Part X. Optimization and Code Generation

Basic Blocks

- A ***basic block*** is a sequence of statements executed sequentially from beginning to end
- A ***leader*** is the first statement of a basic block

Determine the set of leaders as follows:

- The **first statement** is a **leader**
- Any statement that is the **label** of a goto statement is a **leader**
- Any statement that follows a **goto** statement is a **leader**

Basic Blocks: Example

```
sum := 0;
i := 0;
L1:
if i >= n goto L2;
sum := sum + a[i];
i := i + 1;
goto L1;
L2:
result := sum
```

Basic Blocks: Example

```
sum := 0;
i := 0;
L1:
if i >= n goto L2;
sum := sum + a[i];
i := i + 1;
goto L1;
L2:
result := sum
```

LEADER
(first statement)

Basic Blocks: Example

```
sum := 0;
i := 0;
L1:
if i >= n goto L2;
sum := sum + a[i];
i := i + 1;
goto L1;
L2:
result := sum
```

LEADER (first statement)

LEADER (label)

Basic Blocks: Example

```
sum := 0;
i := 0;
L1:
if i >= n goto L2;
sum := sum + a[i];
i := i + 1;
goto L1;
L2:
result := sum
```

LEADER (first statement)

LEADER (label)

LEADER (follows a goto statement)

Basic Blocks: Example

```
sum := 0;
i := 0;
L1:
if i >= n goto L2;
sum := sum + a[i];
i := i + 1;
goto L1;
L2:
result := sum
```

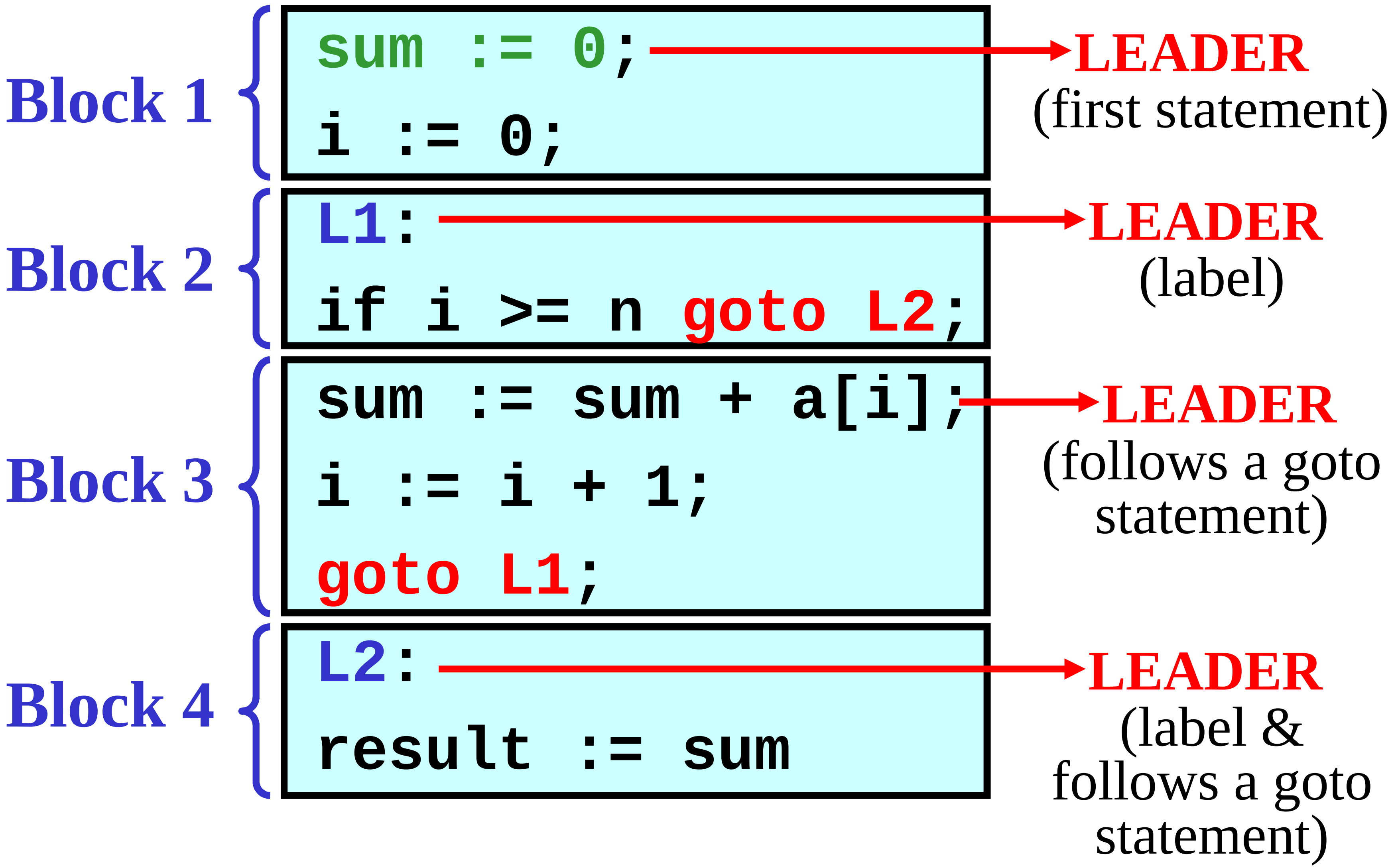
LEADER (first statement)

