Code optimization Module 5

Code optimization improves the intermediate code.

→ Less space and time.

Code optimization techniques:

1) Common sub expression elimination.
2) Constant folding.
3) Copy propagation
4) Dead code elimination.
5) Code Motion
6) Induction variable elimination and Reduction in strength.

These are common examples of such function-preserving or semantics-preserving transformations.

Code optimization :-

Elimination of unnecessary instructions in object code, or replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called 'code improvement' or 'code optimizations'.

Local code optimization :-

Code improvement within a basic block.

Global code optimization :-

Mostly based on data-flow analyses.

1) Common sub-expression elimination

Common subexpression –

i) if it was previously computed.

ii) Values of variables have not changed.

```
a = b+c
b = a-d
c = b+c
d = a- d
```

⟹

```
a = b+c
b = a-d
c = b+c
d = b
```

2) Constant folding

If the value of an expression is a constant, use the constant instead of expression.

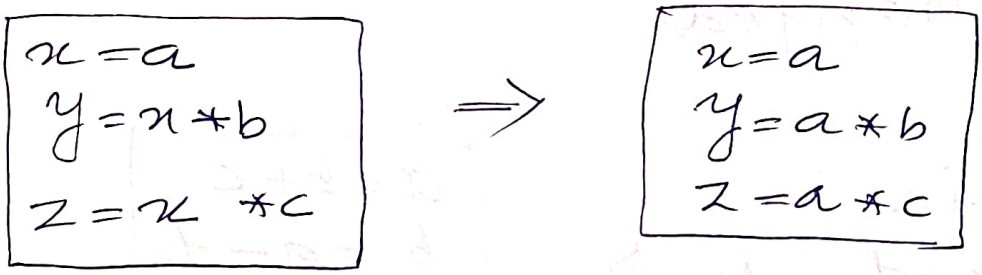
$PI = \frac{22}{7}$ ⟹ $PI = 3.14$

3. Copy Propagation

If $f = g$ // copy statement.

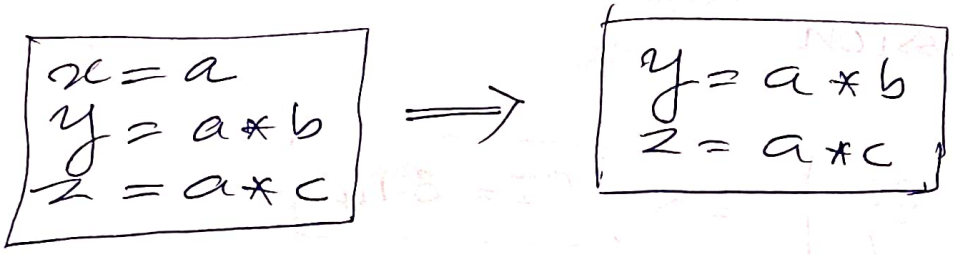
use g for f after $f = g$.

Eg:-



4. Dead code elimination

A variable is live, if it's value can be used subsequently; otherwise it is dead.



here, x is a dead variable, as we're no longer using 'x' in the computations.

5. Code motion

→ moves code outside a loop.

```
while(i<10)
{
    x = y+z;
    i = i+1;
}
```

// this statement not working based on the loop condition.

So can move outside.

```
x = y+z;
while (i<10)
{
    i = i+1;
}
```

6. Induction variables and Reduction in strength:

A Variable 'x' is said to be an 'induction variable' if there's a positive or negative constant c such that each time x is assigned, its value increases by c.

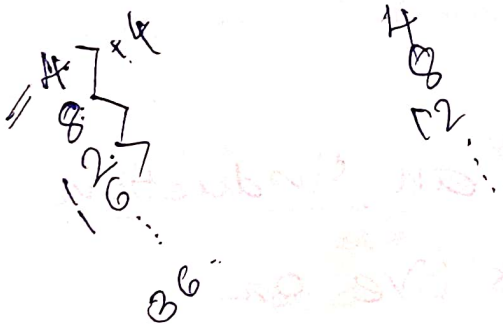
The transformation of replacing an expensive operation by cheaper one, is known as Strength reduction.
(multiplication can be replaced by addition)

Eg:-

```
i = 1;
while (i < 10)
{
    t = i * 4;
    i = i + 1;
}
```

⇒

```
t = 4;
while (t < 40)
{
    t = t + 4;
}
```



Loop optimization

Most execution time of a program is spent on loops. Decreasing the no. of instructions in an inner loop improves the running time of a program.

