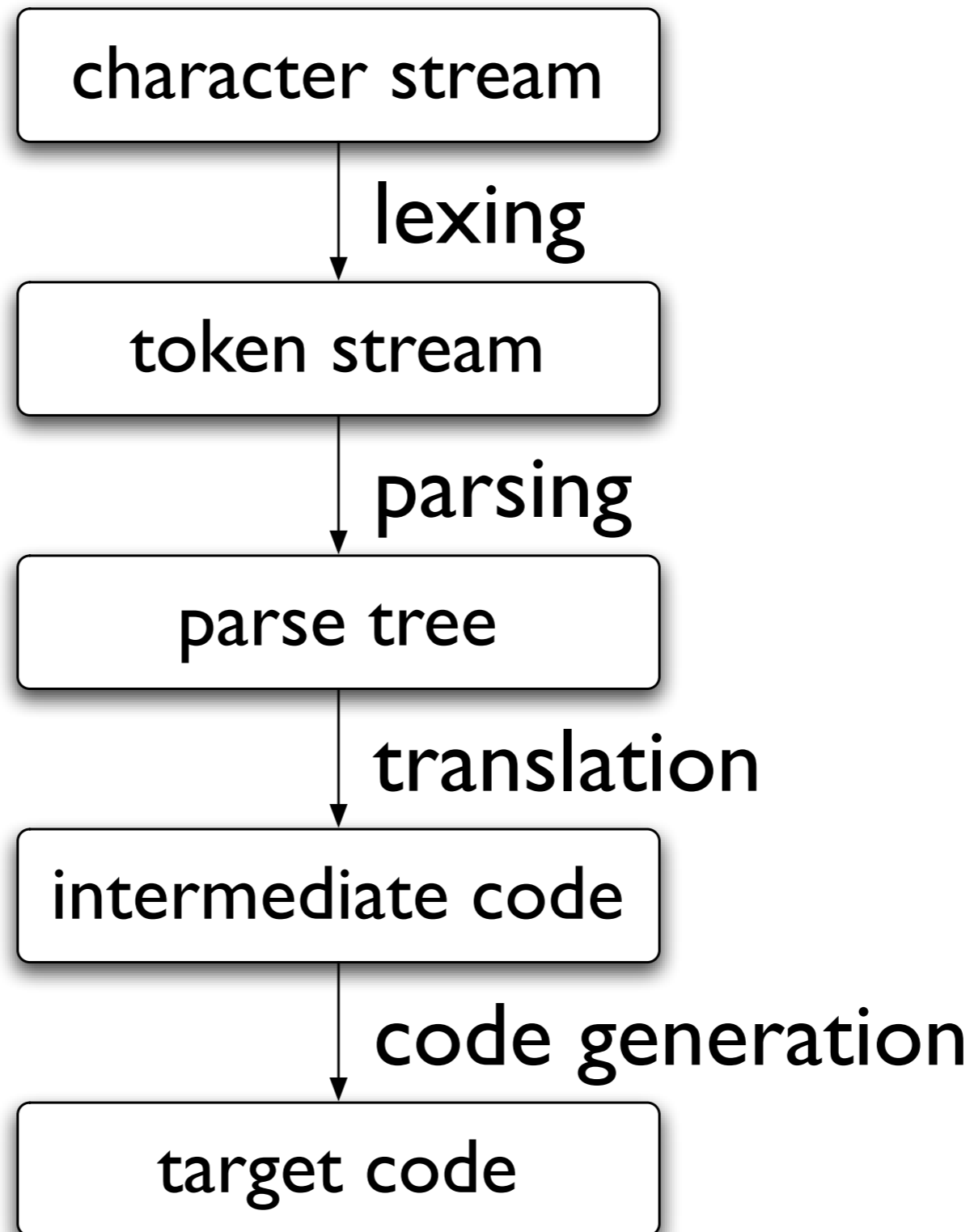


Optimising Compilers

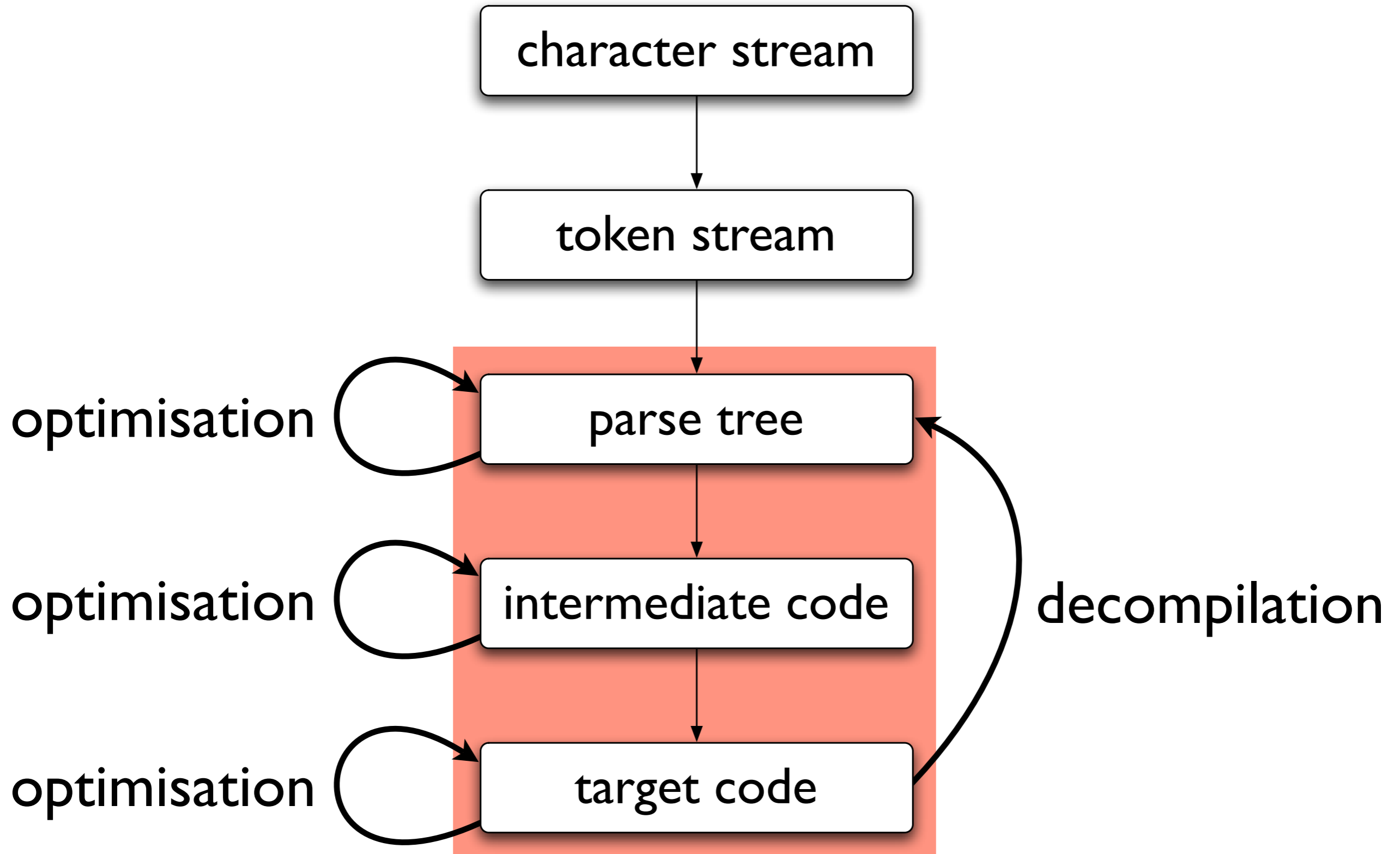
Computer Science Tripos Part II - Lent 2006

Tom Stuart

A non-optimising compiler



An optimising compiler



Optimisation

(really “amelioration”!)

Good humans write simple, maintainable, *general* code.

Compilers should then *remove unused generality*,
and hence hopefully make the code:

- Smaller
- Faster
- Cheaper (e.g. lower power consumption)

Optimisation

=

Analysis

+

Transformation

Analysis + Transformation

- Transformation *does something dangerous*.
- Analysis determines *whether it's safe*.

Analysis + Transformation

- An analysis shows that your program has some property...
- ...and the transformation is designed to be safe for all programs with that property...
- ...so it's safe to do the transformation.

Analysis + Transformation

```
int main(void)
{
    return 42;
}
```

```
int f(int x)
{
    return x * 2;
}
```


Analysis + Transformation

```
int main(void)
{
    return 42;
}
```



Analysis + Transformation

```
int main(void)
{
    return f(21);
}
```

```
int f(int x)
{
    return x * 2;
}
```

Analysis + Transformation

```
int main(void)
{
    return f(21);
}
```



Analysis + Transformation

```
while (i <= k*2) {  
    j = j * i;  
    i = i + 1;  
}
```

Analysis + Transformation

```
int t = k * 2;  
while (i <= t) {  
    j = j * i;  
    i = i + 1;  
}
```



Analysis + Transformation

```
while (i <= k*2) {  
    k = k - i;  
    i = i + 1;  
}
```