LEADER (label)

LEADER (follows a goto statement)

LEADER (label & follows a goto statement)

Basic Blocks: Example

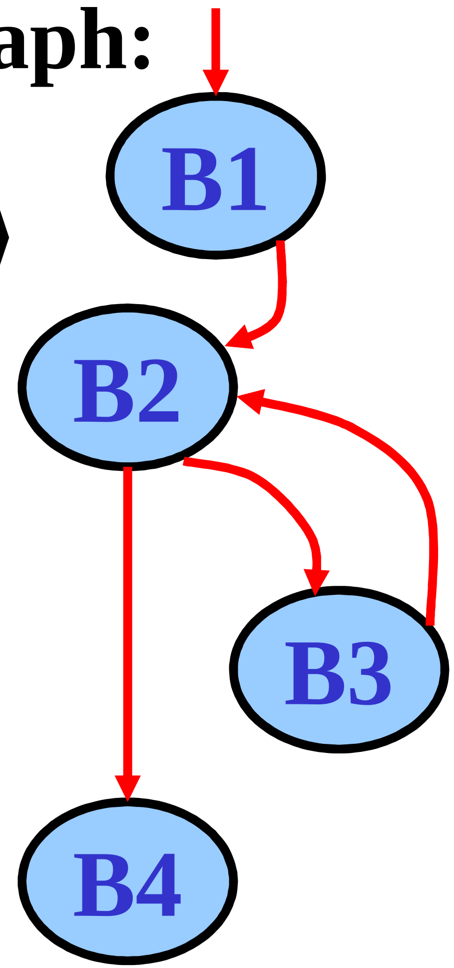


Flow Graph over Blocks

Program with basic blocks:

```
B1  sum := 0;
    i := 0;
B2  L1:
    if i >= n goto L2;
B3      sum := sum + a[i];
        i := i + 1;
        goto L1;
B4  L2:
    result := sum
```

Flow control graph:



Note: Isolated blocks in a flow graph = **dead code**

Optimization: Introduction

Gist: ***Optimizer*** makes a more efficient version of the intermediate or target code

---

Variants of optimizations:

1) Local optimization × **Global optimization**

- Local optimization – within a basic block
- Global optimization – span several basic blocks

2) Optimization for speed × **Optimization for size**

Optimization methods:

1) Constant folding
2) Constant propagation
3) Copy propagation
4) Loop invariant expressions
5) Loop unrolling
6) Dead code elimination

Optimization Methods 1/3

1) Constant folding

```
a := 1;
b := 2;
c := a + b;
```

⇒

```
c := 3;
```

2) Constant propagation

```
a := 3;
b := a;
c := b;
```

⇒

```
c := 3;
```

3) Copy propagation

```
a := x;
b := a;
c := b;
```

⇒

```
c := x;
```

Optimization Methods 2/3

4) Loop invariant expressions

```
for i := 1 to 100 do
  a[i] := p*q/r + i
```

```
x := p*q/r
for i := 1 to 100 do
  a[i] := x + i
```

5) Loop unrolling

```
for i := 1 to 100 do
begin
  for j := 1 to 2 do
    write(x[i, j]);
end;
```

```
for i := 1 to 100 do
begin
  write(x[i, 1]);
  write(x[i, 2]);
end;
```

Optimization Methods 3/3

6) Dead code elimination

- **Dead code:** **a)** Never executed
  b) Does nothing useful

ad **a)**

```
trace := false;
if trace then begin
   writeln(…);
   …
end;
```

⇒ **nothing**

ad **b)**

```
x := x;
```

⇒ **nothing**

Optimization For Size

- This optimization only makes a shorter program

Example:

```
case p of
  1: u := a*b * c;
  2: v := a*b + c;
  3: x := d - a*b;
  4: y := a*b / d;
  5: z := 2 * a*b;
end;
```

```
T := a*b;
case p of
  1: u := T * c;
  2: v := T + c;
  3: x := d - T;
  4: y := T / d;
  5: z := 2 * T;
end;
```

- **Note:** `(a*b)` is very busy.