Loop optimization techniques:

1. Code motion.
2. Induction variable elim. & reduction in strength.
3. Loop unrolling
4. Loop Jamming.

3. Loop unrolling

Duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested.

```
for (i=0; i<100; i++)
{ display();
}
```

```
for(i=0; i<50; i++)
{
    display();
    display();
}
```

4) Loop Jamming / merging

Combines the bodies of 2 adjacent loops that would iterate the same no. of times.

```
for (i=0; i<100; i++)
  a[i] = 1;
for (i=0; i<100; i++)
    b[i]=2
```

```
for (i=0; i<100; i++)
{
    a[i] = 1
    b[i] = 2;
}
```

Peephole optimization

This technique is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) and replace the instructions by a faster or short sequence of instructions.

peephole optimization techniques:

1) Redundant Instruction Elimination

Eg:- Mov R0, x
Move x, R0 ⟶ this code part is redundant.

But if it's labelled as L1 : Mov R0, x
L2 : Mov x, R0.

we can't say that these code is redundant. It may execute at some other time.

2) Unreachable code elimination

```
x=0;
if (x==1)
{a=b;
}
```

already initialized x as 0. So this block is unreachable.

So we can change the code into optimized code as:

```
x=0;
if x==1 goto L1
goto L2
L1 : a=b;
L2 :
```

```
x=0
if x≠1 goto L2
a=b;
L2:
```

Jump to L2.

removing unwanted goto's.

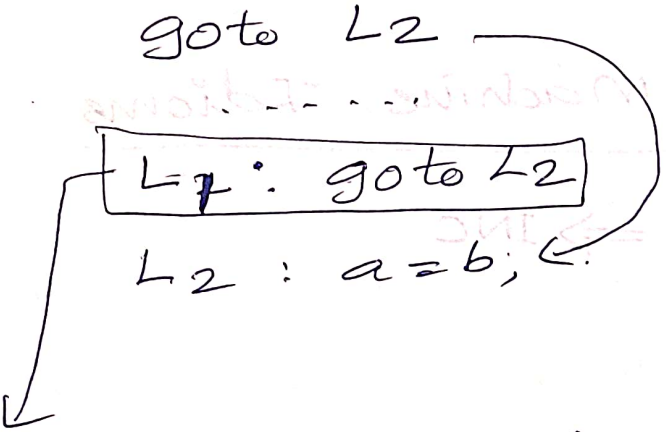
So a=b is unreachable.
So can safely remove.

3) Flow-of-control optimization

Eg:

```
    goto L1
    ....
L1: goto L2
    ....
L2: a=b;
```

We can rewrite it as



If this statement is not a target of any other statement, then this can be removed.

4) Algebraic simplifications

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

} these codes will not change the value of x; so can be removed.

Reduction in strength

x ↑ 2 ⇒ x * x

2 * x ⇒ x + x.

x * 2 ⇒ x << 1

$x * 2^1 \Rightarrow x << 1$

left shift

$x * 4 \Rightarrow x << 2$ ($4 = 2^2$)

$x * 8 \Rightarrow x << 3$ ($8 = 2^3$)

$x / 2 \Rightarrow x >> 1$

$x / 4 \Rightarrow x >> 2$

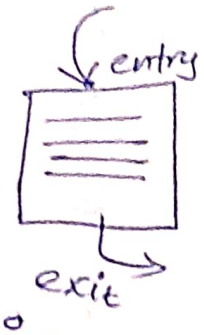
right shift

5) Use of Machine Idioms

$i = i + 1 \Rightarrow$ INC

Basic Blocks and flow graphs.

- A basic block is a collection of sentences which are executed in sequential manner.
- Any basic block must contain one entry and exit points.



- Any basic block should not contain conditional or unconditional statements in the middle of the block.

conditional ⟶ if, if-else, for, while...

unconditional ⟶ break, continue, go to,...

