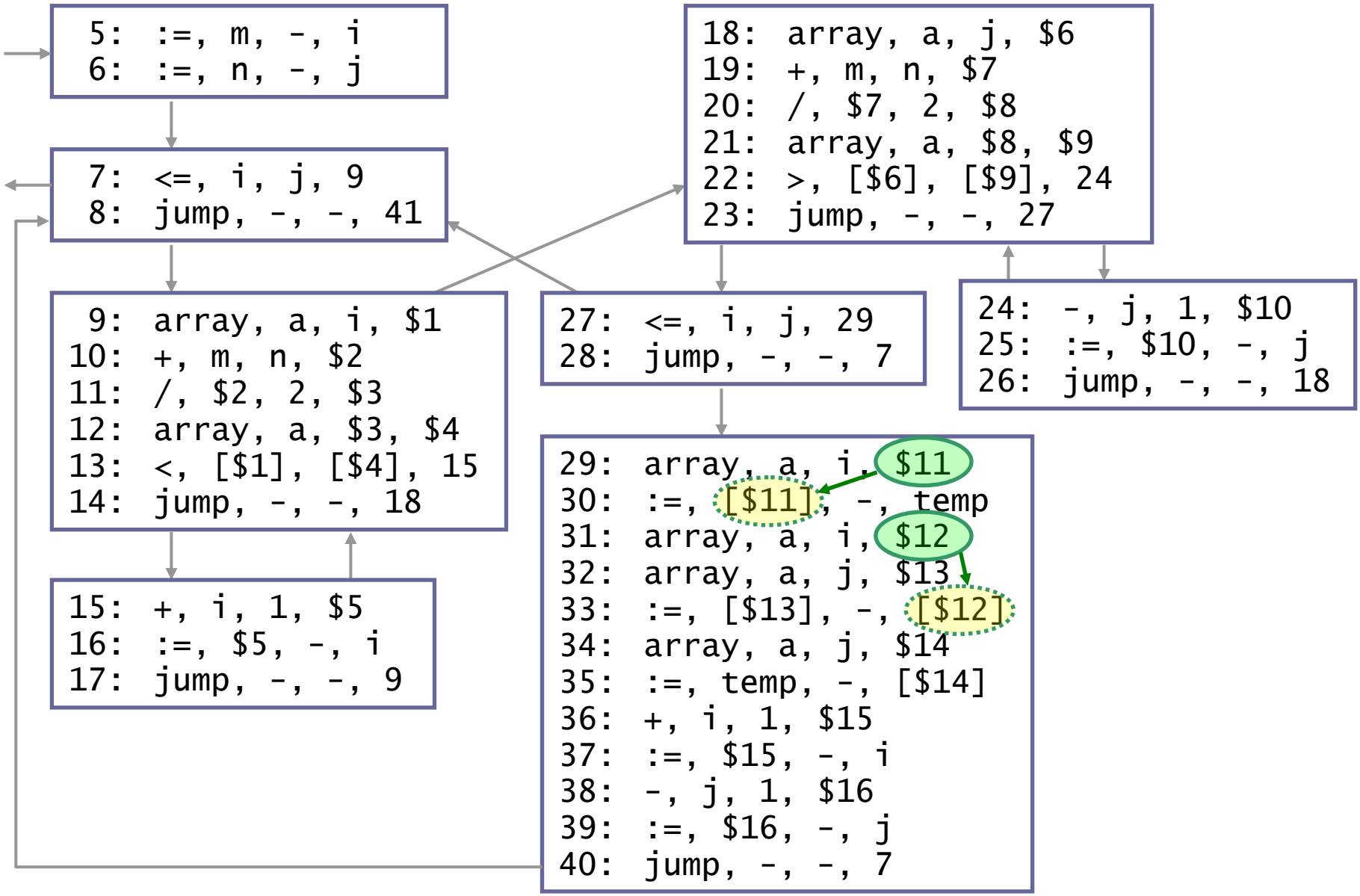
Ανάλυση ροής ελέγχου (iii)



Ανάλυση ροής δεδομένων (i)



Ανάλυση ροής δεδομένων (ii)



Βελτιστοποιητικοί μετασχηματισμοί (i)

- Μετασχηματισμοί υψηλού επιπέδου
  - αποτίμηση σταθερών εκφράσεων
  - αλγεβρικοί μετασχηματισμοί
  - απαλοιφή κοινών υποεκφράσεων
  - διάδοση αντιγράφων
  - ενοποίηση κώδικα
- Μετασχηματισμοί βρόχων
  - μετακίνηση κώδικα
  - απαλοιφή επαγωγικών μεταβλητών
  - αναδιοργάνωση βρόχων
  - απαλοιφή ελέγχου ορίων πίνακα

Βελτιστοποιητικοί μετασχηματισμοί (ii)

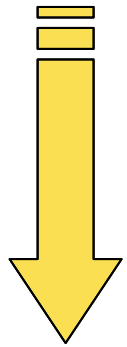
- Μετασχηματισμοί χαμηλού επιπέδου
  - απαλοιφή άχρηστου κώδικα
  - ευθυγράμμιση
  - απλοποίηση συνθηκών και αλμάτων
- Μετασχηματισμοί υποπρογραμμάτων
  - ενσωμάτωση υποπρογράμματος
  - κλήσεις ουράς και συνένωση
  - υποπρογράμματα φύλλα

Αποτίμηση σταθερών εκφράσεων

- **Προϋπόθεση**: το αποτέλεσμα που υπολογίζεται κατά τη μεταγλώττιση να ταυτίζεται με αυτό που θα υπολογιζόταν κατά την εκτέλεση

```
const int ROWS = 100;
const int COLUMNS = 50;
int * p = (int *) malloc(ROWS * COLUMNS * sizeof(int));
```

```
  ROWS * COLUMNS * sizeof(int)
= 100 * 50 * 4
= 20000
```



```
const int ROWS = 100;
const int COLUMNS = 50;
int * p = (int *) malloc(20000);
```

Αλγεβρικοί μετασχηματισμοί (i)

- **Προϋπόθεση**: να μην προκαλείται μεταβολή στο αποτέλεσμα, ακόμα και αν εμφανιστούν εξαιρέσεις κατά την εκτέλεση των πράξεων

- Στις πράξεις ακεραίων, π.χ.