Code Generation: Introduction

Variants of code genarations:

• **Blind generation** **vs.** **Context-sensitive generation**

1) Blind generation

• For every 3AC instruction, there is a procedure that generates the corresponding target code

Main disadvantage:

• As each 3AC instruction is out of the basic block context, a lot of redundant loading and storing occur

2) Context-sensitive generation

• Reduction of loading and storing between registers and memory.

Blind Generation: Example

3AC:	Generated code:
(+ , a, b, r)	load r_i, a add r_i, b store r_i, r
(* , a, b, r)	load r_i, a mul r_i, b store r_i, r
(:=, a, , r)	load r_i, a store r_i, r

Blind Generation

Example:

3AC:

```
(+,a,b,c)
(*,c,d,e)
```

Generated target code:

```
load  r1, a
add   r1, b
store r1, c
load  r1, c
mul   r1, d
store r1, e
```

Blind Generation

Example:

3AC:

```
(+,a,b,c)
(*,c,d,e)
```

Generated target code:

load r1, a
add r1, b
store r1, c
~~load r1, c~~
mul r1, d
store r1, e

A redundant instruction

Context-Sensitive Generation (CSG)

- Minimization of loading and storing between registers and memory:
- **General rule:** If a value is in a register and will be used "***soon***", keep it in the register

Information needed :

1) **Question:** Which variables are needed later in the block and where?
Answer is in the ***Basic block table*** (**BBT**)

2) **Q:** Which registers are in use and what they hold?
A is in the ***Register association table*** (**RAT**)

3) **Q:** Where the current value of a variable is to be found?
A is in the ***Address table*** (**AT**)

CSG: Analysis within a Basic Block

• A variable is ***live*** if it is used later in the block

Example:

live

```
(i) a := b + c
    :        :
(j) d := a + b
```

No occurrence of variable "a"

next use

Question: How to detect live variables effectively?

Answer: Apply ***backward finding***—that is, read the instructions from the block end towards its begin

Symbol Table (ST)

Extetion of a ST:

variable	*status*	*next use*
a	**live**	**(10)**
b	**live**	**(20)**
pos	**dead**	**none**
⋮	⋮	⋮

***Status*:** **live** / **dead**

***Next use*:** **none** / **(*i*)**

- *i* = number of a line

Initial assumption:

- All programmer variables: ***Status*:** **live**
- All temporary variables: ***Status*:** **dead**
- All variables: ***Next use*:** **none**

Basic Block Table (BBT)

Structure of a BBT:

line	*instruction*	*status*	*next use*
⋮	⋮		
(*i*)	a:=b+c		
⋮	⋮		

backward

- **Method:**

Suppose that (*i*) is the current instruction:

1) Move ***status*** and ***next use*** of ***a***, ***b***, ***c*** from **ST** to **BBT**

2) In **ST** make these changes:

For variable ***a***:	***Status***: **dead**	***Next use***: **none**
For variables ***b***, ***c***:	***Status***: **live**	***Next use***: (*i*)

Changes in a ST: Illustration

backward

⋮

`(i) a:=b+c`

⋮

ST:

var	*status*	*next use*
a	dead	none
b	live	(*i*)
c	live	(*i*)
⋮	⋮	⋮

---

a is dead because `a:=b+c` **kills** any **previous** definition of ***a***

b, ***c*** are alive and used in (*i*); this information reflects the situation **earlier** in the block

Filling BBT: Example 1/8

$$d := \underbrace{\underbrace{(a-b)}_{u} + \underbrace{(c-a)}_{v}}_{w} - \underbrace{\underbrace{(d+b)}_{x} * \underbrace{(c+1)}_{y}}_{z}$$

BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`		
(2)	`v:=c-a`		
(3)	`w:=u+v`		
(4)	`x:=d+b`		
(5)	`y:=c+1`		
(6)	`z:=x*y`		
(7)	`d:=w-z`		