$x = a+b+c$

$t_1 = a+b$

$t_2 = t_1 + c$

$x = t_2$

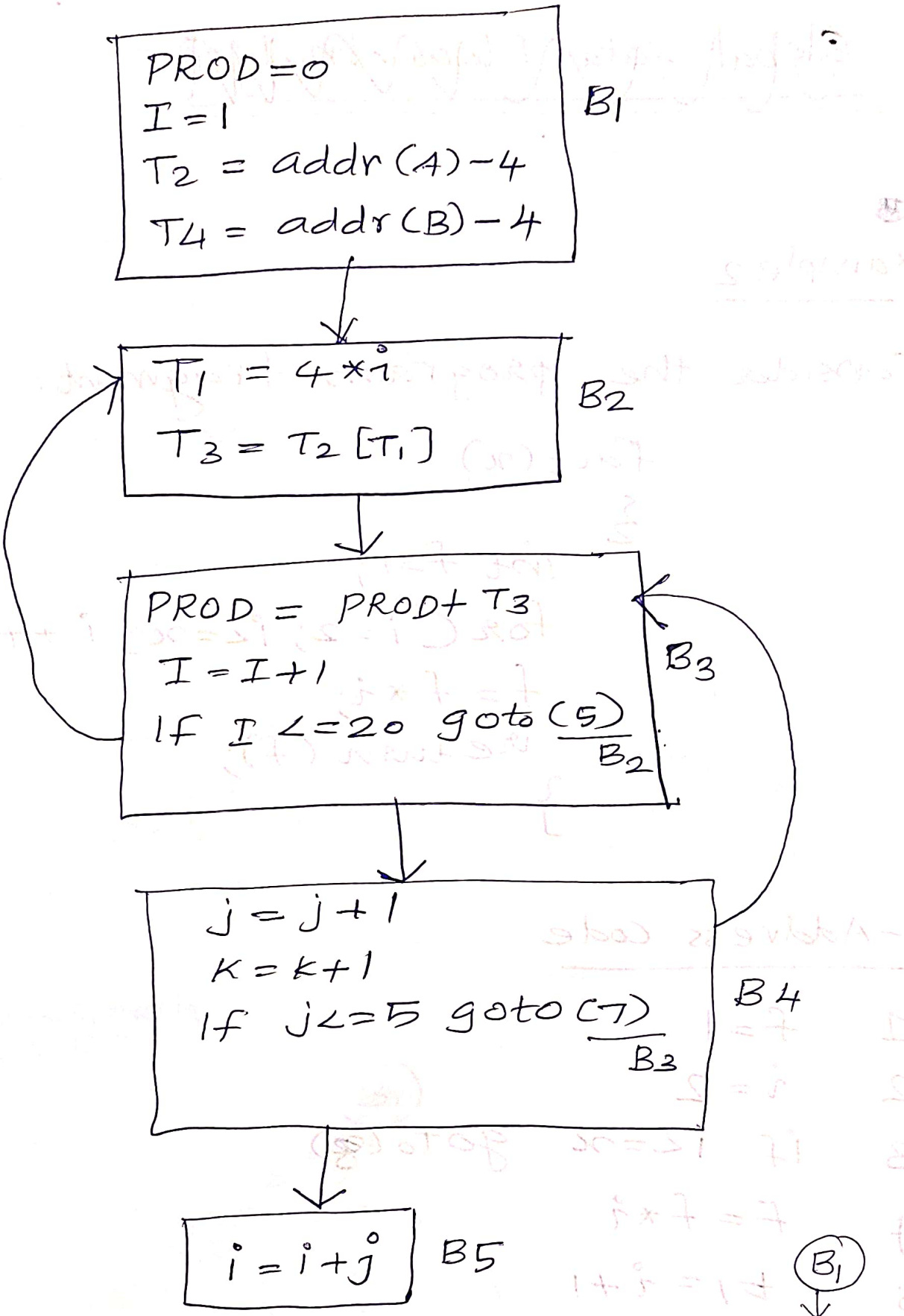
⟩ converted to 3Address code.

Flow graphs

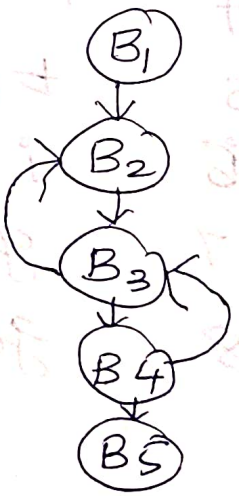
1. PROD = 0 ⟶ 1st stmt is a leader.
2. I = 1
3. T_2 = addr (A) − 4
4. T_4 = addr (B) − 4
5. T_1 = 4 * i
6. T_3 = T_2 [T_1]
7. PROD = PROD + T_3
8. I = I + 1
9. If I <= 20 goto (5)
10. j = j + 1
11. k = k + 1
12. IF j <= 5 goto (7)
13. i = i + j

target of conditional or unconditional jump ~~state~~ is a leader.

statement following cond. or unconditional jump is a leader.