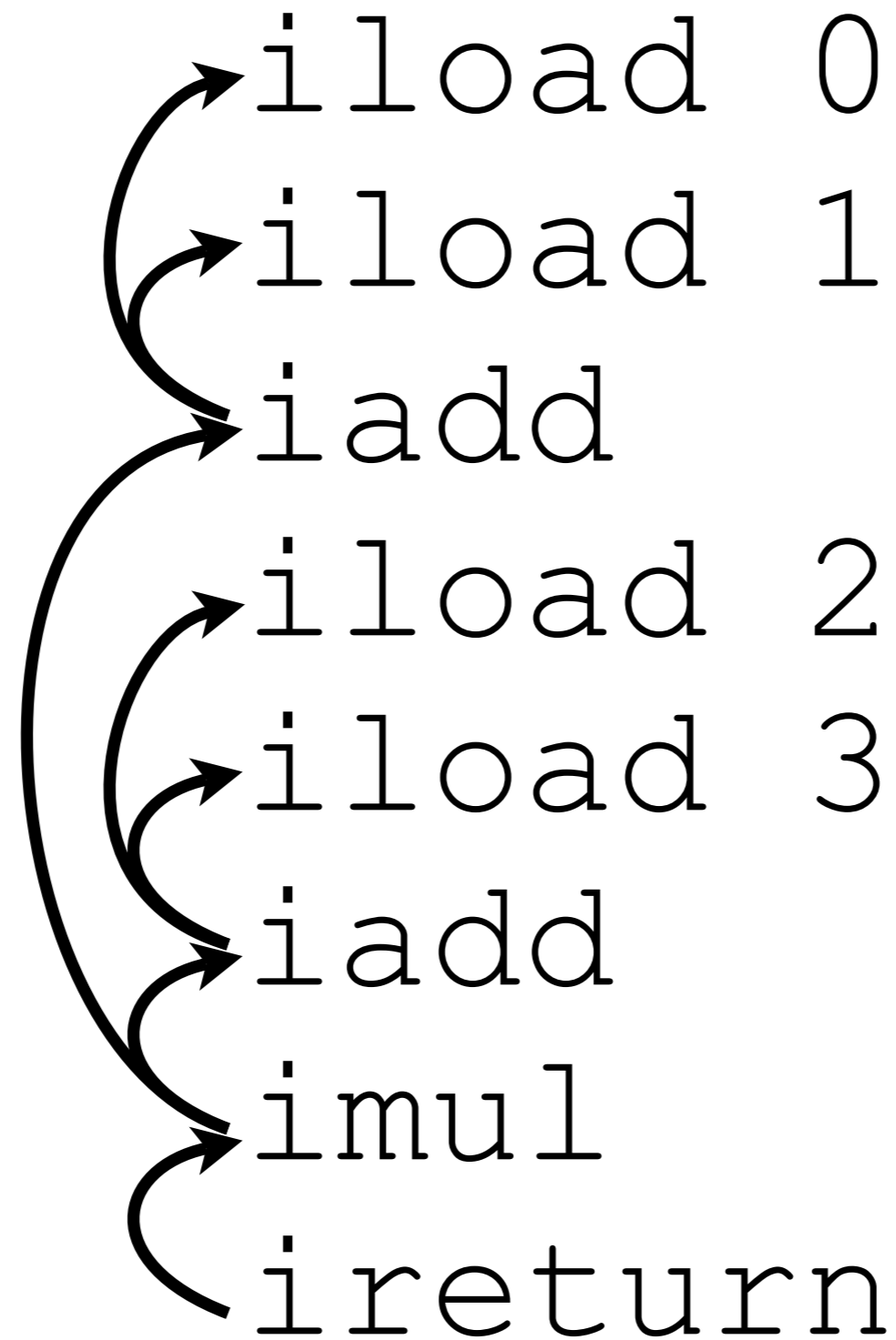
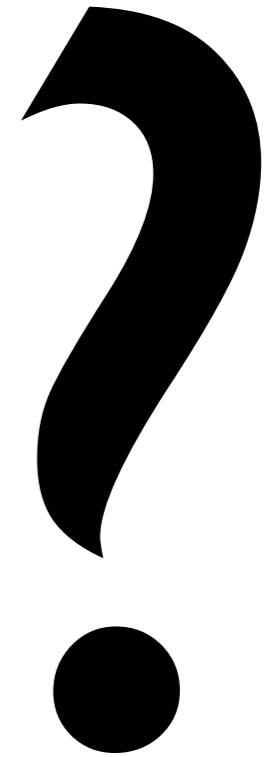
Analysis + Transformation

```
int t = k * 2;  
while (i <= t) {  
    k = k - i;  
    i = i + 1;  
}
```



Stack-oriented code

```
    iload 0  
    iload 1  
    iadd  
    iload 2  
    iload 3  
    iadd  
    imul  
    ireturn
```

A diagram illustrating stack-oriented code. The code consists of nine instructions: `iload 0`, `iload 1`, `iadd`, `iload 2`, `iload 3`, `iadd`, `imul`, and `ireturn`. On the left side, several curved arrows point from the left towards the instructions, indicating stack operations. One arrow points to `iload 0`, another to `iload 1`, and a third to `iadd`. A larger arrow points from the left towards the `iload 2` and `iload 3` instructions, and another points to the `iadd` instruction below them. A final arrow points to `imul`. The `ireturn` instruction has no arrow pointing to it.

3-address code

```
MOV  t32, arg1
MOV  t33, arg2
ADD  t34, t32, t33
MOV  t35, arg3
MOV  t36, arg4
ADD  t37, t35, t36
MUL  res1, t34, t37
EXIT
```

The diagram illustrates the data flow between registers in the provided 3-address code. Each register name is enclosed in a colored box, and lines connect these boxes across different instructions to show dependencies. The connections are as follows:

- A red line connects the `t32` box in the first instruction to the `t32` box in the third instruction.
- A green line connects the `t33` box in the second instruction to the `t33` box in the third instruction.
- A blue line connects the `t34` box in the third instruction to the `t34` box in the eighth instruction.
- A purple line connects the `t35` box in the fourth instruction to the `t35` box in the sixth instruction.
- An orange line connects the `t36` box in the fifth instruction to the `t36` box in the sixth instruction.
- A yellow line connects the `t37` box in the sixth instruction to the `t37` box in the seventh instruction.

C into 3-address code

```
int fact (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fact (n-1);  
    }  
}
```

C into 3-address code

```
ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
lab1: MOV res1, #1
EXIT
```