Filling BBT: Example 2/8

ST - line (7):

program variables: a, b, c, d

temporary variables: u, v, w, x, y, z

var	*status*	*next use*
a	**L**	**N**
b	**L**	**N**
c	**L**	**N**
d	**L** [1]	**N** [3]
u	**D**	**N**
v	**D**	**N**
w	**D** [2]	**N** [3]
x	**D**	**N**
y	**D**	**N**
z	**D** [2]	**N** [3]

L – live

D – dead

N – none

Filling BBT: Example 3/8

BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`		
(2)	`v:=c-a`		
(3)	`w:=u+v`		
(4)	`x:=d+b`		
(5)	`y:=c+1`		
(6)	`z:=x*y`		
(7)	`d:=w-z`	d:L[1]; w,z:D [2]	d,w,z:N [3]

Filling BBT: Example 4/8

ST - line (6):

var	*status*	*next use*
a	**L**	**N**
b	**L**	**N**
c	**L**	**N**
d	**D**	**N**
u	**D**	**N**
v	**D**	**N**
w	**L**	**(7)**
x	**D** [1]	**N** [3]
y	**D** [1]	**N** [3]
z	**L** [2]	**(7)** [4]

Filling BBT: Example 5/8

BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`		
(2)	`v:=c-a`		
(3)	`w:=u+v`		
(4)	`x:=d+b`		
(5)	`y:=c+1`		
(6)	`z:=x*y`	z:L[2]; x,y:D[1]	z:7[4]; x,y:N[3]
(7)	`d:=w-z`	d:L; w,z:D	d,w,z:N

Filling BBT: Example 6/8

ST - line (5):

var	*status*	*next use*
a	L	N
b	L	N
c	L [1]	N [2]
d	D	N
u	D	N
v	D	N
w	L	(7)
x	L	(6)
y	L [1]	(6) [3]
z	D	N

Filling BBT: Example 7/8

BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`		
(2)	`v:=c-a`		
(3)	`w:=u+v`		
(4)	`x:=d+b`		
(5)	`y:=c+1`	y,c:L [1]	y:6 [3]; c:N [2]
(6)	`z:=x*y`	z:L; x,y:D	z:7; x,y:N
(7)	`d:=w-z`	d:L; w,z:D	d,w,z:N

- **Fill the rest analogically.**

Filling BBT: Example 8/8

Final BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`	u,a,b:L	u:3; a:2; b:4
(2)	`v:=c-a`	v,c,a:L	v:3; c:5; a:N
(3)	`w:=u+v`	w:L; u,v:D	w:7; u,v:N
(4)	`x:=d+b`	x,b:L; d:D	x:6; d,b:N
(5)	`y:=c+1`	y,c:L	y:6; c:N
(6)	`z:=x*y`	z:L; x,y:D	z:7; x,y:N
(7)	`d:=w-z`	d:L; w,z:D	d,w,z:N

Register Association Table

Structure of a RAT:

reg.	*contents*
0	**reserved**
1	**reserved**
2	**free**
3	**a**
4	**free**
5	**b**
⋮	⋮

***Contents*:** → **reserved** / **free** / **var**$_i$

- **reversed** for some operation system purposes
- **var**$_i$ = name of variable

- Every use of a register updates RAT.
- RAT indicates the current contents of each register

Address Table (AT)

Structure of an AT:

variable	*address*
a	**in memory**
b	**reg. 5**
c	**nowhere**
⋮	⋮

Address*:** → **in memory** / **reg. *i / **nowhere**

• *i* = number of register

• Address table shows where the current value of every variable can be found.

GetReg

- ***GetReg*** returns an optimal register for **b** in **a := b + c**

***GetReg*:**
begin
  if b is in register R **and** b is dead **and**
    b has no next use **then** return R
  else
  if there is any free register R **then** return R
  else begin
    - select an occupied register R
    - save R's current contents
    - update RAT & AT
    - return R

  end;
end;

GenCode

• ***GenCode*** generate an optimal code for command `a:=b+c`

GenCode:
begin

- Ask ***GetReg*** for a register `R` for `b`
- **if** `b` is not in `R` **then** generate `load R,b`
- **if** `c` is in reg. `S` **then** generate `add R,S`
  else generate `add R,c`
  {= c is in memory}
- Update RAT & AT to indicate that current value of `a` is in `R`
- **if** `c` is in `S` **and** `c` is dead **and** has no next use **then** mark `S` as free in RAT

end;