PROD = 0
I = 1
T2 = addr(A) - 4
T4 = addr(B) - 4
B1
T1 = 4 * i
T3 = T2 [T1]
B2
PROD = PROD + T3
I = I + 1
if I <= 20 goto (5)
B2
B3
j = j + 1
k = k + 1
if j <= 5 goto (7)
B3
B4
i = i + j
B5

A flow graph


B1
B2
B3
B4
B5

Example 2

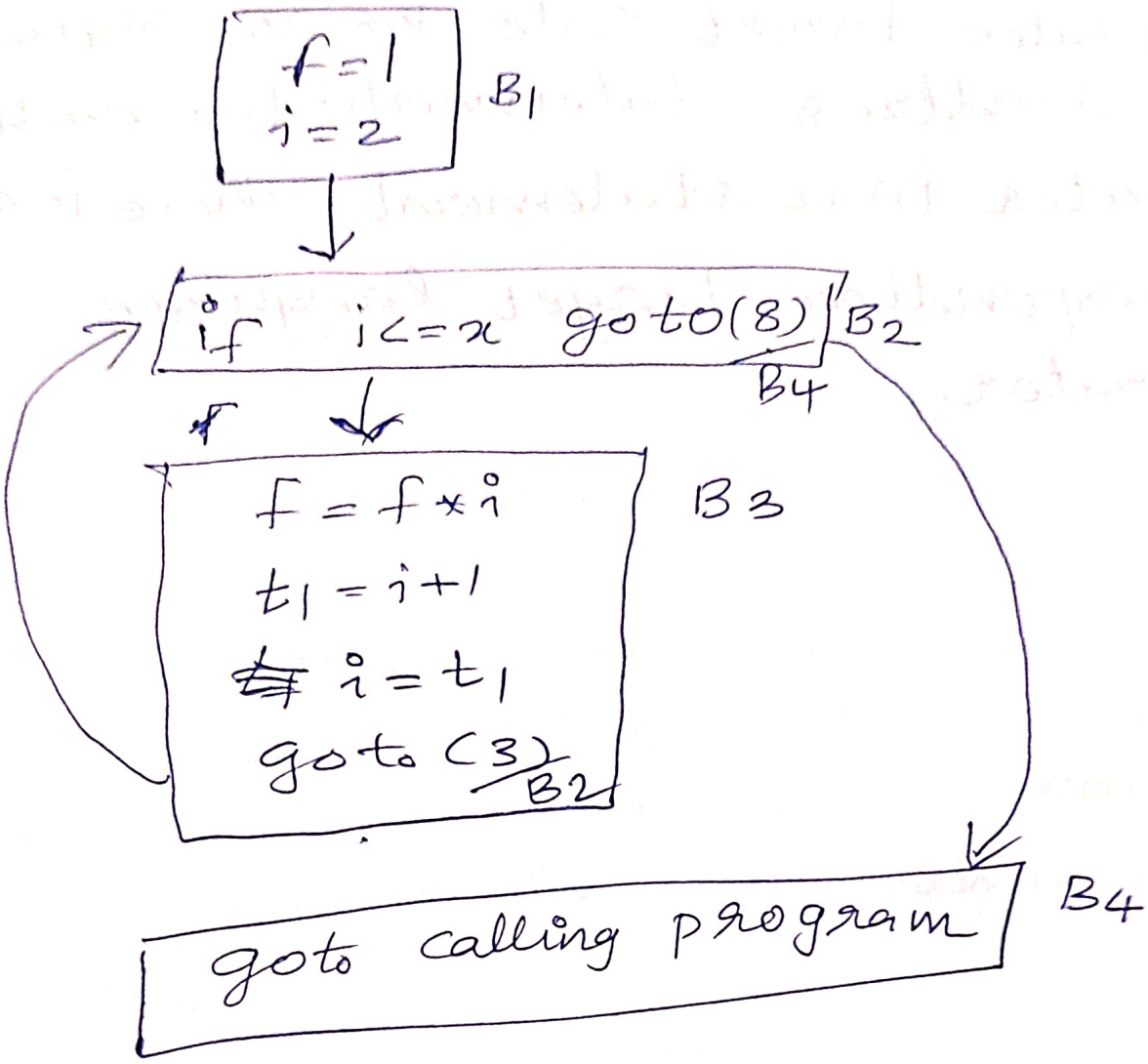
Consider the program fragment:

```
fact(x)
{
    int f=1;
    for(i=2; i<=x; i++)
    f=f*i;
    return(f);
}
```