  `x + 0 = x` `x * 0 = 0` `x * 1 = x`

  `0 + x = x` `0 * x = 0` `1 * x = x`

- Εσφαλμένοι μετασχηματισμοί

  `-(-x) ≠ x` `x + y - x ≠ y`

Αλγεβρικοί μετασχηματισμοί (ii)

- Στις πράξεις κινητής υποδιαστολής

  `x * 0.0 ≠ 0.0`

- Στις λογικές πράξεις

  `b or true = true` `b and true = b`

  `b and false = false` `b or false = b`

  `not (not b) = b`

- Άλλοι μετασχηματισμοί

  `(@x)^ = x` `@(p^) ≠ p`

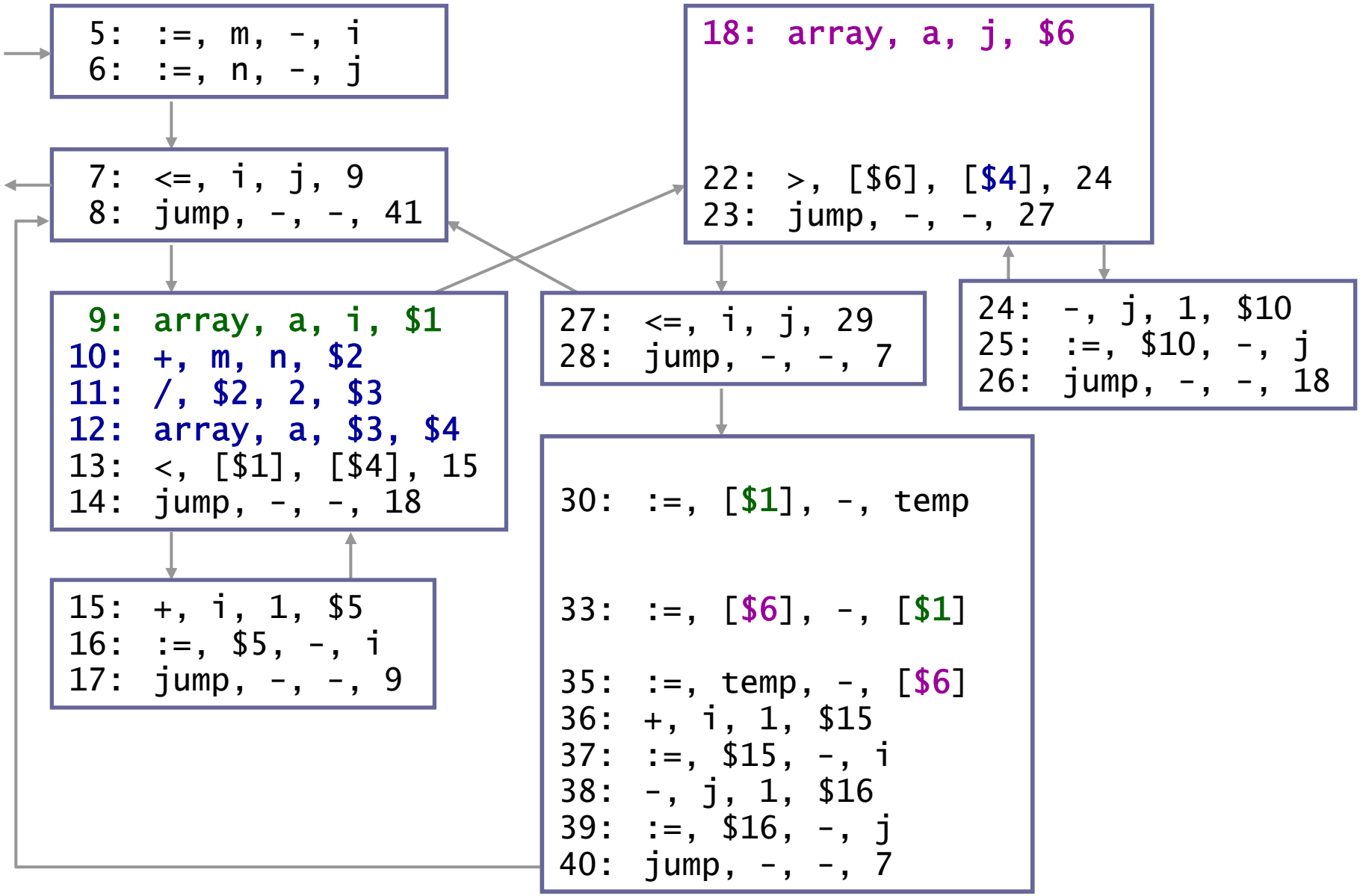
Απαλοιφή κοινών υποεκφράσεων (i)