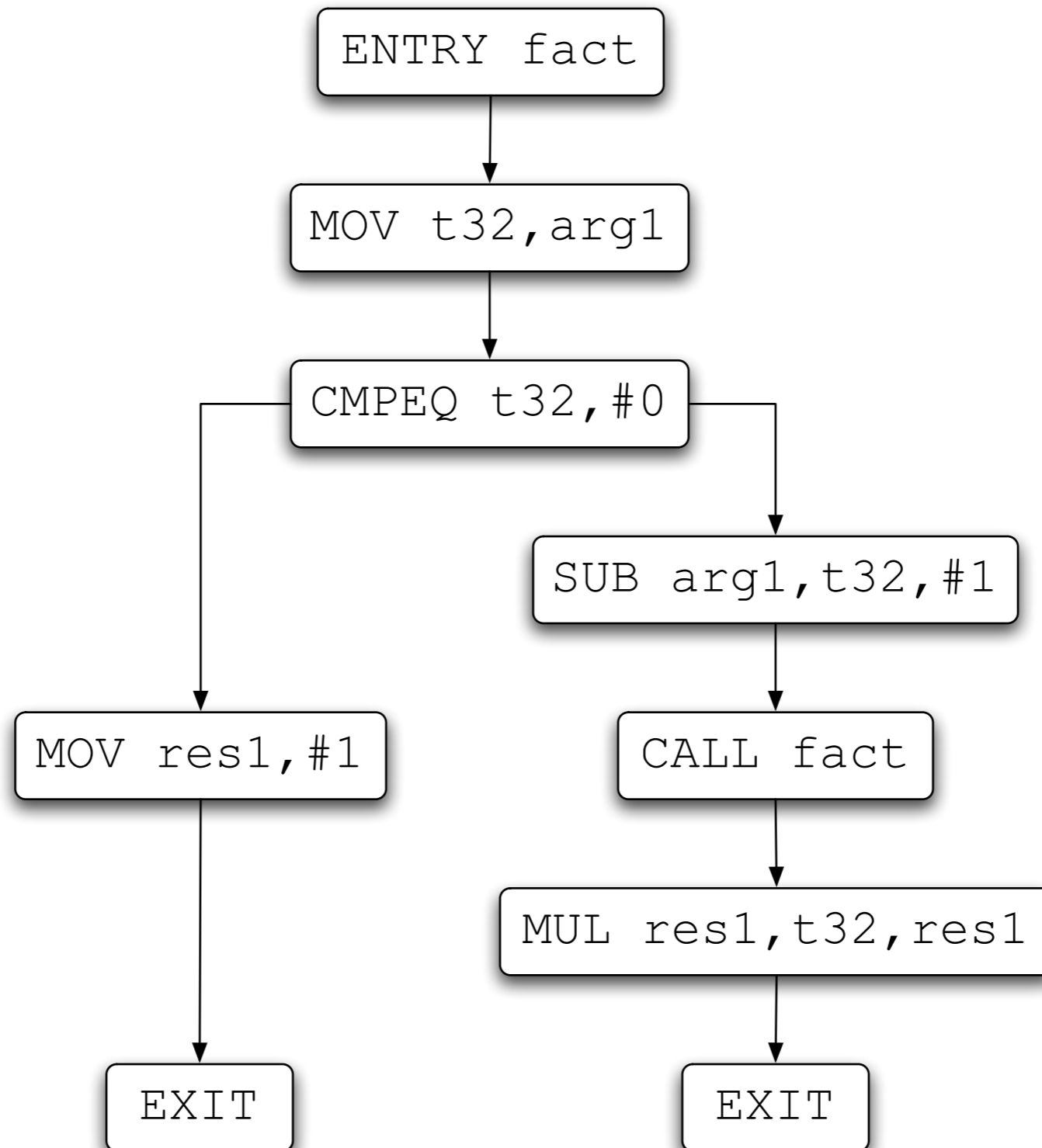
Flowgraphs

- A graph representation of a program
- Each node stores 3-address instruction(s)
- Each edge represents (potential) control flow:

$$pred(n) = \{n' \mid (n', n) \in edges(G)\}$$

$$succ(n) = \{n' \mid (n, n') \in edges(G)\}$$

Flowgraphs

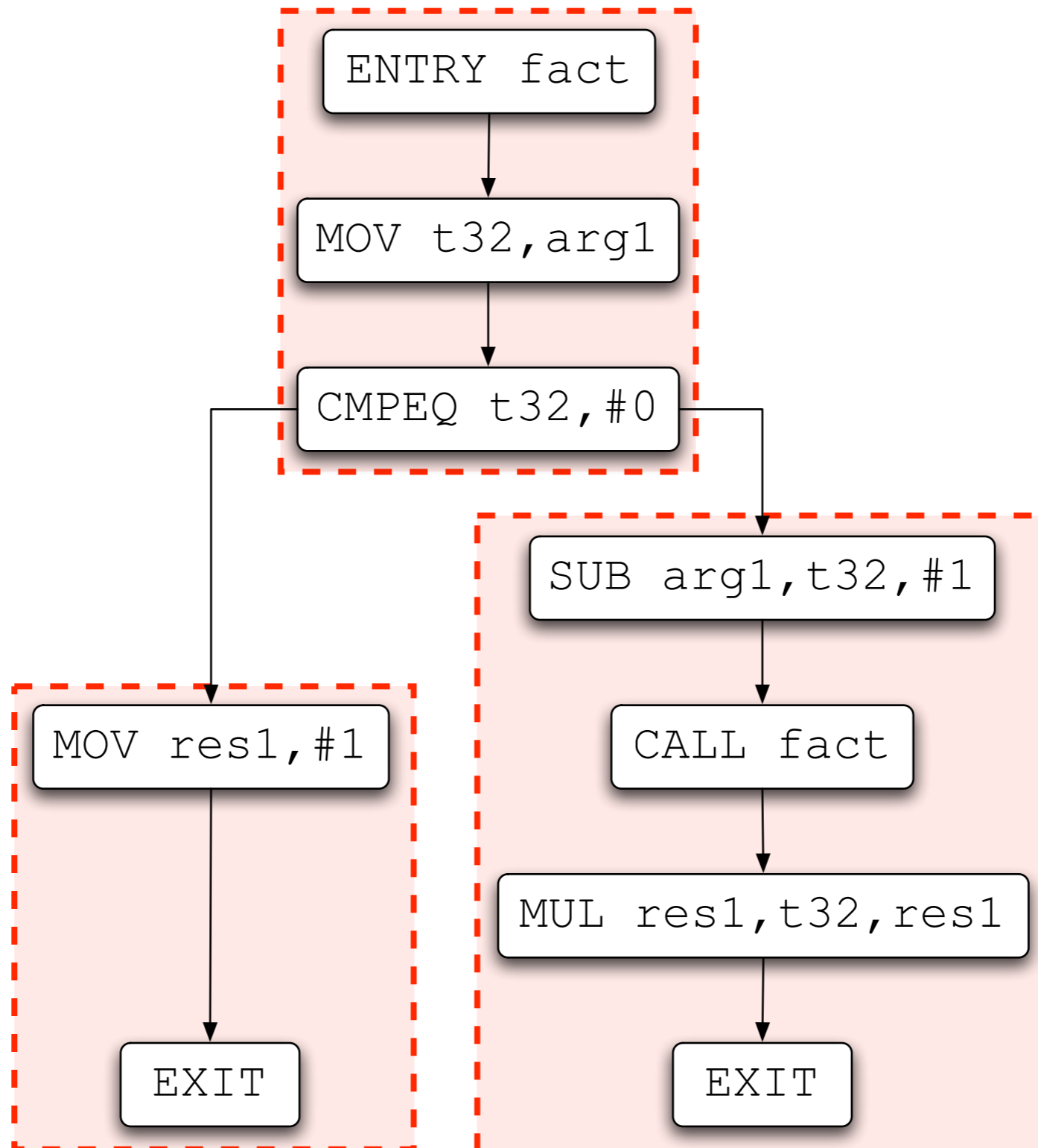


Basic blocks

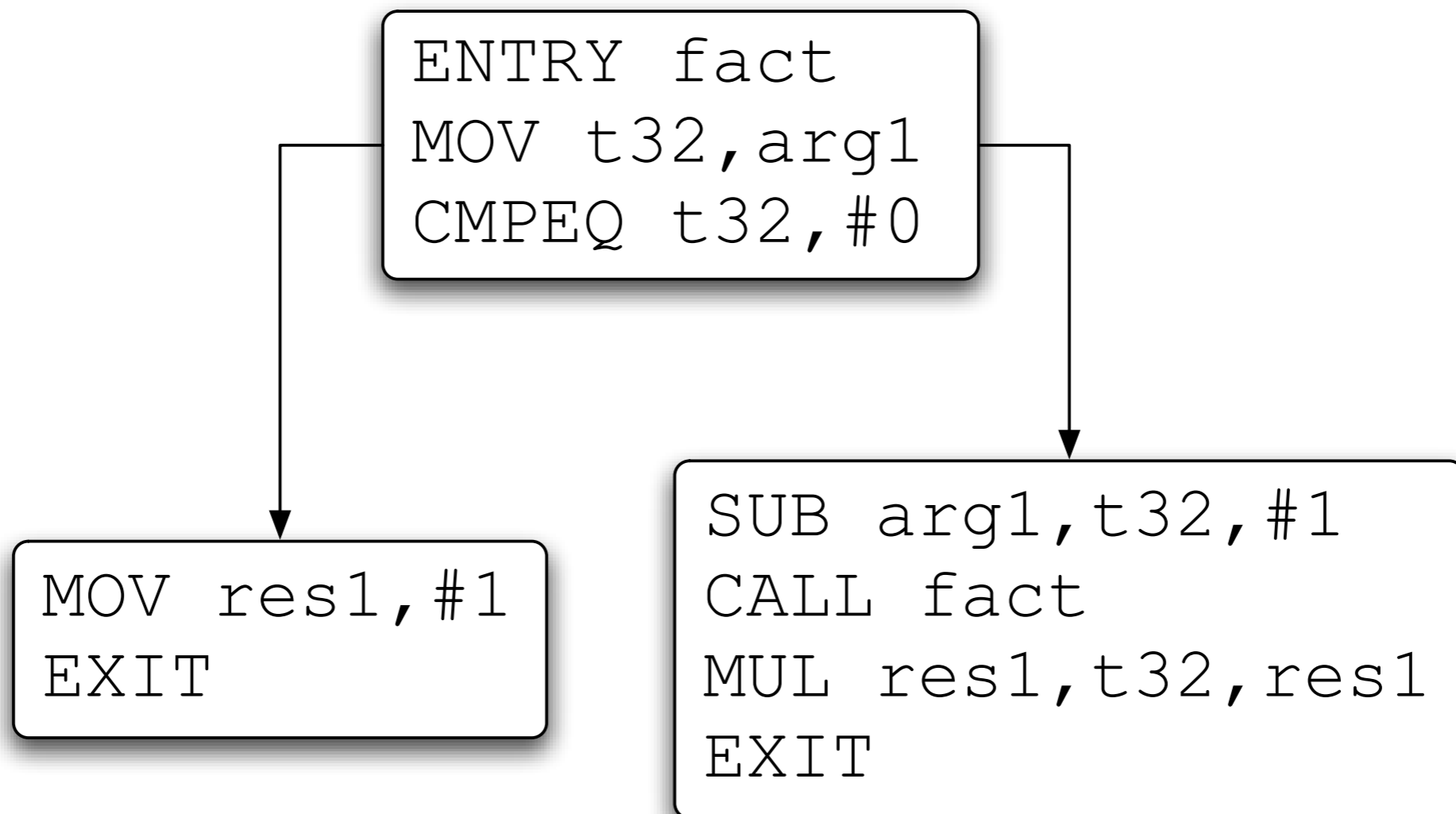
A maximal sequence of instructions n_1, \dots, n_k which have

- exactly one predecessor (except possibly for n_1)
- exactly one successor (except possibly for n_k)