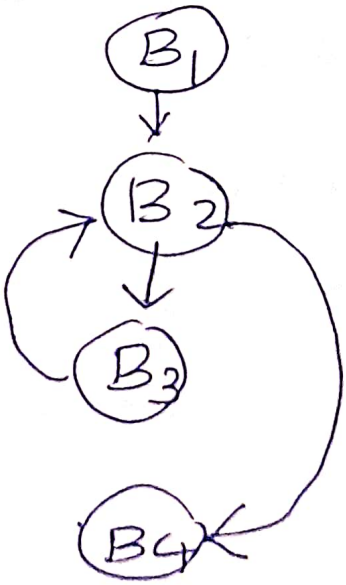
3-Address code

```
1   f=1
2   i=2
3   if i<=x goto (8)
4   f=f*i
5   t1=i+1
6   i=t1
7   goto (3)
8   goto calling program.
```

leader stms ⟶ 1, 3, 4, 8



Flowgraph



The Target Language.

The target computer models a 3AC machine with load and store operations, computation operations, jump operations, and conditional jumps.

The computer is a byte-addressable machine with n-general purpose registers, $R_0, R_1, \ldots R_{n-1}$.

We assume the following kinds of instructions are available.

1) Load operations:

LD dst, addr

2) Store operations:

ST x, r

3) Computation: Operations of the form

OP dst, src_1, src_2

where op is the operator, src_1 and src_2 are the locations.

Eg:- SUB r_1, r_2, r_3

Computes $r_1 = r_2 - r_3$.

→ Unary operators that take only one operand do not have a src_2.

4) Unconditional jumps:-

BR L

Causes control to branch to the machine instruction with label L. (BR stands for Branch)

5) Conditional jumps:-

It is of the form Bcond r, L,

where 'r' is a register, L is a label, and 'cond' stands for any of the common tests on values in the register r.

Eg:- BLTZ r, L

It causes a jump to label L if the value in register r is Less Than Zero. and allows control to pass to the next machine instruction if not.

A simple Code generator

- Generates target code for a sequence of 3-address statements. For each operator in a statement, there is a corresponding target language operator.

A simple Code generator

- Generates target code for a sequence of 3-address statements. For each operator in a statement, there is a corresponding target language operator.

Register and Address Descriptors

1. Register Descriptor:

keeps track of what currently in each register. Initially all registers are empty.

2. Address Descriptor:

keeps track of the location where the current value of the name can be found. Location may be register, a stack location or memory address.

A code Generation Algorithm

For a three-address instruction such as $x = y \text{ op } z$, do the following:

① Use getReg $(x = y + z)$ to select registers for x, y and z. Call these R_x, R_y and R_z.

(2) If y is not in Ry (according to the register descriptor), then issue an instruction LD Ry, y', where y' is one of the memory locations for y (according to the address descriptor)

(3) similarly, if z is not in Rz,

LD Rz, z'

(4) Issue the instruction ADD Rx, Ry, Rz.

Eg:-

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

3-address code

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

~~t_4~~ $d = t_3 + t_2$

Ro , R1

statement	code generated	R.D	A.D
$t_1 = a - b$	MOV a, Ro SUB b, Ro	Reg. ose empty Ro contains t_1	t_1 in Ro
$t_2 = a - c$	MOV a, R1 SUB C, R1	Ro Cont t_1 R1 cont t_2	t_1 in Ro t_2 in R1
$t_3 = t_1 + t_2$	ADD R1, Ro	Ro Cont t_3 R1 cont. t_1	t_2 in R1 t_3 in Ro
~~t_4~~ $= t_3 + t_2$	ADD R1, Ro MOV Ro, d	Ro cont. d	d in Ro.

ST d, Ro ; if 'd' is a global variable.

Example

$x = (a - b) + (a + c)$

3AC

$t_1 = a - b$

$t_2 = a + c$

$t_3 = t_1 + t_2$

$x = t_3$

R_0 R_1

Stmt	Code gen.	Reg. Descri.	Addr. Descript.
$t_1 = a - b$	MOV a, R_0 SUB b, R_0	Reg. empty R_0 : t_1	t_1 in R_0
$t_2 = a + c$	MOV a, R_1 ADD c, R_1	R_0 cont t_1 R_1 cont t_2	t_1 in R_0 t_2 in R_1
$t_3 = t_1 + t_2$	ADD R_1, R_0	R_0 cont t_3	t_3 in R_0
$x = t_3$	MOV t_3, x	R_0 cont x	x is in R_0