- **Προϋπόθεση**: η τιμή της κοινής υποέκφρασης να μην έχει μεταβληθεί από τον προηγούμενο υπολογισμό της
- **Προϋπόθεση**: η αποτίμηση της κοινής υποέκφρασης να μην προκαλεί παρενέργειες

```
i := m; j := n;

while i <= j do begin
  while a[i] < a[(m+n) div 2] do i := i+1;
  while a[j] > a[(m+n) div 2] do j := j-1;
  if i <= j then begin temp := a[i];
                       a[i] := a[j];
                       a[j] := temp;
                       i := i+1; j := j-1
                 end
end;
```

Απαλοιφή κοινών υποεκφράσεων (ii)



Διάδοση αντιγράφων (i)

- **Τετράδα αντιγραφής**: `:=, y, -, x`
- **Ιδέα**: να χρησιμοποιείται το πρωτότυπο `y` αντί του αντιγράφου `x`
- **Προϋπόθεση**: να μην έχουν μεσολαβήσει άλλες αναθέσεις στις μεταβλητές `x` ή `y`

```
:=, y, -, x
+, x, 3, $1
:=, $1, -, z
par, z, V, -
call, -, -, P
```

```
:=, y, -, x
+, y, 3, $1
:=, $1, -, z
par, $1, V, -
call, -, -, P
```

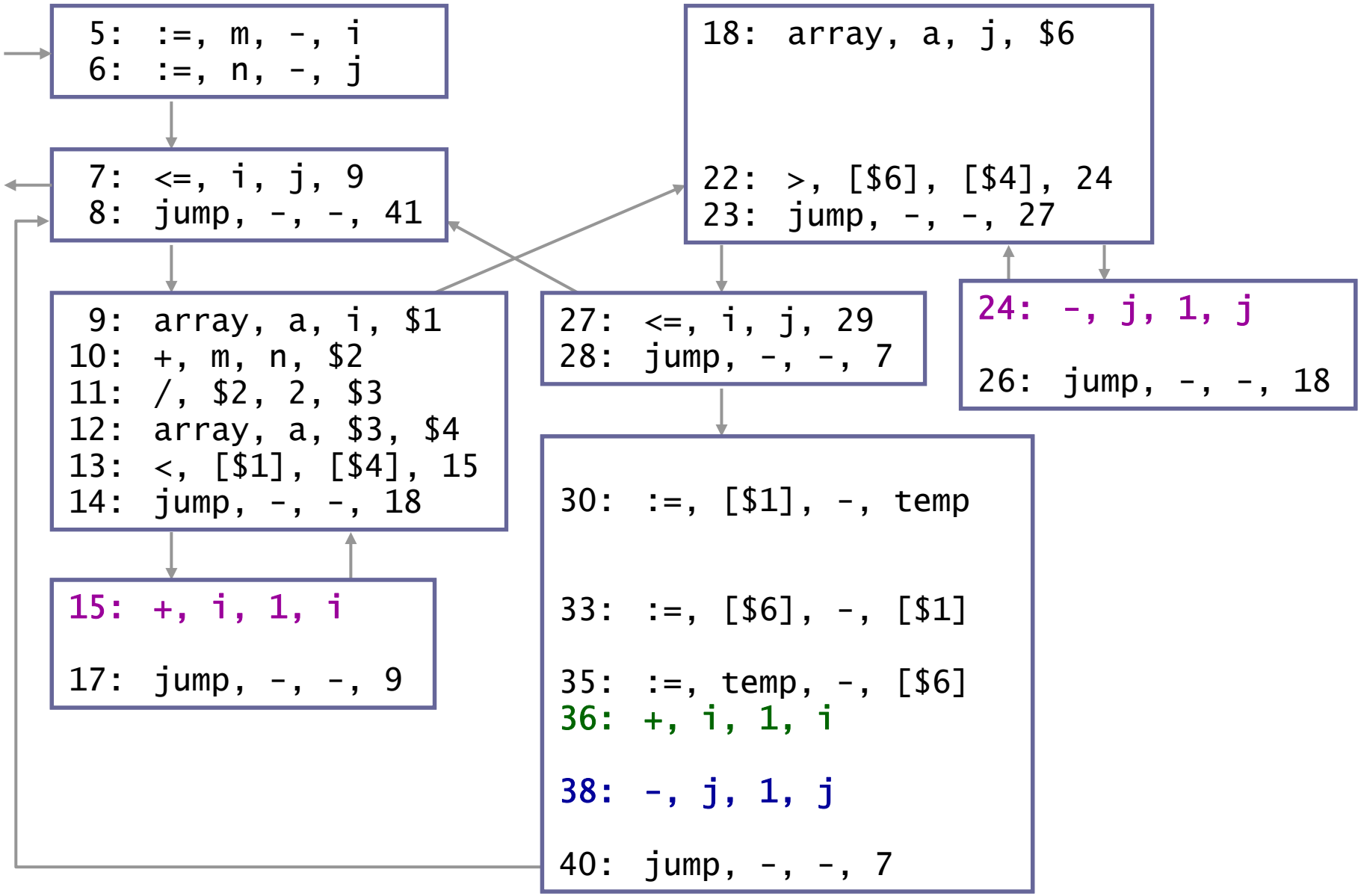
- **Γιατί;** Επιτρέπει άλλες βελτιστοποιήσεις σε μεταγενέστερο στάδιο

Διάδοση αντιγράφων (ii)

- Η αντίστροφη μορφή του μετασχηματισμού διάδοσης αντιγράφων προσφέρεται για το απλό σχήμα παραγωγής ενδιάμεσου κώδικα της PCL

- **Προϋπόθεση**: να μη χρησιμοποιείται αλλού η μεταβλητή $1 (αυτό μπορεί να αποδειχθεί)