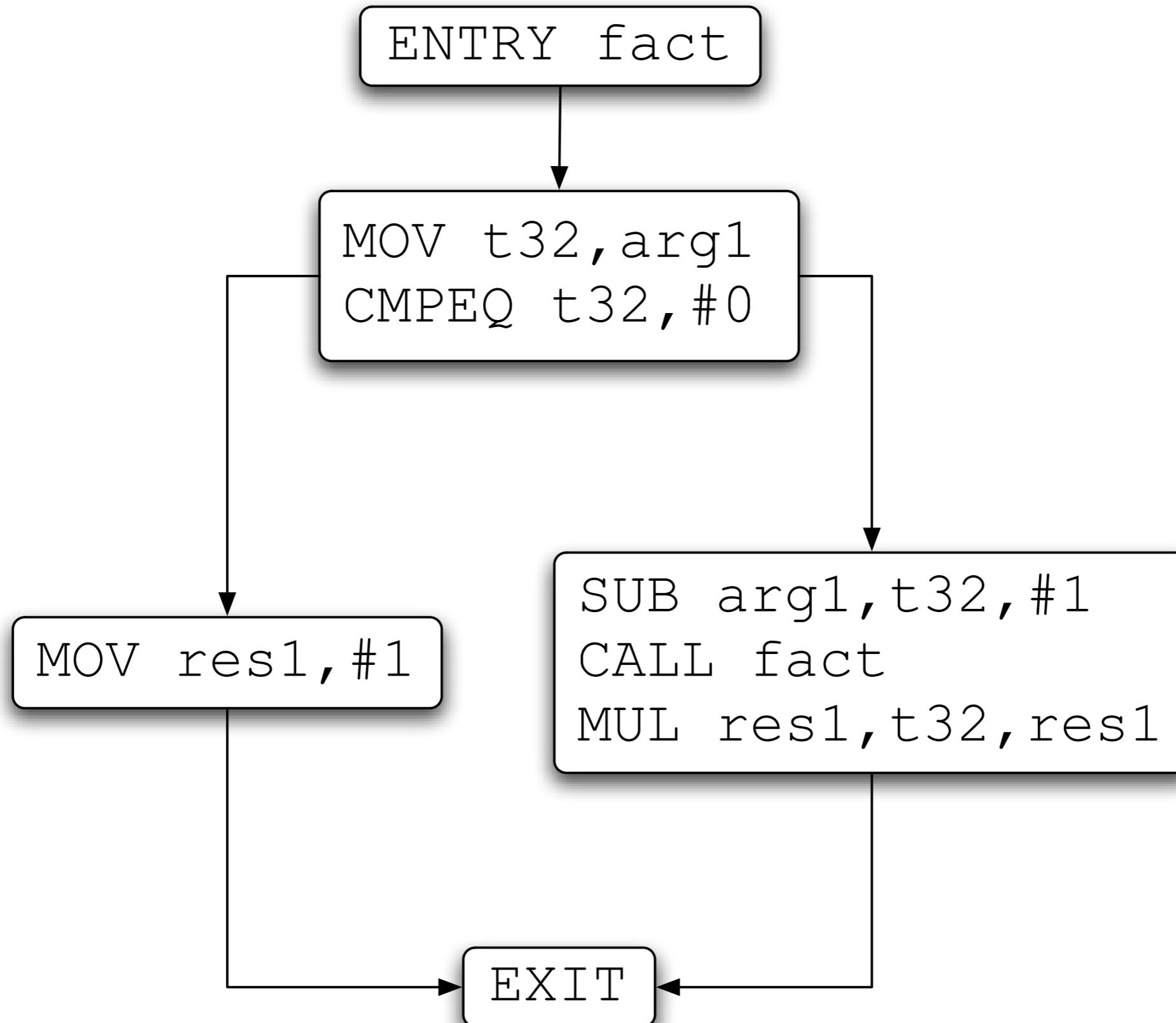
Basic blocks



Basic blocks



Basic blocks



Basic blocks

A basic block doesn't contain any interesting control flow.