GetReg and *GenCode*: Example 1/10

BBT:

line	*instruction*	*status*	*next use*
(1)	`u:=a-b`	u,a,b:L	u:3; a:2; b:4
(2)	`v:=c-a`	v,c,a:L	v:3; c:5; a:N
(3)	`w:=u+v`	w:L; u,v:D	w:7; u,v:N
(4)	`x:=d+b`	x,b:L; d:D	x:6; d,b:N
(5)	`y:=c+1`	y,c:L	y:6; c:N
(6)	`z:=x*y`	z:L; x,y:D	z:7; x,y:N
(7)	`d:=w-z`	d:L; w,z:D	d,w,z:N

RAT:

reg.	*contents*
0,1	reserved
2-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u-z	nowhere

GetReg and *GenCode*: Example 2/10

Instruction: `(1) u:=a-b`
Properties: **u, a, b: live**

***GetReg*:** **R2**
***GenCode*:**
```
load R2,a
sub  R2,b
```

RAT:

reg.	*contents*
0,1	reserved
2-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u-z	nowhere

GetReg and *GenCode*: Example 3/10

Instruction: `(2) v:=c-a`
Properties: **v, c, a: live**

***GetReg*:** **R3**
***GenCode*:**

```
load R3,c
sub  R3,a
```

RAT:

reg.	*contents*
0,1	reserved
2	u
3-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u	2
v-z	nowhere

GetReg and *GenCode*: Example 4/10

Instruction: `(3) w:=u+v`
Properties: **w: live; u, v: dead**

***GetReg*:** **R2**
***GenCode*:** `add R2,R3`

RAT:

reg.	*contents*
0,1	reserved
2	u
3	v
4-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u	2
v	3
w-z	nowhere

GetReg and *GenCode*: Example 5/10

Instruction: `(4) x:=d+b`

Properties: **x, b: live; d: dead**

***GetReg*:** **R3**

***GenCode*:**

```
load R3,d
add  R3,b
```

RAT:

reg.	***contents***
0,1	reserved
2	w
3-11	free
12-15	reserved

AT:

var.	***address***
a-d	in memory
u, v	nowhere
w	2
x-z	nowhere

GetReg and *GenCode*: Example 6/10

Instruction: `(5) y:=c+1`
Properties: **y, c: live**

***GetReg*:** **R4**
***GenCode*:**
```
load R4,c
add  R4,#1
```

RAT:

reg.	*contents*
0,1	reserved
2	w
3	x
4-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u, v	nowhere
w	2
x	3
y, z	nowhere

GetReg and *GenCode*: Example 7/10

Instruction: `(6) z:=x*y`
Properties: **z: live; x, y: dead**

***GetReg*:** **R3**
***GenCode*:** `mul R3,R4`

RAT:

reg.	*contents*
0,1	reserved
2	w
3	x
4	y
5-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u, v	nowhere
w	2
x	3
y	4
z	nowhere

GetReg and *GenCode*: Example 8/10

Instruction: `(7) d:=w-z`
Properties: **d: live; w, z: dead**

***GetReg*:** **R2**
***GenCode*:** `sub R2,R3`

RAT:

reg.	*contents*
0,1	reserved
2	w
3	z
4-11	free
12-15	reserved

AT:

var.	*address*
a-d	in memory
u, v	nowhere
w	2
x, y	nowhere
z	3

GetReg and *GenCode*: Example 9/10

Instruction: ***end of block***
Properties: **d: live;**

***GetReg*:** -
***GenCode*:** `store R2,d`
(save all live variables!)

RAT:

reg.	*contents*
0,1	reserved
2	d
3-11	free
12-15	reserved

AT:

var.	*address*
a-c	in memory
d	2
u-z	nowhere

GetReg and *GenCode*: Example 10/10

- **Resulting Code: 12 instructions instead of 21**

Line	*3AC*	*generated code*
(1)	`u:=a-b`	`load R2, a` `sub R2, b`
(2)	`v:=c-a`	`load R3, c` `sub R3, a`
(3)	`w:=u+v`	`add R2, R3`
(4)	`x:=d+b`	`load R3, d` `add R3, b`
(5)	`y:=c+1`	`load R4, c` `add R4, #1`
(6)	`z:=x*y`	`mul R3, R4`
(7)	`d:=w-z`	`sub R2, R3` `store R2, d`

Parallel Compilers: Introduction

- ***Lexical analyzer*** translates a **complete** source program into tokens

---

- **Preparation of the syntax analysis in parallel:**
  - A separation of some substrings of tokens. These substrings and the rest, called the program ***skeleton***, are parsed in parallel.
  - In the skeleton, the removed substrings are replaced with ***pseudotokens***.