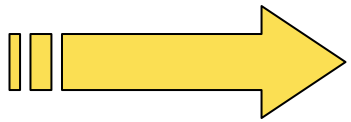
Διάδοση αντιγράφων (iii)



Ενοποίηση κώδικα

- **Ιδέα**: μεταφορά κώδικα που εκτελείται ανεξάρτητα της εκάστοτε ροής εκτέλεσης όσο το δυνατόν νωρίτερα (αργότερα)

```
if i > 0 then
begin
  x := 2*i;
  s := s + x*i;
  i := i-1
end
else
begin
  s := 0;
  x := 2*i;
  i := i-1
end
```

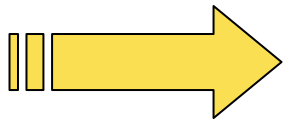


```
x := 2*i;
if i > 0 then
  s := s + x*i
else
  s := 0;
i := i-1
```

Μετακίνηση κώδικα (i)

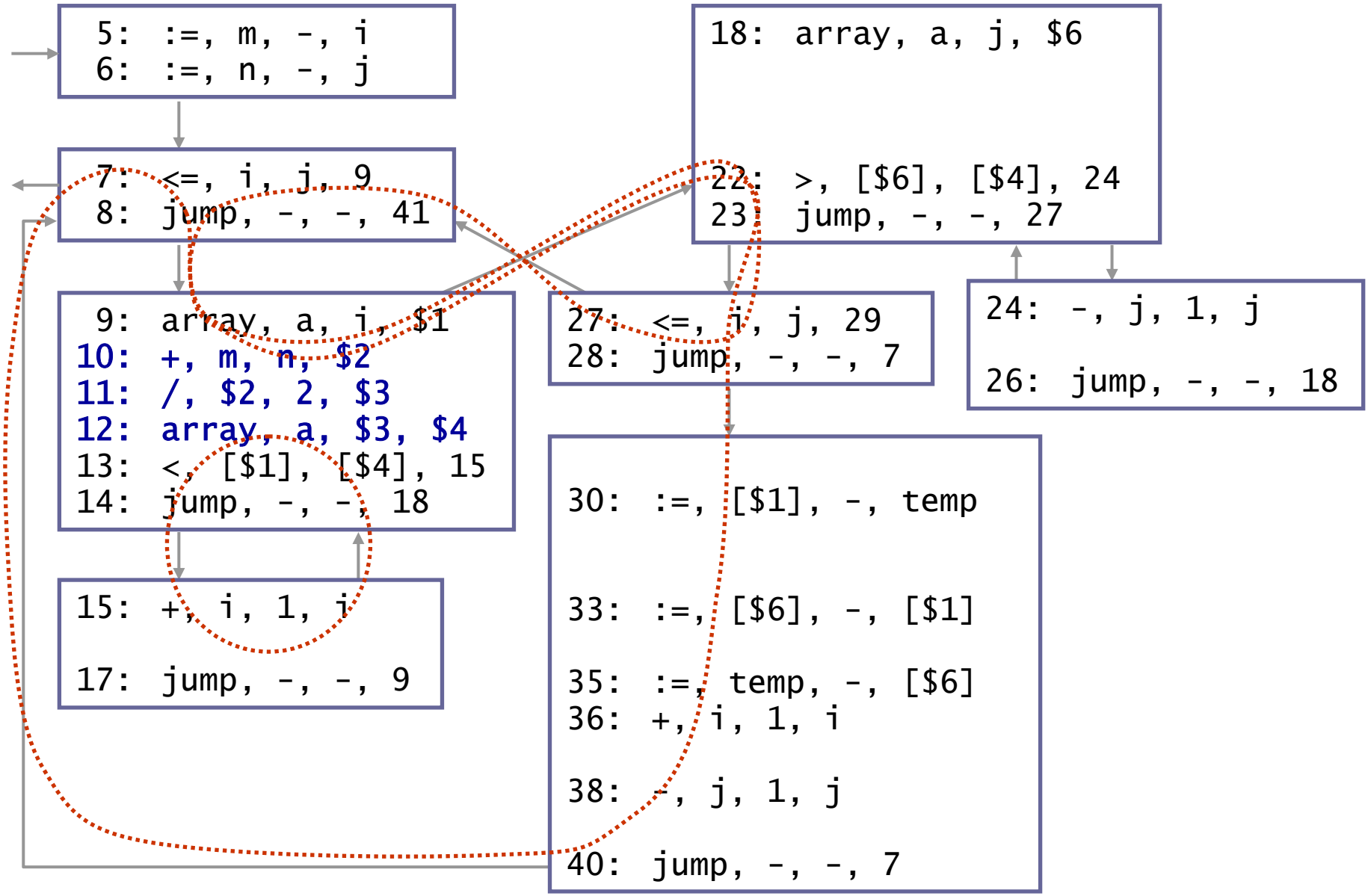
- **Ιδέα**: κώδικας που εκτελεί υπολογισμούς αναλλοίωτους κατά την εκτέλεση βρόχων μετακινείται εκτός των βρόχων
- **Προϋπόθεση**: να μην προκαλείται μεταβολή της σημασίας του προγράμματος

```
while i <= limit-2 do
begin
  s := s + a[i] - 4*n;
  i := i - 1
end
```