Basic blocks

Reduce time and space requirements
for analysis algorithms

by calculating and storing data flow information

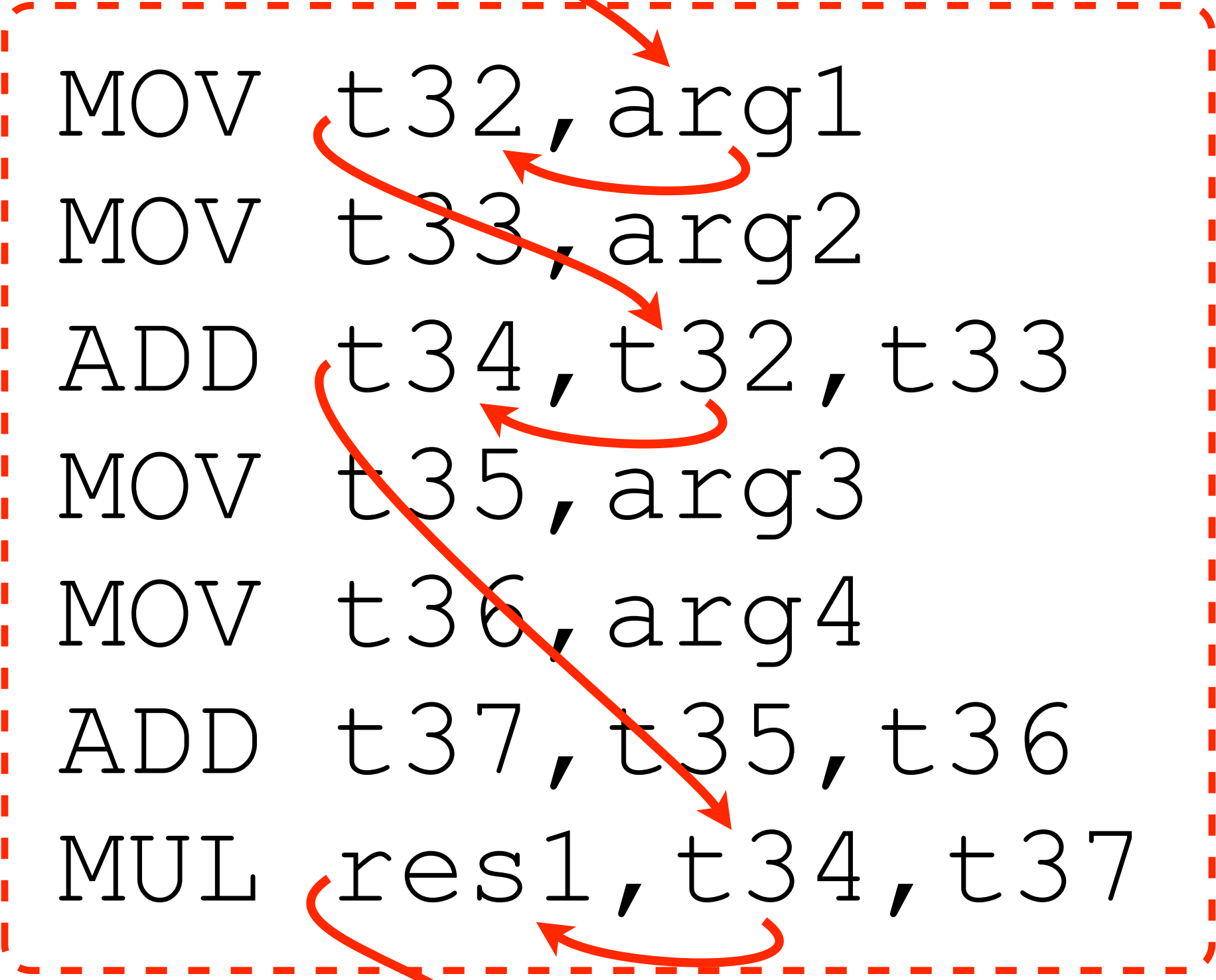
once per block

(and recomputing within a block if required)

instead of

once per instruction.