Parallel Compilers: Separation of Conditions

:
if $cond_1$ **then …**
:
while $cond_2$ **do …**
:
repeat … until $cond_3$
:

:
if [$cond$, 1] **then …**
:
while [$cond$, 2] **do …**
:
repeat … until [$cond$, 3]
:

- **Table of condition:**

1	$cond_1$
2	$cond_2$
3	$cond_3$

Parallel Compilers: Multi-Level Separation

```
:
if a + b > c * d and a – b = c + d then …
:
```

```
:
if [cond, 1] then …
:
```

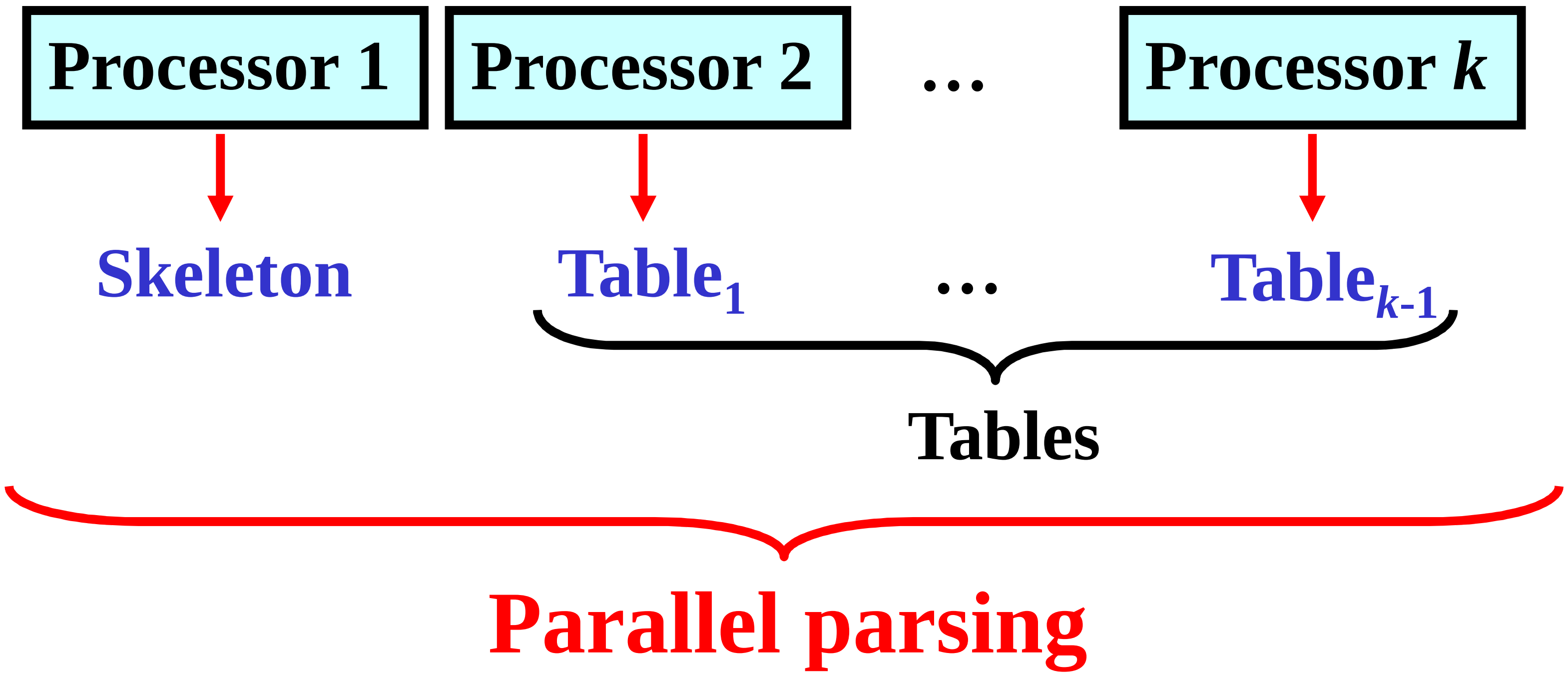
- **Table of expressions:**

1	**a + b**
2	**c * d**
3	**a - b**
4	**c + d**

- **Table of condition:**

1	[*expr*, 1] > [*expr*, 2] and [*expr*, 3] = [*expr*, 4]
2	…

Parallel Compilers: Parsing



- **different methods 1 – *k***
- **different intermediate codes**