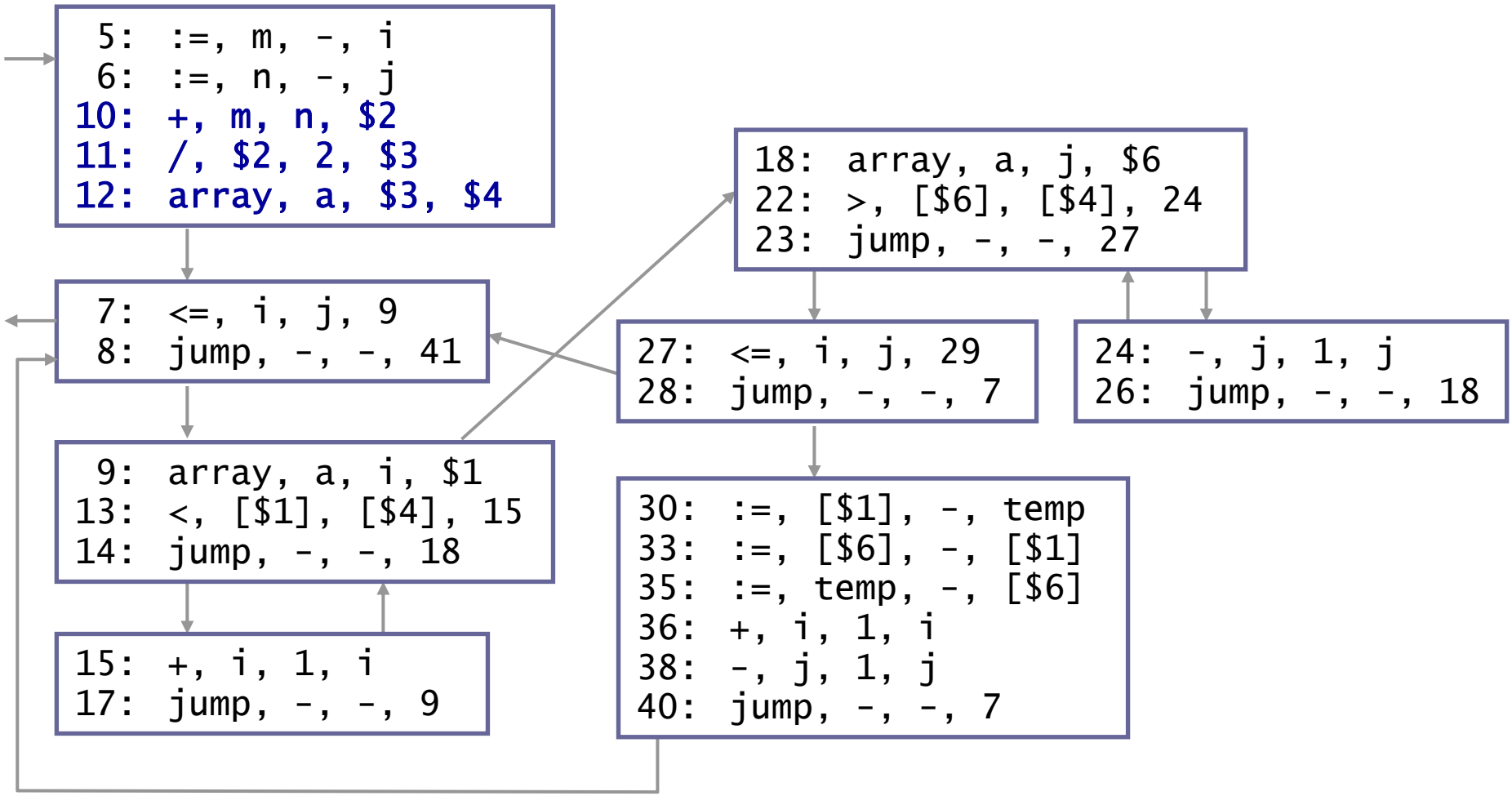


```
t1 := limit-2;
if i <= t1 then
  t2 := 4*n;
while i <= t1 do
begin
  s := s + a[i] - t2;
  i := i + 1
end
```

Μετακίνηση κώδικα (ii)



Μετακίνηση κώδικα (iii)



Απαλοιφή επαγωγικών μεταβλητών (i)