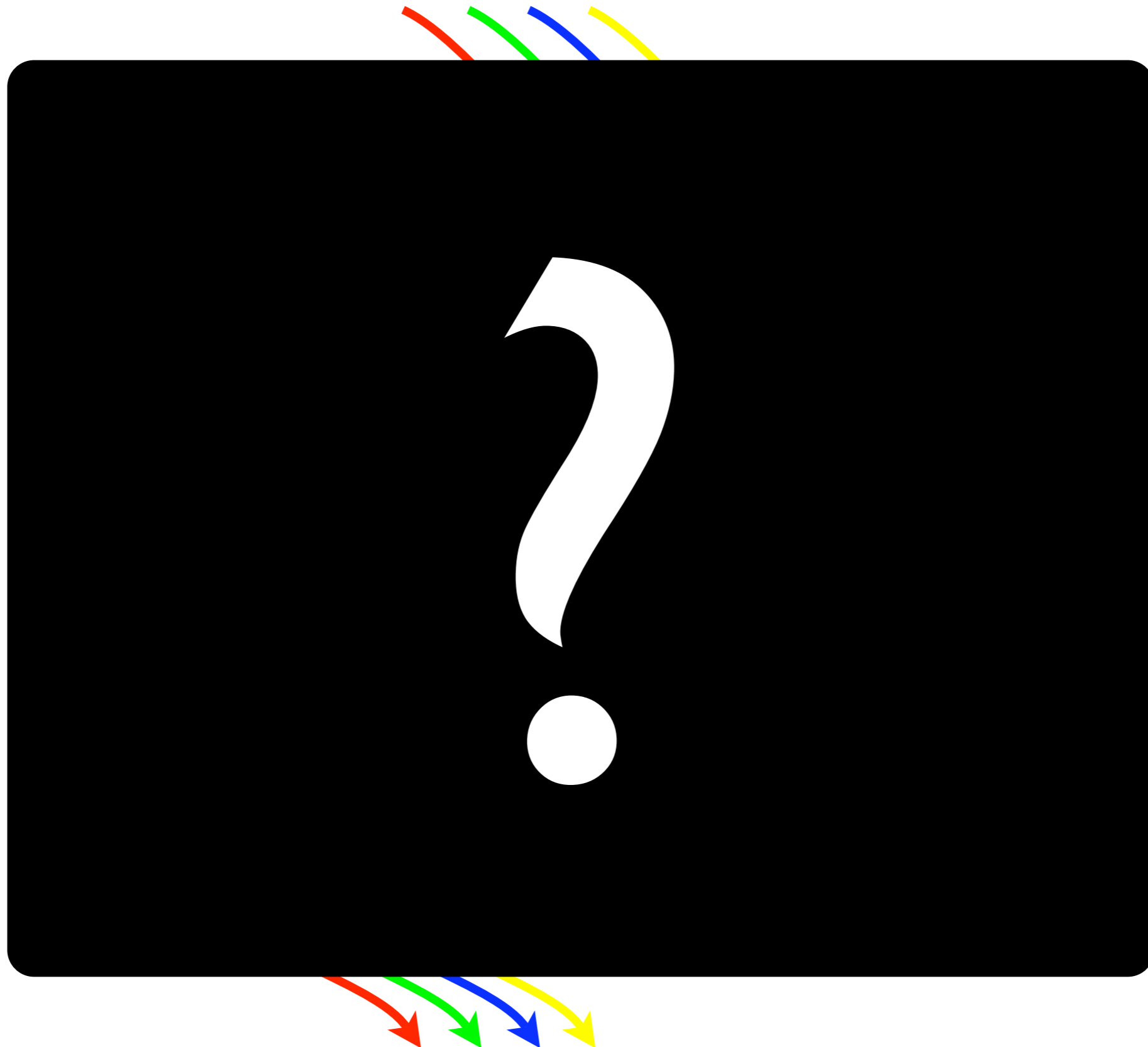
Basic blocks



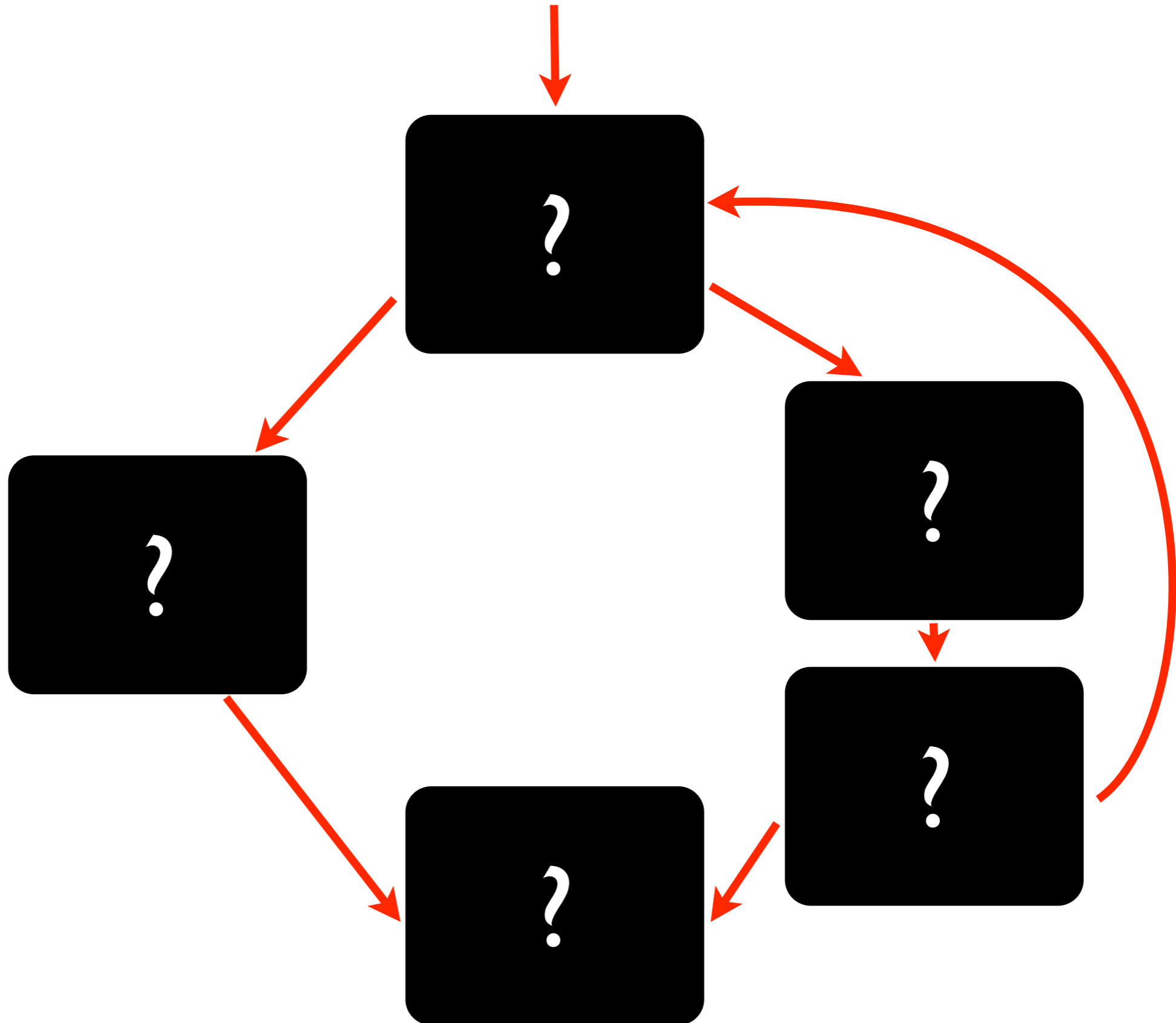
A diagram illustrating a basic block of code. The code is enclosed in a dashed red box. Red arrows indicate control flow: one arrow points from above into the top of the box, another points from the top of the box to the right, a third points from the right of the box to the bottom, and a fourth points from the bottom of the box to the left. The code consists of seven lines of assembly-like instructions:

```
MOV t32, arg1  
MOV t33, arg2  
ADD t34, t32, t33  
MOV t35, arg3  
MOV t36, arg4  
ADD t37, t35, t36  
MUL res1, t34, t37
```

Basic blocks



Basic blocks



Types of analysis

(and hence optimisation)

Scope:

- Within basic blocks (“local” / “peephole”)
- Between basic blocks (“global” / “intra-procedural”)
 - e.g. live variable analysis, available expressions
- Whole program (“inter-procedural”)
 - e.g. unreachable-procedure elimination

Peephole optimisation

```
ADD t32, arg1, #1
MOV r0, r1
MOV r1, r0
MUL t33, r0, t32
```

↓

```
ADD t32, arg1, #1
MOV r0, r1
MUL t33, r0, t32
```

matches

replace

```
MOV x, y
MOV y, x
```

with

```
MOV x, y
```


Types of analysis

(and hence optimisation)

Type of information:

- Control flow
 - Discovering control structure (basic blocks, loops, calls between procedures)
- Data flow
 - Discovering data flow structure (variable uses, expression evaluation)

Finding basic blocks

1. Find all the instructions which are *leaders*:
 - the first instruction is a leader;
 - the target of any branch is a leader; and
 - any instruction immediately following a branch is a leader.
2. For each leader, its basic block consists of itself and all instructions up to the next leader.

Finding basic blocks

```
ENTRY fact
```

```
MOV t32, arg1
```

```
CMPEQ t32, #0, lab1
```

```
SUB arg1, t32, #1
```

```
CALL fact
```

```
MUL res1, t32, res1
```

```
EXIT
```

```
lab1: MOV res1, #1
```

```
EXIT
```

Finding basic blocks

```
ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
lab1: MOV res1, #1
EXIT
```

Summary

- Structure of an optimising compiler
- Why optimise?
- Optimisation = Analysis + Transformation
- 3-address code
- Flowgraphs
- Basic blocks
- Types of analysis
- Locating basic blocks