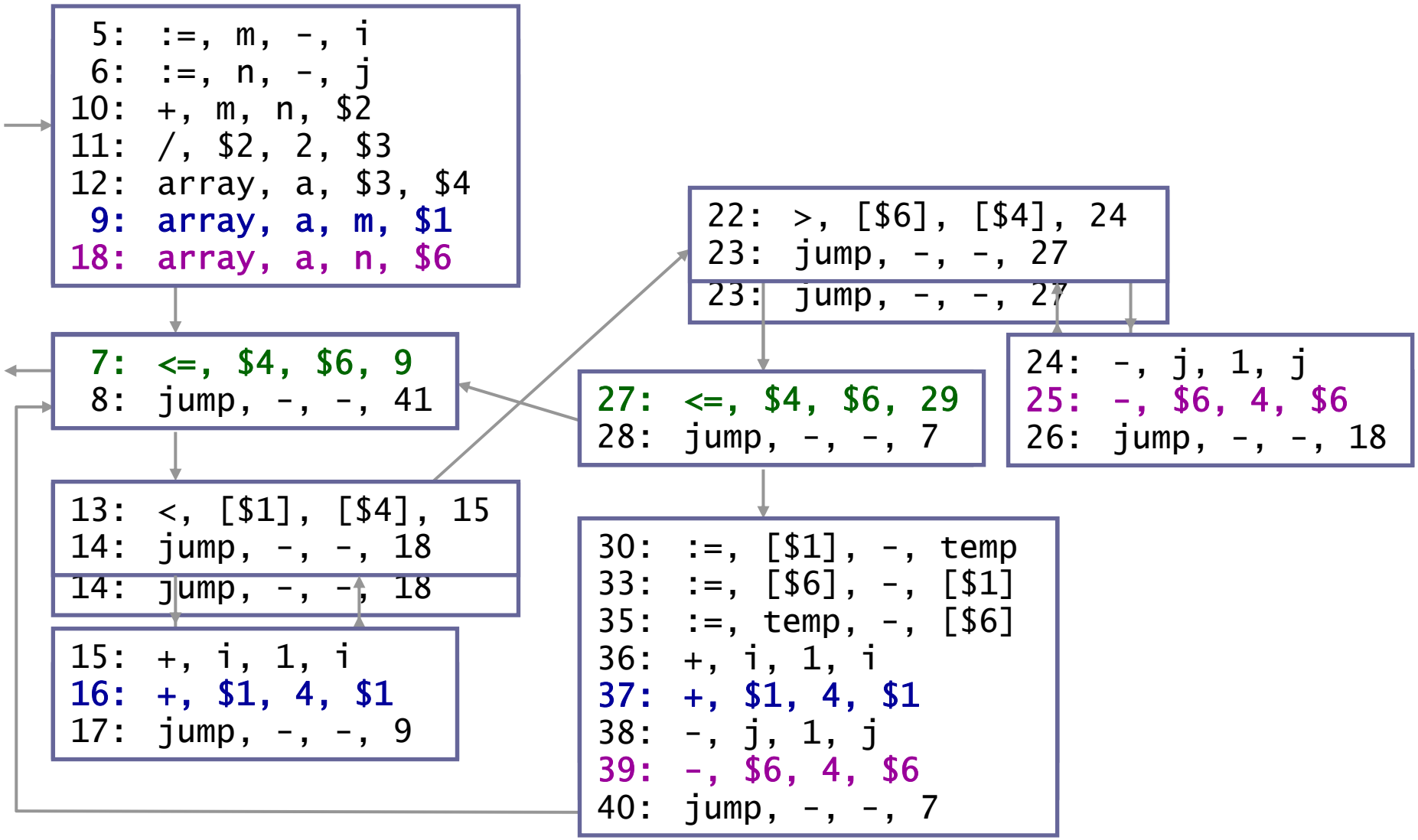
- **Επαγωγική μεταβλητή**: χρησιμοποιείται στο εσωτερικό ενός βρόχου και η τιμή της ορίζει μια αριθμητική πρόοδο
- **Ιδέα**: αν σε ένα βρόχο υπάρχουν περισσότερες επαγωγικές μεταβλητές, διατηρείται μία και οι άλλες υπολογίζονται μέσω αυτής

```
for i := 1 to n do
begin
  x := x + 5
  ...
end
```

$_0$ + 5*(i-1)

- **Υποβιβασμός ισχύος**: διατηρούμε τις μεταβλητές που υπολογίζονται ευκολότερα

Απαλοιφή επαγωγικών μεταβλητών (ii)

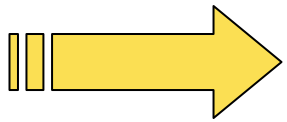


$1 = a + 4*i $6 = a + 4*j i ≤ j ≡ $4 ≤ $6

Αναδιοργάνωση βρόχων

- **Απαλοιφή βρόχου**: αν το σώμα είναι κενό
- **Ξετύλιγμα βρόχου**: αν είναι γνωστό (και μικρό) το πλήθος των επαναλήψεων
- **Αντιστροφή βρόχου**: `while` σε `repeat`
- **Αποδιακλάδωση**: μετακίνηση αναλλοίωτης `if`

```
for i := 1 to 100 do
  if x > 0 then
    s := s + a[i]
  else
    s := s - 1
```



```
if x > 0 then
  for i := 1 to 100 do
    s := s + a[i]
else
  for i := 1 to 100 do
    s := s - 1
```

```
if x > 0 then
  for i := 1 to 100 do
    s := s + a[i]
else
  s := s - 100
```

Απαλοιφή ελέγχου ορίων πίνακα

- **Παρατήρηση**: για γλώσσες που κάνουν τέτοιους ελέγχους και τους υλοποιούν με ενδιάμεσο κώδικα

```
var a : array [1..100] of integer;
...
     a[i]
...
for i := 2 to n do
  a[i] := a[i-1] + 1
```

- $i \geq 2 \rightarrow i \geq 1$
- $i-1 \geq 1$
- $i \leq 100 \rightarrow i-1 \leq 100$

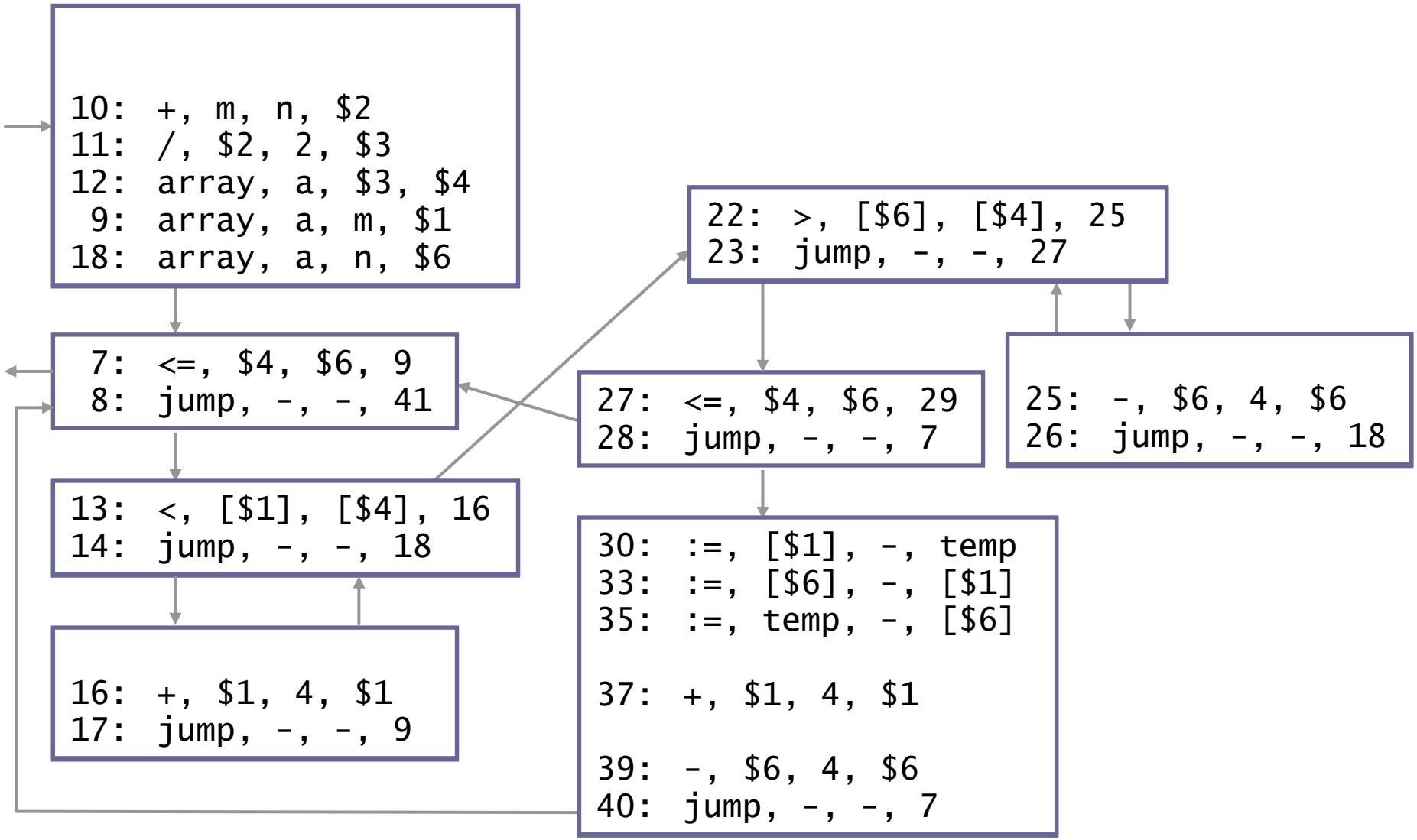
```
   1:   :=, 2, -, i
   2:   >, i, n, 13
   3:   <, i, 1, 9999
   4:   >, i, 100, 9999
   5:   array, a, i, $1
   6:   -, i, 1, $2
   7:   <, $2, 1, 9999
   8:   >, $2, 100, 9999
   9:   array, a, $2, $3
  10:   +, [$3], 1, [$1]
  11:   +, i, 1, i
  12:   jump, -, -, 2
  13:   ...

9999: ...
```

Απαλοιφή άχρηστου κώδικα (i)

- **Απροσπέλαστος κώδικας**: κώδικας που ποτέ δεν εκτελείται
  - εντοπίζεται κατά τη διάρκεια της ανάλυσης ροής ελέγχου
- **Άχρηστος κώδικας**: ακολουθία εντολών που υπολογίζουν τιμές που δε χρησιμοποιούνται
- **Ανάλυση χρόνου ζωής**
  - εντοπισμός μεταβλητών που περιέχουν άχρηστες τιμές
  - γίνεται κατά τη διάρκεια της ανάλυσης ροής δεδομένων

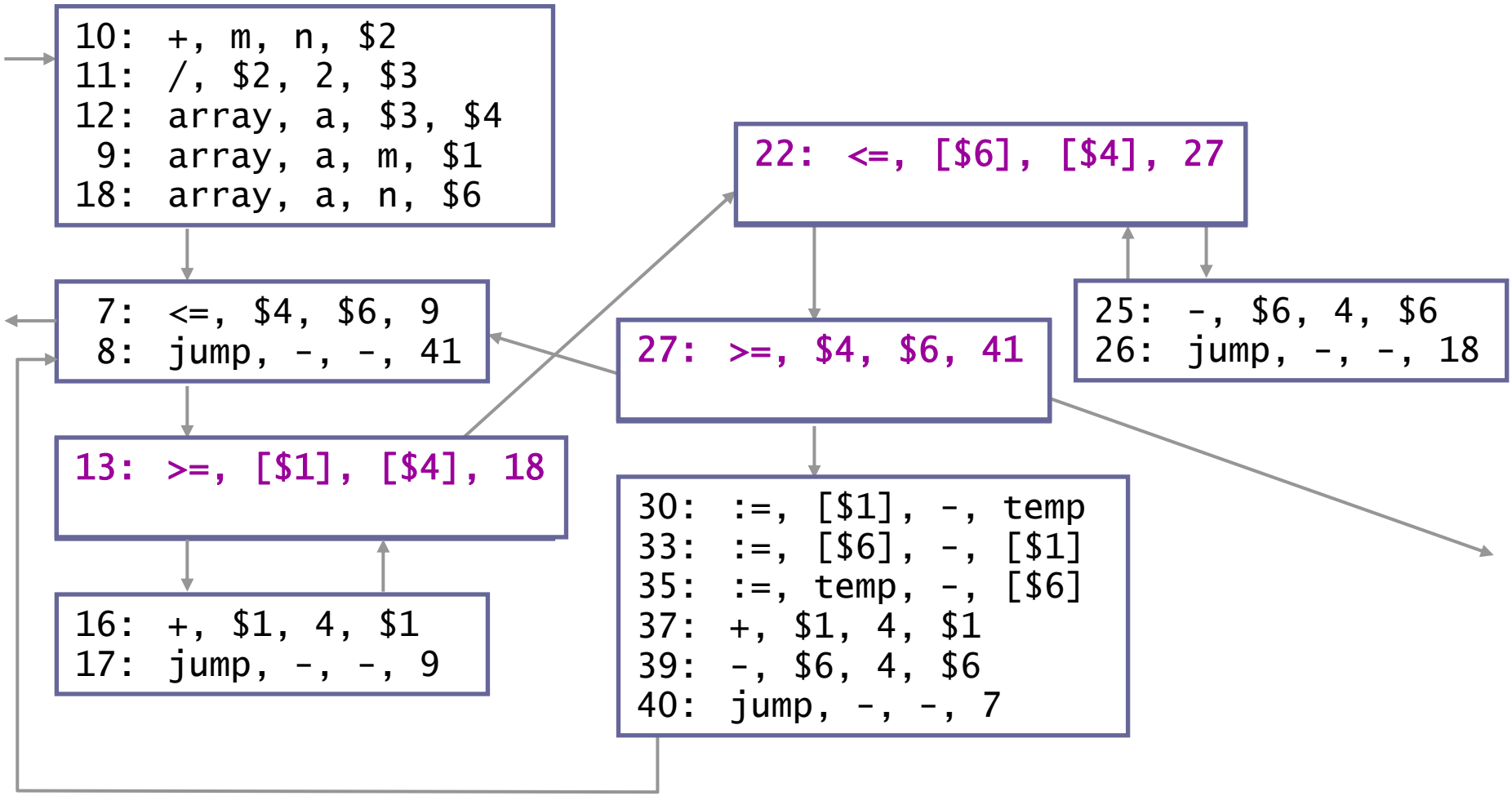
Απαλοιφή άχρηστου κώδικα (ii)



Ευθυγράμμιση

- Συνένωση βασικών ενοτήτων B_i και B_j όταν στο γράφο ροής ελέγχου
  - από την B_i εξέρχεται μόνο μια ακμή και πηγαίνει στην B_j
  - στην B_j εισέρχεται μόνο μια ακμή και είναι από την B_i
- Ο κώδικας των B_i και B_j συνενώνεται
- Μερικές φορές απαιτεί απαλοιφή αλμάτων και αναδιάταξη περισσότερων βασικών ενοτήτων

Απλοποίηση συνθηκών και αλμάτων



- **Απλοποίηση συνθήκης**: τετράδα 28, βλέπε 27, 7 και 8
- **Απλοποίηση αλμάτων**: τετράδες 13, 22 και 27

Ενσωμάτωση υποπρογράμματος

- (inline expansion)

```
function max (a, b : integer) : integer;
begin
  if a >= b then
    result := a
  else
    result := b
end;
...
y := max(x, 3)
```

```
if x >= 3 then
  y := x
else
  y := 3
```

Κλήσεις ουράς και συνένωση (i)

- (tail calls, tail recursion)

```
procedure p (x : integer);
begin
  ...
  q(x+1)
end;
```

κλήση ουράς

```
procedure q (y : integer);
begin
  while y > 2 do
    r(y)
end;
```

όχι κλήση ουράς!

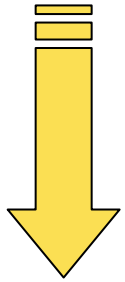
```
function gcd (n, m : integer) : integer;
begin
  if (n = 0) or (m = 0) then
    result := n + m
  else if n > m then
    result := gcd(m, n mod m)
  else
    result := gcd(n, m mod n)
end;
```

αναδρομικές κλήσεις ουράς!

Κλήσεις ουράς και συνένωση (ii)

- Από απλή αναδρομή σε αναδρομή ουράς

```
function factorial (n : integer) : integer;
begin if n > 1 then result := n * factorial(n-1)
               else result := 1
end;
```



```
function factorial (n : integer) : integer;

  function tailrec (prod, n : integer) : integer;
  begin if n > 1 then result := tailrec(n*prod, n-1)
        else result := prod
  end;

begin result := tailrec(1, n);
end;
```

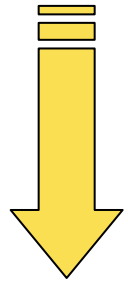
Κλήσεις ουράς και συνένωση (iii)

- Απαλοιφή αναδρομής ουράς

```
function factorial (n : integer) : integer;

  function tailrec (prod, n : integer) : integer;
  begin if n > 1 then result := tailrec(n*prod, n-1)
                 else result := prod
  end;

begin result := tailrec(1, n);
end;
```



```
function factorial (n : integer) : integer;

  function tailrec (prod, n : integer) : integer;
  begin while n > 1 do begin prod := n*prod; n := n-1 end;
        result := prod
  end;

begin result := tailrec(1, n);
end;
```

Κλήσεις ουράς και συνένωση (iv)

- Συνένωση ουράς

```
function factorial (n : integer) : integer;

  function tailrec (prod, n : integer) : integer;
  begin while n > 1 do begin prod := n*prod; n := n-1 end;
        result := prod
  end;

begin result := tailrec(1, n);
end;
```

```
function factorial (n : integer) : integer;
  var prod : integer;
begin
  prod := 1;
  while n > 1 do
  begin prod := n*prod; n := n-1 end;
  result := prod
end;
```